CHAPTER - 2

Basic Concepts

The area of program slicing has been enriched over the last two decades by contributions from several researchers. Since its inception many new concepts have been introduced to speed up the process of slicing, the precision of slices, and to make program slicing usable in different applications. The technique of program slicing has been extended to sequential, non-sequential multi-procedure programs, as well as object-oriented programs, distributed programs, concurrent programs, web applications, aspect-oriented programs, and UML (Unified Modeling Language) models. Also, these slicing techniques have been applied to diverse problem areas. Some of the important applications of slicing have been discussed in Section 2.1.6 of this chapter.

This chapter provides a detailed description of the background theory used in the rest of this thesis. For the sake of conciseness and to avoid trivial discussions, we do not aim to present a detailed description of the background theory. Instead, we provide a brief introduction aimed at highlighting the basic concepts and definitions that contribute to the understanding of this thesis. The basic concepts and definitions are used in subsequent chapters of this thesis.

Section 2.1.1 contains some definitions which will be used later in our dynamic slicing algorithms. Section 2.1.2 describes some intermediate program representation concepts which are commonly used in slicing techniques. Section 2.1.3 discusses program dependence graph. Section 2.1.4 discusses some basic concepts on slicing which will help in understanding our dynamic slicing algorithms. Section 2.1.5 briefly discusses the concepts of precision and correctness of slice. Some important applications are given in Section 2.1.6. Section 2.2.1 discusses the SOA layers. Section 2.2.2 briefly discusses the SOA architectural principles. Section 2.3 discusses some basic concepts of software testing. Section 2.4 discusses SOA testing challenges. Section 2.5 discusses testing SOA functional requirements. Finally, Section 2.6 summarizes this chapter.
2.1 Program Slicing

Program slicing is a technique that extracts the statements of a program $P$ that (potentially) affect the computation of a variable of $P$ at a specific location of $P$. The tuple consisting of the program, the variable, and the program location is referred to as the slicing criterion. The slicing criterion is sometimes extended to a set of variables. The parts of the program selected by program slicing for a slicing criterion constitute the program slice with respect to that criterion.

The original concept of program slicing was proposed by Mark Weiser [140] as another approach for debugging sequential programs. Weiser claims that program slicing corresponds to the mental abstraction performed by programmers while debugging programs. In 1986, Weiser and Lyle [141] performed a set of experiments that showed that some variations of slices significantly reduced the amount of time needed to debug programs. Program slicing is an active area of research, and this is reflected in various research works [115, 119, 146], Tip’s survey [69], and in overviews [56, 147].

2.1.1 Preliminary Concepts and Definitions

In this section, we discuss a few concepts, notations, and terminologies that are used later in this thesis. The existing program slicing literature shows a wide variation, diversity, disparity and disagreement in the notations used in program slicing. We explain our usage here because of this lack of common consensus. The usage presented here does not come from any single source but rather is a personal blending of ideas from many sources. In the following definitions and throughout the thesis, we use the terms node and vertex interchangeably.

**Definition 2.1 Directed Graph:** A directed graph $G$ is a pair $(N, E)$ where $N$ is a finite non-empty set of nodes, and $E \subseteq N \times N$ is a set of directed edges between the nodes.

Let $G = (N, E)$ be a directed graph. If $(x, y)$ is an edge of $G$, then $x$ is called a predecessor of $y$ and $y$ is called a successor of $x$. The number of predecessors of a node is its in-degree, and the number of successors of a node is its out-degree. A directed path (or path) from a node $x_1$ to a node $x_k$ in a graph $G = (N, E)$ is a sequence of nodes $x_1, x_2, \ldots, x_k$ such that $(x_i, x_{i+1}) \in E$ for every $i, 1 \leq i \leq k - 1$. In this thesis, we use the words graph and directed graph interchangeably.

**Definition 2.2 Flow Graph:** A flow graph is a quadruple $(N, E, \text{Start}, \text{Stop})$ where $(N, E)$ is a graph, $\text{Start} \in N$ is a distinguished node of in-degree 0 called the start node, $\text{Stop} \in N$ is
a distinguished node of out-degree 0 called the stop node, there is a path from Start to every other node in the graph, and there is a path from every other node in the graph to Stop.

Definition 2.3 Dominance: If $x$ and $y$ are two nodes in a flow graph then $x$ dominates $y$ iff every path from Start to $y$ passes through $x$. $y$ post-dominates $x$ iff every path from $x$ to Stop passes through $y$.

Let $x$ and $y$ be nodes in a flow graph $G$. Node $x$ is said to be immediate post-dominator of node $y$ iff $x$ is a post-dominator of $y$, $x \neq y$ and each post-dominator $z \neq x$ of $y$ post-dominates $x$. The post-dominator tree of a flow graph $G$ is the tree that consists of the nodes of $G$, has the root Stop, and has an edge $(x, y)$ iff $x$ is the immediate post-dominator of $y.$

Consider the flow graph of the example program of Fig. 2.1, which is given in Fig. 2.2. In the flow graph, each of the nodes 4, 5 and 6 dominate 7. Node 8 does not dominate node 10. Node 10 post-dominates each of the nodes 4, 5, 6, 7, 8 and 9. Node 9 post-dominates node 8. Node 9 post-dominates none of the nodes 4, 5, 6, 7, 10, 11 and 12. Node 6 is the immediate post-dominator of node 5. Node 10 is the immediate post-dominator of node 7.

2.1.2 Intermediate Program Representation

Various types of program representation schemes exist which include high-level source code, pseudo-code, a set of machine instructions in a computer’s memory, a flow chart, and others. The purpose of each of these representations depends upon the exact context of use. Different representations may be required to facilitate human readability, annotation for verifiability, and transformation for running a program on platforms such as multiprocessors and distributed computers, etc. In the context of program slicing, program representations are used to support efficient automation of slicing.

For a very simple program, a slice for any given slicing criterion can be determined manually. But with increasing size and complexity of the programs, automatic slice computation
is essential. Current automated slicing techniques require that the information available in a source code form of the program to be sliced be transformed into some mathematical representation during the slicing process. Various representation schemes have resulted from the search for ever more complete and efficient slicing techniques.

In the following, we present a few basic concepts associated with intermediate program representations that are used later in this thesis. A common corner stone for most of the slicing algorithms is that programs are represented by a directed graph which captures data and control dependencies.

The standard intermediate program representations for slicing are the Control Flow Graph (CFG) and the Program Dependence Graph (PDG). CFG based slicing algorithms usually address data flow analysis, whereas PDG slicing algorithms address the graph-reachability problems. Generally, PDG slicing is much more efficient than CFG slicing.

It is important to note that there is no single correct way of constructing, say, a Control Flow Graph (CFG) or Program Dependence Graph (PDG), nor there is a set of rules or standards which must be available for slicing. Researchers present different techniques according to the needs of the problem at hand. Nevertheless, there is a general agreement as to the class information to be contained in each type of program representation for the purpose of slicing. The various representations shown here are illustrative of these general agreements but are not necessarily faithful to any single researcher’s style.

2.1.2.1 Control Flow Graph

The Control Flow Graph (CFG) is an intermediate representation for programs that are useful for data flow analysis and in many code optimizing transformations such as common subexpression elimination, copy propagation, dead code elimination, induction variables elimination, reduction in induction variable’s strength and code motion [9, 70]. But, the CFG does not give any information regarding data dependency between two nodes.

Definition 2.4 Control Flow Graph: Let the set $N$ represents the set of statements of a program $P$. The Control Flow Graph of program $P$ is the flow graph $G = (N_1, E, Start, Stop)$ where $N_1 = N \cup \{Start, Stop\}$. An edge $(m, n) \in E$ indicates the possible flow of control from node $m$ to node $n$.

Note that the existence of an edge $(x, y)$ in the control flow graph means that control must transfer from $x$ to $y$ during program execution. Fig. 2.2 represents the CFG of the example program given in Fig. 2.1. The CFG of a program $P$ models the branching structures of the program, and it can be built while parsing the source code using algorithms that have linear
2.1.2.2 Data Dependence Graph

The CFG of a program represents the flow of control through the program. However, the concept that is often more useful in program analysis is the flow of data through a program. Data flow describes the flow of the values of variables from the points of their definitions to the points where their values are used.

Dependencies are relationships among program statements. There are two major type, control and data dependencies. Other dependencies do exist in the context of concurrent/distributed programs that we would discuss in the chapters to follow. In a program, these two types create dependencies among the statements of a program.

Definition 2.5 Data Dependence: Let $G$ be the CFG of a program $P$. A node $n$ is said to be data dependent on a node $m$ if there exits a variable $var$ of the program $P$ such that the following hold:

(i) the node $m$ defines $var$,

(ii) the node $n$ uses $var$, and

(iii) there exists a directed path from $m$ to $n$ along which there is no intervening definition of $var$.

Consider the example program given in Fig. 2.1 and its CFG in Fig. 2.2. Node 8 has data dependence on each of the nodes 5, 6, and 8. Node 11 has data dependence on node 5. Note that node 11 has data dependence on none of the nodes 6 and 8.
The term *reaching definition* is used to mean that a value defined at a node may be used at another node [9]. That is, node $x$ is a reaching definition for a node $y$ iff $y$ is data dependent on $x$. A data dependence from node $x$ to node $y$ indicates that a value computed at $x$ may be used at $y$ under some path through the control flow graph. A dependence from $x$ to $y$ is a conservative approximation which says that under some conditions a value computed at $x$ may be used at $y$.

**Definition 2.6 Data Dependence Graph:** The data dependence graph of a program $P$ is the graph $G = (N, E)$, where each node $n \in N$ represents a statement of the program $P$ and $(x, y) \in E$ iff $x$ is data dependent on $y$.

2.1.2.3 Control Dependence Graph

Ferrante et al. [70] introduced the notion of *control dependencies* to represent the relations between program entities arising due to control flow.

**Definition 2.7 Control Dependence:** Let $G$ be the CFG of a program $P$. Let $x$ and $y$ be two nodes in $G$. Node $y$ is *control dependent* on a node $x$ if the following hold:

(i) there exists a directed path $D$ from $x$ to $y$,

(ii) $y$ post-dominates every $z$ in $D$ (excluding $x$ and $y$), and

(iii) $y$ does not post-dominate $x$.

Let $x$ and $y$ be two nodes in the CFG $G$ of a program $P$. If $y$ is *control dependent* on $x$, then $x$ must have multiple successors in $G$. Conversely, if $x$ has multiple successors, then, at least, one of its successors must be control dependent on it.

Consider the example program given in Fig. 2.1 and its CFG in Fig. 2.2. Each of the nodes 8 and 9 is control dependent on the node 7. Note that the node 7 has two successor nodes 8 and 10, and the node 8 has control dependence on 7.

**Definition 2.8 Control Dependence Graph:** The control dependence graph of a program $P$ is the graph $G = (N, E)$, where each node $n \in N$ represents a statement of the program $P$ and $(x, y) \in E$ iff $x$ is control dependent on $y$. 

35
2.1.3 Program Dependence Graph

Ferrante et al. [70] presented a new mechanism of program representation called Program Dependence Graph (PDG). Unlike the flow graphs, an important feature of PDG is that it explicitly represents both control and data dependencies in a single program representation. A PDG models a program as a graph in which the nodes represent the statements and the edges represent inter-statement data or control dependencies.

**Definition 2.9 Program Dependence Graph (PDG):** The program dependence graph $G$ of a program $P$ is the graph $G = (N, E)$, where each node $n \in N$ represents a statement of the program $P$. The graph contains two kinds of directed edges: control dependence edges and data dependence edges. A control (or data) dependence edge $(m, n)$ indicates that $n$ is control (or data) dependent on $m$.

![PDG Example](image)

**FIGURE 2.3:** The PDG of the example program given in Fig. 2.1

Note that the PDG of a program $P$ is the union of a pair of graphs: the data dependence graph of $P$ and the control dependence graph of $P$. Consider the program given in Fig. 2.1. Its PDG is given in Fig. 2.3. In Fig. 2.3, the nodes of the graph represent the statements of the example program of Fig. 2.1. The solid edges represent the control dependencies and the dotted edges represent the data dependencies.

The program dependence graph of a program $P$ can be built from its control flow graph in $O(n^2)$ time, where $n$ is the number of nodes in the control flow graph [70].

2.1.3.1 System Dependence Graph

The PDG of a program combines the control dependencies and the data dependencies into a common framework. The PDG has been found to be suitable for intra-procedural slicing. However, it cannot handle procedure calls. Horwitz et al. [90] enhanced the PDG representation to facilitate inter-procedural slicing. They introduced the System Dependence
Graph (SDG) representation which models the main program together with all associated procedures. The SDG, an extension of the PDG, models programs in a language with the following properties [90]:

- A complete program consists of the main program and a collection of auxiliary procedures.
- Procedures end with return statements. A return statement does not include a list of variables.
- Parameters are passed by value-results.

The SDG is very similar to the PDG. Indeed, a PDG of the main program is a subgraph of the SDG. In other words, for a program without procedure calls, the PDG and SDG are identical. The technique for constructing a SDG consists of first constructing a PDG for every procedure, including the main procedure, and then adding auxiliary dependence edges which link the various subgraphs together. This results in a program representation which includes the information necessary for slicing across procedure boundaries.

A SDG includes several types of nodes to model procedure calls and parameter passing:

- **Call-site nodes** represent the procedure call statements in a program.
- **Actual-in and actual-out nodes** represent the input and output parameters at call sites. They are control dependent on the call-site nodes.
- **Formal-in and Formal-out nodes** represent the input and output parameters at the called procedures. They are control dependent on the procedure’s entry node.

Control dependence edges and data dependence edges are used to link the individual PDGs in an SDG. The additional edges that are used to link the PDGs together as follows:

- **Call edges** link the call-site nodes with the procedure entry nodes.
- **Parameter-in edges** link the actual-in nodes with the formal-in nodes.
- **Parameter-out edges** link the formal-out nodes with the actual-out nodes.

Finally, summary edges are added to represent the transitive dependencies that arise due to procedure calls. A summary edge is added from an actual-in node A to an actual-out node
FIGURE 2.4: An example program consisting of a main program and two procedures

\[ \text{main( )} \]
\[ \text{int } s, i; \]
\[ \{ \]
\[ \quad s = 0; \quad \text{void add(int a, int b)} \]
\[ \quad i = 1; \quad \{ \]
\[ \quad \text{while (} i < 10 \text{) do} \]
\[ \quad \{ \]
\[ \quad \quad \text{add}(s, i); \]
\[ \quad \quad \text{inc}(i); \]
\[ \quad \}
\[ \quad \text{write}(s); \]
\[ \} \]

\[ \text{void inc(int z)} \]
\[ \{ \]
\[ \quad \text{add}(z, 1); \]
\[ \quad \text{return; } \]
\[ \} \]

\[ \text{FIGURE 2.4: An example program consisting of a main program and two procedures} \]

If the value associated with the actual-in node \( A \) affects the value associated with the actual-out node \( B \), due to the transitive flow of dependence. The transitive flow of dependence may be caused by data dependencies, control dependencies or both. Fig. 2.5 represents the SDG of the example program given in Fig. 2.4.

2.1.4 Basic Concepts in Static Slicing

In this section, we first discuss some work on static slicing of object-oriented programs which are essential for understanding static slicing of service-oriented software. In the next chapter, we discuss how these basic slicing techniques have subsequently been extended to handle dynamic slicing of service-oriented software.

The intermediate program representation is analyzed to compute a static slice. Horwitz et al. [90] developed system dependence graph (SDG) as an intermediate program representation and proposed a two-phase graph reachability algorithm on the SDG to compute inter-procedural slice. The first pass of the inter-procedural slicing algorithm traverses backward along all the edges of the SDG except parameter-out edges, and marks those vertices reached. The second pass traverses backward from all vertices marked during the first pass along all edges except call and parameter-in edges and marks the reached vertices. The slice is the union of the vertices marked during pass one and pass two.

Larson and Harrold [132] enhanced the SDG [90] to represent object-oriented programs. Their SDG successfully represents object-oriented features such as method calls, inheritance, and polymorphism. After constructing the SDG for a complete object-oriented program, they have used the two-pass graph reachability algorithm [90] for computing static slices. One limitation of this approach is that it fails to consider the fact that in different method invocations, the data members used by the methods might belong to different objects. So, the resulting data dependencies become imprecise. A second limitation of the approach is that it does not handle cases in which an object is used as a parameter or as a data member.
FIGURE 2.5: The SDG of the example program given in Fig. 2.4

of another object.

2.1.5 Precision and Correctness of Slice

Let $P$ be a program, and $S$ be a static slice of $P$ with respect to a slicing criterion $C$. In the original definition of Weiser [140], the reduced program $S$ is required to be an executable program and its behavior with respect to the slicing criterion must be same as the original program $P$. A slice $S$ of $P$ with respect to a slicing criterion $C$ is statement-minimal if no other slice of $P$ with respect to the slicing criterion has fewer statements than $S$. Weiser [140] has shown that the problem of computing statement-minimal slices is undecidable.

Another common definition of a static slice is the following: a slice $S$ of a program
$P$ with respect to a slicing criterion $C$ is a subset of the statements of the program which directly or indirectly affect the slicing criterion [70, 90, 177]. Note, that such a slice need not be executable. Unless specified otherwise, we follow this definition of a slice throughout the discussion in the thesis.

Let $G_C$ be the control flow graph (CFG) of a program $P$. In all the existing program slicing frameworks, for each statement $s$ in the program $P$, two sets are maintained. One set contains the variable names used at $s$ and the other set contains the variable names defined at $s$. The inter-statement dependencies in the program $P$ are captured using the CFG $G_C$ and the variable names in these two sets, for each statement $s$.

Note that statement 4 of the example program given in Fig. 2.6 uses the variable $m$. Though statement 4 assigns the value zero ($m - m$) to the variable $z$, it has a dependence on statement 1 in the program slicing frameworks since statement 1 is a reaching definition of the variable $m$ for the statement 4.

It is, therefore, reasonable to define the precision of a dynamic slice in the existing program slicing frameworks as follows. A dynamic slice is said to be precise if it contains only those statements that actually affect the slicing criterion in the particular execution.

Note that a precise dynamic slice need not be a statement-minimal slice. Consider the example program given in Fig. 2.6. For any input value of the variable $m$, the statement-minimal slice with respect to the slicing criterion $< 4, z >$ should be empty set as $z$ is always assigned the value $0 = m - m$. In the existing program slicing frameworks, the precise slice for the slicing criterion is certainly $\{1\}$ as statement 1 is a reaching definition of the variable $m$ for the statement 4.

A slice is said to be correct if it contains all the statements that affect the slicing criterion. A slice is said to be incorrect if it fails to contain some statements that affect the slicing criterion. Note that the whole program is always a correct slice of any slicing criterion. A correct slice is imprecise if it contains, at least, one statement that does not affect the slicing criterion.
2.1.6 Applications of Program Slicing

This section describes the use of program slicing techniques in various applications. In trying to use the basic slicing concepts in diverse domains, several variations of the notions of program slicing as described in Section 1.1 are developed. The program slicing technique was originally developed to realize automated static code decomposition tools. The primary objective of those tools was to aid program debugging. From this modest beginning, the use of program slicing techniques have now ramified into a powerful set of tools for use in such diverse applications as program understanding, program verification, automated computation of several software engineering metrics, software maintenance and testing, functional cohesion, dead code elimination, reverse engineering, parallelization of sequential programs, software portability, reusable component generation, compiler optimization, program integration, showing differences between programs, software quality assurance, software refactoring etc. [71, 78, 79, 91, 127, 148, 149, 261]. A comprehensive study on the applications of program slicing is made by Binkley and Gallagher [57] and Lucia [128]. In the following, we briefly discuss some of these applications of program slicing.

2.1.6.1 Differencing

Programmers often face the problem of finding the differences between two programs. Algorithms for finding textual differences between programs are often insufficient. Program slicing can be used to identify semantic differences between two programs [58]. There are two related differencing problems:

1. Find all the components of two programs that have different behavior.

2. Produce a program that captures the semantic differences between two programs.

For programs old and new, a straightforward solution to problem 1 is obtained by comparing the backward slices of the vertices in old and new’s dependence graphs $G_{old}$ and $G_{new}$. Here, the backward slice is computed with respect to a given slicing criterion. Components whose vertices in $G_{new}$ and $G_{old}$ have isomorphic slices have the same behavior in old and new; thus the set of vertices from $G_{new}$ for which there is no vertex in $G_{old}$ with an isomorphic slice approximates the set of components new with changed behavior.

A solution to the second differencing problem is obtained by taking the backward slice with respect to the set of affected points (i.e., the vertices in $G_{new}$ with different behavior than in $G_{old}$). For programs with method calls, two modifications are necessary: First, interprocedural slicing techniques are required to be used to ensure that the resulting program
slice is executable. Second, this solution is overly pessimistic: consider a component \( c \) in method \( P \) that is invoked from two call-sites \( c_1 \) and \( c_2 \). If \( c \) is identified as an affected point by a forward slice that enters \( P \) through \( c_1 \) then we will include \( c_1 \) but not \( c_2 \) in the program that captures the differences. However, the backward slice with respect to \( c \) would include both \( c_1 \) and \( c_2 \).

### 2.1.6.2 Debugging

Finding bugs in a program is always difficult. The process of finding a bug usually involves running the program over and over, learning more and narrowing down the search each time, until the bug is finally located. In distributed systems, the problem is more difficult because of control and data dependencies and also communication dependencies that might lead to additional, often non-repeatable bugs. Program slicing was originally proposed by observing the operation typically carried out by programmers while debugging a piece of code. Programmers mentally slice a code while debugging it. Even after several advancements to the basic slicing techniques, program debugging remains the main application area of slicing techniques. Debugging can be a difficult task when one is confronted with a large program and little clues regarding the location of a bug. During debugging, a programmer usually has a test case in mind which causes the program to fail. Program slicing is useful for debugging, because it potentially allows one to ignore many statements in the process of localizing the bug [127, 140]. If a program computes an erroneous value for a variable \( x \) only the statements in the slice with respect to \( x \) have (possibly) contributed to the computation of that value; all statements which are not in the slice can safely be ignored. A program slicer that is integrated into the debugger can be very useful in discovering the reason for the error by visualizing control and data dependencies and by highlighting the statements that are part of the slice.

Several variants of program slicing have been developed to further assist the programmer in debugging: program dicing [127] identifies statements those are likely to contain bugs by using information that some variables fail some tests while others pass all tests. Slices can be combined with each other in different ways: for example, the intersection of two slices contains all statements that lead to an error in both test cases; the intersection of slice \( a \) with the complement of slice \( b \) excludes from slice \( a \) all statements that do not lead to an error in the second test case. Another variant of program slicing is program chopping [195]. It identifies statements that lie between two points \( a \) and \( b \) in the program which will be affected by a change made at \( a \). This can be useful when a change at \( a \) causes an incorrect result at \( b \). Debugging should be focussed on the statements between \( a \) and \( b \) that transmit the change of \( a \) to \( b \).
2.1.6.3 Software Maintenance

Software maintenance is a costly process because each modification to a program must take into account many complex dependence relationships in the existing software. The main challenges in effective software maintenance, are to understand various dependencies in an existing software and to make changes to the existing software without introducing new bugs. One of the problems in software maintenance is that of the ripple effect, i.e., whether a code change in a program will affect the behavior of other codes of the program. To avoid this problem, it is necessary to know which variables in which statements will be affected by a modified variable, and which variables in which statements will affect a modified variable during software maintenance. The needs can be satisfied by slicing the program being maintained [78].

2.1.6.4 Testing

Software maintainers often carry out regression testing. Regression testing essentially implies retesting software after modification [59, 194, 252]. Even after the smallest change to a piece of code, extensive tests may be necessary which might involve running a large number of test cases to rule out any unwanted behavior arising due to the change. While decomposition slicing eliminates the need for regression testing on the complement, there may still be a substantial number of tests to be run on the dependent, independent and changed parts. Slicing can be used to reduce the number of these tests.

Suppose a program modification only changes the value of the variable $x$ at program point $p$. If the forward slice with respect to $x$ and $p$ is disjoint from the coverage of regression test $t$, then the test $t$ does not have to be rerun. Suppose a coverage tool reveals that a use of variable $x$ at program point $p$ has not been tested. What input data is required in order to cover $p$? The answer lies in the backward slice of $x$ with respect to $p$. A lot of work has also been reported in order to test programs incrementally, to simplify testing, to apply program slicing to regression testing, to partition testing, and to test path selection [36, 72, 79, 91, 150, 252].

2.1.6.5 Program Integration

Programmers frequently face the problem of integrating several variants of a base program. Of course, the first step is to look for textual differences. More sophisticated techniques for this purpose have now become available. Semantic-based program integration is a technique that attempts to create an integrated program that incorporates the changed computations of the variants as well as the computations of the base program that are preserved in all variants.
Horwitz et al. [79] presented an algorithm for semantic-based program integration that creates the integrated program by merging certain program slices of the variants. Their integration algorithm takes as input three programs, Base, A and B, where A and B are variants of Base. The integrated program is produced by (1) building graphs that represent Base, A and B, (2) combining program slices of the program dependence graphs of Base, A and B to form a merged graph, (3) testing the merged graph for certain interference criteria, and (4) reconstructing a program from the merged graph. Yang extends the algorithm of Horwitz et al. [79] for detecting program components with equivalent behaviors and it can accommodate semantics-preserving transformations.

2.1.6.6 Functional Cohesion

Cohesion measures the relatedness of the code of some component [37]. A highly cohesive software component is one that has only one function that is indivisible. Bieman and Ott [27] define data slices to consist of data tokens (instead of statements). Data tokens may be variables of constant definitions and references. Data slices are computed for each output of a procedure (e.g., output to a file, output parameter, assignment to a global variable). The tokens that are common to more than one data slice are the connections between the slices. They are called glue. The glue binds the slices together. The tokens that are in every data slice of a function are called super-glue. Strong functional cohesion can be expressed as the ratio of super-glue tokens to the total number of tokens in the slice, whereas weak functional cohesion may be seen as the ratio of glue tokens to the total number of tokens. The adhesiveness of a token is another measure expressing how many slices are glued together by that token.

2.1.6.7 Software Quality Assurance

Software quality assurance auditors have to locate safety critical code and to ascertain its effect throughout the system. Program slicing can be used to locate all code that influences the values of variables that might be part of a safety critical component. But beforehand these critical components have to be determined by domain experts.

One possible way to assure high quality is to make the system redundant [57, 194]. If two output values are critical, then these output values should be computed independently. They should not depend on the same internal functions, since the same error might manifest in both output values, in the same way, thereby hiding the error. One technique to defend against such errors is to use functional diversity, where multiple algorithms are used for the same purpose. Thus, the critical output values depend on different internal functions. Program
slicing can be used to determine the logical independence of the slices computed for the two output values.

### 2.1.6.8 Parallelization

Program slicing can be used to decompose a conventional program into substantially independent slices for assignment to separate processors as a way to parallelize the program. A goal of such parallelization is to determine slices with almost no overlap. Assuming that a combination slicer-compiler could produce a sliced executable code suitable for a parallel machine, an issue of some complexity is the problem of reconstructing the original behavior by splicing the results of the separate outputs of different slices. Such a technique is investigated in [149].

### 2.1.6.9 Anomaly Detection

Static slicing methods can detect dead code, i.e., statements which cannot affect any output of the program. Often such statements are not executable because of the presence of a bug. Static slicing can also be used to determine uninitialized variables which are used in expressions, another symptom of an error in the program.

### 2.1.6.10 Program Specialization and Reuse

Executed slices can be thought of as specialized programs. Slices can be used to make code reuse more efficient [133]. Instead of reusing the entire package, a slice can be used to identify only those parts that are really needed. Slicing isolates computational threads, which helps in identifying logical components. The threads can be extracted and either replaced or used to create new programs.

### 2.1.6.11 Reverse Engineering

Reverse engineering concerns the problem of comprehending the current design of a program and the way this design differs from the original design [194]. This involves abstracting out of the source code, the design decisions and the rationale for the initial development (design recognition) and understanding the algorithms (algorithm recognition).

Program slicing provides a tool set for this type of re-abstraction [28]. For example, a program can be displayed as a lattice of slices ordered by the is-a-slice-of relation. Com-
paring the original lattice and the lattice after (years of) maintenance can guide an engineer towards places where reverse engineering energy should be spent. Because slices are not necessarily contiguous blocks of code they are well suited for identifying differences in algorithms that may span multiple blocks or procedures.

2.1.6.12 Other Applications of Program Slicing

As mentioned earlier, program slicing methods have been used in several other applications such as compiler optimizations, detecting dead code, software portability analysis, program understanding, program verification, software refactoring, measuring class cohesion, etc. These applications can be found in [37, 78, 189].

2.2 Service-Oriented Architecture (SOA)

Service-oriented architecture (SOA) is an architectural style for designing and developing distributed systems. As organizations expand their systems to cross organizational boundaries, testers will have to test the use of SOA as an architectural style in these systems. They may need to test their systems in such a way that qualities are met without sacrificing the loosely coupled, stateless, standards-based nature of the relationship between service consumers and service providers characteristics that have made SOA a worthwhile technology to adopt.

In this section, we discuss the SOA layers and their architectural principles which are important for understanding the design and development of SOA-based software.

2.2.1 Service-Oriented Architecture (SOA) Layers

There are many research articles that provide a reference architecture or layered approach for systems that use a service-orientation approach [12, 30]. These layers facilitate separation of concerns and designers have a set of architectural decisions that need to be made in each layer. Fig. 2.7 shows the typical layers of a service-oriented system that are primarily functional in nature.
The service-oriented system includes the following layers [184]:

- **Presentation Layer**: The benefit of creating a presentation layer is decoupling the client-side presentation implementation from the service implementation to allow each to change independently. This layer is generally not the focus of service-oriented design.

- **Business Process Layer**: The services provided in the services layer are often composed into workflows to assist in the development of applications. This provides flexibility to change the workflows as business processes change.

- **Services Layer**: Services reside in this layer. The invocation of these services can be
determined at design time or bound dynamically through a service registry at runtime. These services are broken into additional layers to assist in the composition of services into complete business processes. The additional layers include:

- **Utility-based services:** This service provides utility-based functions such as notification, logging, and exception handling. These operations are largely agnostic to business processes and can, therefore, be reused in multiple business processes.

- **Entity-based services:** These services operate on a set of business entities (or data entities). These operations are largely agnostic to business processes and can, therefore, be reused in multiple business processes.

- **Task-based services:** These services are “business services with a functional boundary directly associated with a specific parent business task or process” [184].

• **Enterprise Layer:** These components contain code that specifically fulfills service needs or code that accesses the functionality in operational systems. “These special components are managed, and governed set of enterprise assets that are funded by the enterprise or the business unit level. This layer typically uses container-based technologies such as application servers to implement the components, workload management, higher availability, and load balancing” [12].

• **Operational Systems Layer:** This layer consists of existing custom-built, commercial, external systems or some combination of these. Careful analysis of these systems needs to be completed to determine the operations that should be exposed as services. These services are considered to have enterprise-wide utility.

### 2.2.2 SOA Architectural Principles

SOA architectural principles are general guidelines for designing service-oriented systems. These principles are ideally enabled by the decisions taken during designing the system. The service-oriented architecture at a higher level of abstraction, consists of four major types of elements: service consumers, SOA infrastructure, service interfaces, and service implementation, as shown in Fig. 2.8. Erl [219] defined additional principles for service design. The principles in this section are similar, but they are applied to the full architecture of the service-oriented system: the integration of services (interface and implementation), service consumers, and the SOA infrastructure.
SOA Architects often find themselves in conflicting situations. On one hand, they have business/mission goals and QoS requirements driving the architecture of a system. On the other hand, there are principles of service-orientation that influence the architecture of a system and impact a system’s QoS attributes. It is at this intersection of these two sets of quality attributes where conflicts arise and an SOA architect needs to make suitable decisions. Therefore, the responsibility of the SOA architect is to try to apply each principle in the context of the business goals of the system and to balance trade-offs and architectural decisions in order to meet the system’s business goals [184].

1. Interoperability
One of the key enablers of widespread SOA adoption is standardization at multiple levels, as shown in Fig. 2.9. Standardization in service-oriented systems has multiple advantages including tool support and leverage of third-party system components that, in the end, can lead to shorter development times [184].

The WS-* base stack (HTTP, XML, SOAP, and WSDL) is fairly stable and has large tool support. For example, there are multiple tools that will take a web service description language (WSDL) document as input and produce all the code necessary to invoke the associated service. However, beyond the base stack, it is not that straightforward because of the over-abundance of standards.
There are currently over 100+ WS-* standards produced by organizations such as OA-SIS [176] and W3C [238] in areas that include business process specification, composition, messaging, reliable messaging, transaction management, security, and management. Some of these standards are complementary and some have competitive behavior specifications. Additionally, many of these standards have extensions, that can be interpreted in different ways.

The web services interoperability organization (WS-I) is an organization chartered to promote web services interoperability across platforms, applications, and programming languages [244]. WS-I has profiles for the basic stack and for security to provide clarifications, refinements, interpretations, and amplifications in areas of the standards that are subject to multiple interpretations. There are also tools to check that XML artifact (e.g., a WSDL file) and actual messages being exchanged are in conformance with the profiles. The WS-I tools are especially useful in cases in which WSDL and XML files are automatically generated and may not conform to the assumptions of the development and deployment environment, e.g., different XML schema versions, different namespaces, malformed XML, etc. The tools that facilitate the usage of these

FIGURE 2.9: Web service standardized protocol stack and WS-* extensions
standards will be one criterion that an architect uses to select between competing standards.

2. **Loose Coupling**
Loose coupling is also one of the key goals for SOA adoption. Service consumers should be allowed to select technologies that best fit their organizational context and policies. The SOA architect should define how messages will be exchanged, how messages will be designed, and how services will be described. SOA implementations that are built using web services use XML for message exchange and WSDL and stable standards such as SOAP and HTTP for communications. These XML technologies make loose coupling possible in service-oriented environments.

3. **Reusability** The goal of increasing reuse is mostly associated with service reusability. Services are reusable because they represent self-contained functionality that can be used in multiple business processes. SOA architect can abstract utility services and may reuse it.

4. **Composibility** A composition may rely heavily on infrastructure such as an orchestration engine for the choreography of services based on business workflows of BPEL.

5. **Discoverability** In a service-oriented environment, services are created and published in a UDDI that is accessible to service consumers. Ideally, service consumers can query this service registry looking for services that satisfy desired business functionality.

### 2.3 Software Testing

When a program is developed as an implementation of any algorithm or logic, the developers are always doubtful about its performance and correctness. The developers must have the confidence that the software achieves a certain level of quality. Software testing can be appropriately used to ensure the quality of the software to a certain level. A quality software should be correct. A software can only be correct, iff it computes results for the entire domain of input, and all the results it computes are specified. Thus, the software requires exhaustive testing to validate the input domain. A software testing approach can only suggest the presence of faults and cannot highlight their absence if it is not exhaustive. According to Karner et al. [105], exhaustive testing of the software is not possible for the following reasons:

- the input domain is too large, for example, to test the greatest among two numbers, the input domain can be any number n that belongs to the Integer set, $I$. 

51
there are too many possible input paths to test, so the difficulties alluded to by this assertion are exacerbated by the fact that certain execution paths in a program could be infeasible [171, 225].

design and specifications can change during software development and are thus difficult to test, this is because software testing is an algorithmically unsolvable problem and specification errors cause major design errors [39]. According to Manna et al. [158], it is not possible to surely know the correctness of the specifications.

Hamlet et al. [94] have formally stated the goals of a software testing methodology and Morell et al. [159] have highlighted its limitations. Young et al. [256] observed that there was always a trade-off between exhaustive testing and computational cost because the presence of defects was always undecidable. Therefore, no testing technique can be completely accurate and generic to all the programs. Even though the testing process is challenged with many limitations, but the consistent application of a testing technique in an intelligent manner can ensure an acceptable level of software quality. Therefore, testing is an important phase in software development life cycle. This phase incurs 60% of the total cost of the software. Therefore, it becomes highly essential to devise proper testing techniques in order to design the test cases so that the software can be tested properly. Testing strategies are based on verification and validation. The static techniques available for a testing map to the verification process without executing the code, whereas the dynamic testing techniques map to the validation process by executing the code.

Fig. 2.10 shows the hierarchical decomposition of the testing strategies along with their association with different test adequacy criteria. The decomposition shown in Fig. 2.10 follows the definitions given in [39, 265]. The execution-based testing techniques are decomposed into either program-based, specification-based, or combined as shown in Fig. 2.10. The chosen adequacy criterion C determines the types of test cases that belong to the test suite, T. A program-based testing approach creates T by analyzing the source code of a program, P, based upon its structure and attributes. A specification-based testing technique creates the desired test suite from the functional and/or non-functional requirements for P. Whereas, combined testing uses both program-based and specification-based testing approaches to generate T. Based on the kind of testing strategy that is followed to create T, the test cases are categorized into three types:

- **Black-box**, test cases that are created without knowledge of P’s source code.
- **White-box**, test cases that are created considering the entire source code of P.
- **Grey Box**, test cases only consider a portion of P’s source code.
Fig. 2.10 also shows the association of these testing strategies with the corresponding test adequacy criteria. A structure-based criterion requires T to satisfy exercising of certain control structures and variables within P, such as statement coverage, branch coverage, condition coverage, path coverage, etc. Therefore, structure-based test adequacy criterion requires program-based testing. Fault-based test adequacy criterion ensures that the types of faults that are commonly introduced into P by the programmers are revealed by T. Finally, error-based testing approaches rely upon the fact that T does not deviate from the specifications in any way. Therefore, error-based adequacy criteria motivate for specification-based testing approaches.

2.3.1 Test, Test Case, and Test suite

In the context of software testing, test and test cases are often used interchangeably. A test case consists of an initial state, inputs, and expected outputs. The state refers to pre-conditions, if any, that is circumstances that hold prior to the test execution. Each test case is also identified by a unique identification number. The process of testing is to check whether the inputs yield the expected outputs or not. The test case is said to fail if the actual output differs from the expected output. If the test case fails, then it requires debugging to reach the cause of this failure. A set of these test cases designated to test an application is called test suite. A test suite may be segregated into a set of successful and unsuccessful test cases.
2.3.2 Execution-based Software Testing

Fig. 2.11 shows the process of execution-based software testing. In this Figure, the rectangles denote the testing activities and the parallelograms denote the outcome of these activities. The testing process starts with a system under test, P, and a test adequacy criterion, C, as input. The testing process is iterative and stops iterating when the test cases in test suite T satisfy the adequacy criterion C and assure some level of confidence in the quality of P [265]. However, the testing process can also stop in case of deadline misses or budget overrun. The testing process can also halt if the tester gets an intuition of achieving some acceptable level of quality. Even if the testing process stops or meets with C, it does not guarantee that all the defects have been revealed by the test cases. Therefore, testers set many adequacy criteria (different coverage criteria, software metrics, etc.) to build the confidence on quality. The test adequacy criteria depend on the chosen program representation and definition of some quality parameters that T should satisfy. The test specification stage evaluates P in the context
of chosen criterion C in order to construct an adequate test suite T. Once the test case
descriptions for P has been generated, the test case generation phase begins. A lot of
different techniques and tools have been proposed to manually or automatically gener-
ate the test cases. After the generation of the test cases, the execution of the test cases
starts. Once again, the execution of the tests within T can be performed in a manual
or automated fashion. The results from the execution of the test cases are analyzed in
terms of coverage. Thus, iterative testing of P continues throughout its initial develop-
ment. However, it is also important to continue testing after P undergoes changes in
the maintenance phase.

Regression testing is an important software maintenance activity carried out to ensure
that the changes made does not adversely affect the correctness of P. All of the previ-
ously mentioned stages iteratively continue for the regression testing process based on
the existing test cases (new test cases may be added) and the adequacy measurements
defined for these tests [14, 39, 105, 134, 172, 197].

2.4 SOA Testing Challenges

SOA testing challenges are broadly classified according to the functional and non-
functional requirements. The functional requirement testing challenges are associated
with common components, web service, web service composition, and competency of
the tester. The non-functional requirement testing challenges are availability, reliabil-
ity, security, performance and usability. Further, the SOA-specific testing challenges
include interoperability, agility, adaptability, reusability, and governance.

At a higher level, testing SOA implementation (web service) is similar to testing
traditional systems using conventional testing techniques. But at a lower level, it
becomes a complex task. A tester may test its functional and non-functional re-
quirements first. The functional requirement testing of the service-oriented system
must check whether business requirements documented in service level agreement
(SLA)/specification document is traceable up to the SOA deployment level. This ver-
ification includes testing of common components, composite service, and end-to-end
business process. The non-functional requirements testing checks various quality of
service (QoS) attributes like availability, reliability, agility, interoperability, security,
performance, and adaptability etc. A service that does not balance QoS level up to
expected level can be considered as a violation of the SLA. In this section, we discuss
SOA testing challenges according to the classification shown in Fig. 2.12.
2.4.1 Functional Requirement Testing Challenges

In this section, we discuss the major factors and associated challenges that seriously affect the SOA testing process from the perspective of common components (CC) like infrastructure components, web service, composite services, and tester capability.

- **Common Components (CC) Testing Challenges**
  
  - **Highly complex configuration**
    The identified SOA infrastructure components are web services, web servers, universal description discovery and integration (UDDI), enterprise service bus (ESB) and web service clients more specifically a web browser, typically these are vendor supported software. All these have their own configuration for installation, updating and upgradation. Such components must be configured correctly for testing which otherwise may lead to incorrect results. Sometimes these components have multiple inconsistent configurations which complicate the testing effort.
  
  - **Inconsistent SOA implementation platforms**
    Web services are commonly implemented through J2EE and .net technology platforms. Both are inconsistent with respect to standards, protocols, interfaces, service implementation etc. For example, J2EE involves web service description language (WSDL), simple object access protocol (SOAP), and HTTP protocol stack layers while .net involves WSDL, UDDI, SOAP, HTTP etc. These inconsistencies in standards or protocol stack layers lead to misunderstanding of SOA concepts and thus increase problems in testing SOA.
  
  - **Lack of testing environment**
    A testing environment mirrors deployment environment and includes test-
ing tools, test case, and test data. Since SOA deployment environment is distributed, heterogeneous it is very difficult to mirror. Even it cost high.

– **Lack of software version control management tools**

Multiple versions and releases of key SOA infrastructure may be accessible within and across sites. Such software components must be aligned consistently which further necessitates software version control management tools within and across the organizations. Developing and deploying versioning tool will be costly.

– **Lack of quality of service (QoS) information**

Sometimes the infrastructure providers do not have any QoS information which must be desired for safety critical systems. Lack of such information may harm human lives, or even catastrophic events may occur. Even, the system with less QoS requirement will put down consumer’s satisfaction level.

*Web Service Testing Challenges* A web service is a software system designed to support interoperable machine-to-machine interaction over a network [234]. There are two main styles for web services one is representational state transfer (REST) and other is simple object access protocol (SOAP). SOAP web services use SOAP and WSDL protocol stack to carry message payloads and describe service descriptions respectively whereas REST based web services use HTML and HTTP protocol layers to transport, describe, publish and consume messages. In this section we are going to analyze key testing challenges from the perspective of atomic web services, encapsulating functional task.

– **Absence of source and build code**

Generally, the web service is being called by web client more specifically a web browser. In such scenario, the source code or build code is hidden from the client which makes web service to be most eligible for black box testing where service code is hidden. With this fact, a tester can not apply white box testing techniques until (s) he makes the assumption of availability source code or can infer dependencies and behavioral operation by outside observations.

– **Undisclosed usage conditions**

A web service can be better utilized for reuse if its usage conditions are well known. But it is difficult to create a service for potential reuse for all situations or environments. To cope with it service developer may best estimate the common reuse context and prepare test cases in that context only. Ideally, exhaustive testing of all reuse contexts is difficult to achieve.
Unpredictable load demand and adverse impact on QoS requirements
An atomic web service can be consumed by service clients scattered across various sites. The load on entities like network, propagation delay, transfer rates, routing mechanism etc. cannot be estimated until the service invocation. Therefore, it is problematic to test and to validate QoS requirements.

**Web Service Composition Testing Challenges**

- **Unavailability of web service composition members**
  A web service may be an atomic, aggregate or composite. Composite web service relies on service composition members which may be controlled by multiple service providers residing within or across organizations. So during testing, such composite web service tester needs availability of each composition members involved in. Otherwise, the testing process fails at each end of the composite. The tester may put simplified behavior in the absentia of composite members which is not feasible or may schedule it only when all the composite members are available which again requires patience.

- **Lack of cooperation among multiple service provider**
  Composite member service seems to be a black-box to the tester. The service testers do not have sufficient information which may be required from each composite member services for rigorous testing. Sometimes even the service providers are not ready to disclose such information because of confidentiality or policy adopted by an organization or noncooperation due to untrustworthy environments. In such situation, testing will be a long running process involving convincing fruitfulness of disclosing contextual information.

- **Lack of common semantic web standards**
  All participating web service may enforce adoption of common semantic data model like ontology web language (OWL), resource discovery framework (RDF) schema etc. However common consensus on these semantic web standards is notoriously difficult because of multi-party involvement, unaware of inconsistencies induced due to the adoption of various semantic data models.

- **Lack of common fault and exception handling mechanism**
  Fault or exception may occur at any time any point during web service invocation. Exception handling mechanism includes compensation web services in place so that as it generates the compensation web service takes charge and ensures smooth invocation of web services within and across
organizations. Noncooperation among participant or composite service leads to the adoption of inconsistent fault or exception handling mechanism, blurring the testing process as dark.

- **Lack of common transaction management protocols**
  Transaction of an atomic, aggregated or composite service may cross the organizational boundaries. Different composite service may implement different transaction management protocols like ACID-compliant transaction or atomic transaction etc. These differences may arise only in unusual conditions where multiple concurrent transactions from multiple services lead to inconsistent state or service failure. Testing in such inconsistent or failure state will be an enormously challenging task.

- **Lack of common coordination protocols**
  Composite services invocation involves various composition member services. In a composite service, it is difficult to determine which member services play what role for example activation, registration, protocol specific or coordinator services. Due to multi-party involvement within and across organization boundaries, testing in such conflicting environment is too much difficult.

- **Side effect propagation**
  Atomic services may be part of a composite or an aggregate. Testing an atomic service may inadvertently affect other composite member services also. However, it is difficult to determine that a service is a side effect free by tester until it is tested. In the worst case, the tester is unable to determine the roots of side effects due to dynamic binding which occurs within composite and beyond that.

- **Lack of governance and source code control management**
  Neither one owns nor does one control the web services. Any change in composite member service can result in failure in entire composite service. If service tester is unaware of changes, (s)he may be unable to identify the root source of failure. The tester may perform a set of regression test to cease foreseen errors.

- **Lack of service comprehension tool**
  Currently, business process composites can be formed and well executed by web service business process execution language (WS-BPEL) and web service choreography description language (WS-CDL). These languages validate the business process and their endpoints. Up to our knowledge, no tool exist that supports service comprehension. Lack of such service comprehension tool increases the testing complexity.
- Lack of debugging capability
  Localization of errors within and beyond composite is difficult due to dynamic binding, non-reproducibility etc. Such unanticipated scenarios worsen the testing effort.

- Lack of common security protocols
  A composite service involves various security protocols for example SAML, kerberos, and x.509. A Common consensus on security protocols is difficult to achieve due to organization policy. Testing in such inconsistent security protocols limits the effort of the tester.

- Competency of tester for testing SOA capabilities

  - Inadequacy of the tester
    A tester who have worked for testing conventional systems using traditional testing techniques like black or white box testing for them SOA testing will be a new one. They do not have technical knowledge about how the SOA is designed, developed, integrated and deployed. Even the experienced tester might overlook tangible benefits of it. In fact, such testers might resist themselves from learning SOA concepts, and its associated technologies the web services. An SOA tester must have knowledge of all interfaces, organizational business requirements and strategies.

  - Unable to capture global snapshot
    The tester does not have zooming capability to visualize and record all the contextual information, SOAP request/response messages generated across various sites which further complicates inconsistency within and across organizations. In general, it is difficult to capture consistent global cut due to the limited capacity of the tester in visualizing and recording. Without this, it is difficult to test across the inconsistent cut.

  - Lack of efforts and funds in establishing simulation environment for testing
    The simulation environment is analogous to the actual deployment environment. This environment enables a tester to design and execute test cases prior to deployment of web services. However, establishing such environment is difficult because of participation of distributed, heterogeneous infrastructure components, their cost and security concern.

  - False information provided by service providers
    The service provider registers their service using WSDL. In doing so service provider may provide incorrect information which probably increase
their selection rank among other competitive service providers and thus influences consumer’s decision.

2.4.2 Non-functional Requirement Testing Challenges

In this section, we discuss testing aspects of major quality of service (QoS) parameters like availability, reliability, security, performance, and usability.

- **Availability Testing**
  A service is autonomous. So, when it is invoked it may not be available at the same time. To ensure high availability, replicas of services may be distributed across various sites, which may again lead to inconsistencies among them. Again each replica increases the cost of testing effort.

- **Reliability Testing**
  A message-level reliability can be achieved with various standardized message exchange patterns (MEPs) and reliable transport protocols like TCP/IP. Reliability testing ensures that service or business process gets reproducible results when invoked repeatedly over a certain period of time. In short, high availability gives rise to the reliability of services.

- **Security Testing**
  The major initiatives towards SOA adoption are its exposed nature over world wide web. This will open the services for vulnerabilities, attacks, and penetration. Security policies and trustworthy environment should be placed at each interaction level. Sometimes, security testing outperforms reusability testing when service consumptions violate trustworthy environments among the federated organizations. A Security testing must validate security even in an untrustworthy environment. The identified common vulnerabilities are denial-of-service attacks encapsulated in SOAP message retrieved by SOA intermediary nodes, bypassing authentication and authorization policies, executing malicious code, tracking username and passwords [54].

- **Performance Testing**
  The performance of the service-oriented system with respect to response time and throughput is very poor due to XML-based communication, layered architecture and distribution of services across world wide web (WWW) network with associated latency. Such system has high XML payload compared to conventional binary communication payloads. Even, the small services SOAP message need to be processed by each intermediate node which add extra overhead in computing the results. The performance testing of service-oriented systems
should be performed from the end-to-end business process, at unit level of web services, on validating each service interfaces and load testing of each service.

- **Usability Testing**
  SOA service does not have a user interface. Even, these services are invisible to the testing team. Testing such invisible services demands specifically designed testing tools. Without such automated tools, tester spent lots of time in tedious operations like composing WSDL, SOAP message and calling the services etc.

### 2.4.3 SOA Specific Testing Challenges

In this section, we discuss SOA specific testing challenges like interoperability, agility, adaptability, reusability, and governance.

- **Interoperability Testing**
  Developing business over the web through web services involves different process, technology, and tools, it is possible that service providers may use different tools to design service interface (WSDL), SLA and SOAP messages, and it is expected that all the deployed services must conform to standard interoperability profiles to become interoperable. The web service interoperability (WS-I) organization has developed basic profile (BP) and basic security profile (BSP) to enforce compatibility and interoperability. The WS-I has designed testing tools to test whether web services conforms to WS-I profiles.

  Interoperability exists between services and is not inherent in the services. Interoperability of web service in SOA environment requires standardization or contract at different layers between service consumers and service providers. The contract includes contracts at each layer of TCP/IP and ISO, contracts at XML based standards WSDL, SOAP, REST, and UDDI, contract at semantic standards OWL, RDF, OWL-S, contracts at transaction protocols ACID, WS-Atomic transaction, contract at organizational level SLA, WS-BPEL, WS-CDL, and WS-Coordination, contracts at exception handling mechanism. Thus interoperability testing becomes a matter of testing the service’s adherence to standards.

- **Agility Testing**
  A service-oriented system is expected to be modified easily. As its internal structure is not visible to the service consumer the system may change its implementation without knowing its consumers and without changing its interfaces. Testing complexity increase because each change in the system needs to be tested
properly. An organization may deploy a source code control system to align service versions properly.

- **Adaptability Testing**
  Services are expected to be agile enough to cope with changing business needs. Adaptability testing necessitates the understanding of overall service usage scenarios and how quickly the services can be provisioned to accept new customers. Identifying volatile part of requirement helps to test service for future change if arises. Since business environment changes rapidly, a complete system testing must be carried out in prior to deploying it thoroughly. Automated testing tools and regression testing will play a crucial role in this context.

  Syntactic interoperability can be achieved by standardizing service interface and its protocols. As, SOA capability spans various federated organizations, and non-interoperable security policies adopted by them it becomes difficult to test interoperability among them.

- **Reusability Testing**
  Service in a service-oriented system may be reused at each possible context of usage if it is standardized and identified properly. Reusability testing fosters standardization among heterogeneous systems. These data format conversion among service-oriented systems leads to significant overhead in processing. A tested service needs to be tested again at each possible reuse context.

- **SOA Governance Testing**
  The SOA governance testing is mainly challenged by inter-organizational communication of services and dynamic composition of services during run-time. SOA Governance testing ensures that all the standards, policies, and rules contracted by peer business organizations related to design, development, and deployment of services-oriented systems conform to it. Bertolino et al. [2] have proposed a governance monitor for the establishment of governance standards, policies and rules defined during foundation time, design time and runtime.

### 2.4.4 Testing Environment Challenges

It is difficult and costlier to establish an SOA testing environment mirroring/simulating the actual SOA deployment environment because of distributed environment, heterogeneity among SOA software and hardware. Even, this environment must enable the tester to design, develop, and executes test cases. Additionally, it enables the tester to capture all contextual information related to web services, business processes and QoS parameters of SOA.
Some organizations might place stubs, proxies and virtual machines to provide complete SOA deployment environment. These stubs and proxies serve the roles on behalf of missing services or business processes. And the virtual machines simulate the SOA software and hardware components. This testing environment should be available to service providers during development so they can start testing service and business processes in parallel with development. Doing so enables service providers to verify phase containment errors and can trace business requirements early in the development phase.

In short, the testing environment should incorporate testing tools to create, manage, executes and validates test cases, management of test inputs/outputs, version management among the service or business processes, and simulation of in absentia of services or processes.

2.5 Testing SOA Functional Requirements

The components which built service-oriented systems are web services, web servers, UDDI, ESB and web service clients (web browsers).

2.5.1 SOA Web Service Testing

Web services are the main capabilities in the service-oriented system. The first step towards web service testing is to determine whether its functional requirements are met. Additionally, a test case can be executed to ensure interoperability.

- **Unit Testing**
  
  Testing SOA web services in the absence of customized automated testing tools impose great challenges due to lack of control over web service and absence of other services which the service under test (SUT) call or may be called by. Web service testing is very similar to the unit testing of regular software. Stubs and proxies may be used to mock the behaviour of unavailable services. In such circumstances, the quality of testing depends heavily on the stub and proxy services. Complex services are difficult to be mocked because understanding the complex behaviour and developing the stub services is a complex task. Even the interface description of stub service is evolving since development is not yet completed, leads to uncertainty in testing results.

- **WSDL Testing**
  
  SOA service descriptions must be correctly described in a WSDL document. It must include XML elements like service name, the location of the service
(URL), service operations, input/output messages, service port, and service binding. These predefined XML elements must be validated prior to its publishing.

- **Web Service Functional Testing**
  The main goal of this testing is to ensure that service responds with correct response for given request message. Due to a variety of service consumers/clients (web browsers), it is impossible to anticipate exactly types of request message the clients will send. Exhaustive testing of all possible request is unbounded.

- **SLA Testing**
  Service provider and consumer stipulate an SLA, in which the service provider guarantees to consumer certain functionality with a given level of QoS. Thus, an SLA spells out who is responsible for what, what each party will do, and what they will not do. SLA testing involves various measurable qualities like accuracy, availability, capacity, cost, latency, reliable messaging and scalability. Optionally it includes unmeasurable attributes like interoperability, modifiability, and security.

- **Regression Testing**
  Regression testing of web service ensures that modifications due to bug fixes do not cause unexpected changes or failures. Each time a defect is fixed the potential exists to inadvertently introduce new errors, problems or defects. Regression testing of SOA services is the selective retesting of an old test suite on the each modification to the services or after each bug is fixed to ensure that no new bugs have been introduced as a result of bug fixing. The main purpose of regression testing is that web services up to the point of repair have not been adversely affected by the fix. Business organizations may devise guidelines to decide when to start regression testing. It includes up gradation/updating of services, up gradation of WSDL or SLA, change in the deployment environment, and retirement of services. The actual decision regarding when to perform regression testing will be based on many factors, which includes nature of change and usage environment of services. An organization must perform regression testing in a situation where the services involve human life, environmental change or business economy change.

- **Robustness Testing**
  Robustness Testing ensures that proper recovery mechanism is placed to handle the unexpected behaviour of services. Compensation services will handle exceptions when they arise. For that, both WSDL and SOAP messages may include fault section to cope with it.
• **Load Testing**
  Load or stress testing of service is used to test the performance of the web service when many users are accessing the services simultaneously. The response of the web service must be constant and its performance must not degrade with the increase in the number of users.

### 2.5.2 Web Service Integration Testing

Problems encountered during the integration testing of SOA web service are same to integration testing of object-oriented systems. Late binding, missing service participant at the moment of testing and dependencies to external services are the main problems associated with it. For example, which particular service is invoked in correspondence to service call can not be determined due to late binding. External service to which the service is dependent can change without notice.

• **Dynamic Web Service Composition Testing**
  Dynamic web service composition is the key characteristics of SOA, enabling runtime service composition by combining pre-existing services using standard languages. Web service composition is a group of related or confined services either locally or globally hosted on web servers. These compositions often span across the enterprise boundaries and deals with the involvement of stakeholders requirements and goals to meet the desired design process. More ever, these composite services is a business process that usually requires advanced programming skills and vast knowledge about SOA-enabled technologies like web services [260]. Recently, many web service languages have been proposed. These languages support long running transactions and multi-service scenarios [103]. SOA composite services can be formed by choreography and orchestration [180]. Therefore, testing service composition can be classified into two categories choreography and orchestration testing.

• **Choreography Testing**
  Service choreography involves interaction among multi-party with multiple resources, and each party is autonomous and controls its own behaviour. The choreography is the process that tracks the message sequences among multiple parties and sources [218]. A choreography of services between distributed parties, specifies which services to use, in which order, and under which conditions [77]. Additionally, web service choreography describes collaboration protocols of cooperating web service participants from a global view [193]. An example is web service choreography description language (WS-CDL). Testing service
choreography refers to testing of the entire path involved in executing the busi-
ess process. The path encounters a number of services, disparate applications, and multiple participants. Therefore choreography testing should consider gov-
ernance testing across federated organizations, testing of long-running business processes, testing of loosely coupled services including regression testing. The testing process for choreography is similar to the testing process of typical software systems. Scenarios for the business process should be devised along with basic, critical and optional flow paths. Test cases must traverse these paths and can additionally test the dynamism of composites services. The critical flow path must be identified carefully and specifically tested for both functional and non-functional characteristics.

- **Orchestration Testing**
  Orchestration refers to an executable business process that can interact with both internal and external web services [180]. It is the process where interactions between web services occur at the message level. The interaction procedure includes business logic and task execution order. The orchestration process can span applications and organizations, as a whole to achieve long-term transac-
tions and multi-step process model. The scenario of web service involves party’s perspective and orchestration always achieve control considering the party’s perspective. The orchestration is used to describe and execute a single viewpoint model [235]. Web service orchestration refers to web service description which takes a local point of view [205]. This means orchestration resembles web service collaboration in predefined patterns based on a local decision about web service interactions at message level or execution level. A representative for this is WS-BPEL [68].

  The WS-BPEL models business processes by specifying the workflows of car-
ying out business transactions. From a general perspective, the orchestration of services means to provide specification and realization of business processes that allows the message exchanges between one or more web services. Testing of BPEL is similar to testing services, and its interactions.

### 2.5.3 SOA Registry/UDDI Testing

The main objective of testing SOA registry is to ensure that only qualified services become part of SOA registry. Validation of service description involves proper mapping of WSDL to tModel data structure of UDDI, searching and invoking services based on certain business functions. Testing should be extended to know that registry data is valid, legitimate and up to date. Additionally, UDDI operations like identify-
ing, naming, describing, classifying, relating, grouping, and associating related shared information needs to be tested. Moreover, the XML data structure (UDDI) used to store service description should be tested for overflow and underflow conditions. Additionally, the tester may validate bindingTemplate data structure to ensure that only specification compliant services are included. SOA registry can be deployed publicly or privately. Testing private SOA registry will be difficult due to limited visibility and control. Testing public SOA registry involves business services publishing across the WWW, which requires a rigorous testing process in place. The public service registry may be tested for inconsistencies among various replicas of registry distributed across various sites. The tester should expect change control and version control management system should be in place to align versions and update UDDI across sites.

Testing dynamic service discovery capabilities of service registry ensures that only published or subscribed services can be found while invoking services. Also, it includes searching the web services using various inputs and rules. Subscription and notification capabilities of UDDI can be tested when a new version of service description is published all the concerned service consumers should be informed. Service monitors may be deployed at SOA registry level to log service usage data which enables the service providers to analyze data for identifying frequent service consumers, highly loaded and lightly loaded services, identifying security threats, analyzing load balancing scheme, audits etc. In this context, service virtualization may be employed to create virtual service endpoints effective in load balancing of highly and lightly loaded services. Finally, testing should be carried out in SOAP messages as UDDI runs over SOAP messages.

### 2.5.4 ESB Testing

ESB, a middleware software supports SOA services and applications integration, coordinate resources and manipulate information driven by business requirements. Ribarov et al. [129] have proposed a test-enabled ESB which intercepts and controls the communications between ESB and services. The intercepted message may be modified to support fault injection and test generation. Further, this modified message can be recorded for debugging, visualizing, analysis and black-box testing.

### 2.5.5 SOA System Testing

This testing phase will ensure that the SOA technical solution has delivered the defined business requirements and has met the defined business acceptance criteria. To ensure that this phase/level of testing is targeting only the key business scenarios of the
solution, the business stakeholders and testers must fully understand the quality and test coverage that have been achieved in the previous test phases.

2.6 Summary

In this chapter, we have discussed some definitions and concepts that will be used later in our algorithms. First, we have discussed various intermediate program representations. Next, we have discussed some basic concepts of slicing which are required to understand our dynamic slicing techniques. Then, we briefly dealt in precision and correctness issues of slices. We provided an overview of some important applications of program slicing. Then, we discussed the SOA layers and their architectural principles. Finally, we have discussed software testing and testing challenges associated with SOA.