CHAPTER - 1

Introduction

Program slicing is a method of separating out the relevant parts of a program with respect to a particular computation. Thus, a slice of a program is a set of statements of the program that affects the value of a variable at a particular point of interest. Program slicing was originally introduced by Mark Weiser [140] as a method for automatically decomposing programs by analyzing their data flow and control flow dependencies starting from a subset of a program’s behavior. Slicing reduces the program to a minimal form that still produces the same behavior. The input that the slicing algorithm takes is usually an intermediate representation of the program under consideration [263]. Normally, the intermediate representation of the program under consideration is a graph.

The first step in slicing a program involves specifying a point of interest, called the slicing criterion, which is expressed as \((s, v)\), where \(s\) is the statement number and \(v\) is the variable that is being used or defined at \(s\). Since the last couple of decades, the area of program slicing has been enriched by contributions from several researchers. Since its inception by Mark Weiser [140] as a debugging aid, many new techniques have evolved to enhance the accuracy, preciseness, and speed up of the process of slicing and to make program slicing usable in different applications. The technique of program slicing has evolved to handle unstructured and multi-procedure programs, structured as well as object-oriented, aspect-oriented, and feature-oriented programs. Also, these slicing techniques have found application in diverse problem areas such as software testing, debugging, software maintenance, understanding, code reuse, software refactoring, change impact analysis, coupling measurement etc. [71, 78, 79, 91, 127, 148, 149, 215, 216, 261].

Service-oriented architecture (SOA) is an architectural style for designing and developing distributed systems. Drivers for SOA adoption are easy and flexible for integration with legacy systems, streamlined business processes, reduced costs, and ability to handle rapidly changing business processes. From architectural and QoS parameters perspective, these drivers usually translate to interoperability and modifiability, which are achieved by
adhering to a set of architectural principles for service-oriented systems such as loose coupling, standardization, reusability, composability, and discoverability.

Testing of the service-oriented system is challenged by key factors like autonomy, heterogeneity, and dynamism. These autonomous systems have services/resources in their own control. The tester can request to test the system but may not force it to do so. Heterogeneity promotes software architect to design and construct components in their ways but during testing, the tester needs to test a variety of levels from networking protocols, encoding of information up to date formats. Dynamic service-oriented systems behave arbitrarily and may or may not participate in the business process. With these characteristics, testing of SOA-based applications is a difficult task because services may change their interfaces or behaviour at runtime.

1.1 Types of Program Slice

Many different notions of program slices and slicing criteria have been proposed since Weiser’s definition [140] of a slice. The main reason for this diversity is the fact that various applications of program slicing have been investigated, and that each application requires specific properties of the slices.

In this section, we discuss the various types of program slices: static and dynamic slices, backward and forward slices, and executable and closure slices, intra-procedural and inter-procedural slices, structured and unstructured slices, amorphous program slice, software architecture slice, real-time program slice, conditioned slice, and UML model slice.

1.1.1 Static and Dynamic Slices

One of the criteria for classifying slices is to distinguish between static and dynamic slices. Static slices are computed without making any assumptions regarding the values of the variables. A static slice with respect to a variable and a location includes all the statements that could possibly affect the computation of that variable at that location. In contrast, dynamic slicing makes use of the input of the program in order to reduce the size of the slice. In other words, we can say that dynamic slicing techniques compute precise slices. Fig. 1.1(b) shows a static slice of the sample program given in Fig. 1.1(a) with respect to slicing criterion <13, prod>.
Dynamic slicing was first suggested by Korel et al. [29] as a means for locating the cause of a program fault that is revealed by a particular execution. A **dynamic slice with respect to a slicing criterion** \(< s; V >\), for a particular execution, contains those statements that actually affect the slicing criterion in the particular execution. Therefore, dynamic slices are usually smaller than static slices and are more useful in interactive applications such as program debugging and testing. In such a situation, only the program’s behaviour for a specific input, rather than for the set of all inputs, is of interest. The sequence of statements that have actually been executed for the specific input is referred to as a trajectory or execution history. A trajectory differentiates each occurrence of the same statement. Typically a dynamic slice is computed with respect to an input, a variable, and a position in the trajectory. Dynamic slicing cannot be used to reason about all the possible computations with respect to a variable, and this characteristic limits its application to debugging. Static slices are computed using only static information whereas dynamic slices are generally (but not necessarily) computed using only dynamic information. However, Choi et al. [111], and Duesterwald et al. [65] reduce the amount of computations that have to be performed during the execution of the program, by using static information. Fig. 1.2(b) shows a dynamic slice of the sample program given in Fig. 1.2(a) with respect to slicing criterion \(< 13, \text{sum}, i = 5 \& n = 9 >\).
A number of hybrid approaches act as a bridge between the two extremes of static and dynamic slicing, by considering a set of executions. The notion of quasi-static slice in which the values of some inputs are fixed while the other inputs are not specified was proposed by Venkatesh [80]. For Field et al. [112] a slice where any subset of the inputs of the program may be supplied is called a constrained slice. Ning et al. [113] have proposed the notion of condition-based slicing in which the inputs in the slicing criterion were characterized by a logical expression (for example age $> 30$).

### 1.1.2 Backward and Forward Slices

Backward slices are composed of statements and control predicates that are collected by traversing backward. Thus, a static backward slice provides the answer to the question: “which statements affect the slicing criterion?”. Hence, these slices are called backward slices. Fig. 1.3(b) shows a backward slice of the sample program given in Fig. 1.3(a) with respect to slicing criterion $< 7, i >$. 

![Figure 1.2: Dynamic slice of a sample program](image-url)
Forward slices consist of the statements, which could (potentially) be affected by a given slicing criterion. A forward slice provides the answer to the question: “which statements will be affected by the slicing criterion?”. This variation of program slicing was first introduced by Bergeretti et al. [114] who called it information propagation within a program. The word “forward” was first used by Reps et al. [223]. Fig. 1.4(b) shows a forward slice of the sample program given in Fig. 1.4(a) with respect to slicing criterion < 2, sum >. in Fig. 1.4(b). Unless otherwise specified, we consider backward slices throughout this thesis.
1.1.3 Executable and Closure Slices

Slices can be either executable or closure. Executable slices are syntactically valid programs, which preserve the behaviour of the original program with respect to the slicing criterion. On the other hand, closure slices consist of the statements and control predicates of the program that directly or indirectly affect the values computed at the criterion, but that do not necessarily constitute executable programs. An example that illustrates the difference between executable and closure slices is shown in Fig. 1.5. Fig. 1.5(a) shows the original program; Figs. 1.5(b)-(d) show the slices with respect to \( g2 \) at line (17). Fig. 1.5(b) is a closure slice and Figs. 1.5(c)-(d) are executable slices [156], where procedure \( p \) is specialized into two variants: \( p_1 \) and \( p_2 \).

1.1.4 Intra-procedural and Inter-procedural Slices

Intra-procedural slicing computes slices within a single procedure. Calls to other procedures are either not handled at all or handled conservatively. The difficulty associated with inter-procedural slicing is to determine the inter-statement dependencies that exist between the statements from the calling procedure and those from the called procedure and those inter-statement dependencies that result from the call-return execution path. Many reported work [87, 90, 152] have successfully addressed the problems associated with slicing of programs involving inter-procedure calls. For distributed programs, intra-procedural slicing is meaningless as practical distributed programs contain more than one method. So, for distributed programs, inter-procedural slicing is more useful.
1.1.5 Unstructured Program Slice

For many applications, particularly where maintenance problems are the primary motivation for slicing, the slicing algorithm must be capable of constructing slices from “spaghetti” programs, written before the benefits of structured programming were fully appreciated. This is because of unstructured programming style using goto statement; all forms of jump statement, such as break and continue can be regarded as special cases of the goto statement. Such programs are said to exhibit arbitrary control flow and are considered to be “unstructured”. The traditional program dependence graph based approach by Ottenstein et al. [177] incorrectly fails to include any goto statements in a slice.

1.1.6 Amorphous Program Slice

Whether or not a slice is executable or non-executable, it is traditionally computed by deleting statements from the original program. The resulting slice can be regarded as a reduced version of the program. An effect minimal slice (later renamed amorphous slice) maintains the effect that the original program has upon the set of variables for which a slice is constructed. A variant of this conventional concept of a slice was introduced by Harman et al. [153]. Harman et al. [157] defined $q$ as an amorphous static slice of $p$ with respect to the slicing criterion $(K, i)$ if and only if $q$ is a $(\tau^{(K,i)}, \sqsubseteq_{AS})$ projection of $p$, where $\tau^{(K,i)}$ is syntax-preserving semantically equivalent slice and $\sqsubseteq_{AS}$ is semantic equivalence relation.

Consider the example program fragments in Fig. 1.6. All the fragments are $(\tau^{(x,5)})$ equivalent to one another, but none is a traditional static slice of any of the others, as none is a syntactic subset of any of the others. However, since Program 1 contains fewer commands than Program 2, which contains fewer commands than Program 3, it can be concluded that Program 1 is a $(\tau^{(x,5)}, \sqsubseteq_{AS})$ projection of Program 2 and that Program 2 is $(\tau^{(x,5)}, \sqsubseteq_{AS})$ projection of Program 3. Furthermore, Program 1 is a $(\tau^{(x,5)}, \sqsubseteq_{AS})$ minimal projection of Programs 2 and 3 (and of itself), because no other $(\tau^{(x,5)})$ equivalent program has fewer commands. Amorphous static slicing is simply traditional static slicing with a relaxed simplicity measure, allowing, for example, Program 1 of Fig. 1.6 to be regarded as a slice of Programs 2 and 3 of Fig. 1.6.

![Figure 1.6: Examples of amorphous static slices](image)
However, the slice does not have to be a subset of the original program, and transformations that preserve the semantics of the slice are allowed. Amorphous slicing does not slice away statements from the original program, one can argue that it cannot be accepted as a slicing method in the strict sense of the terminology. However, the important property of slicing, relied upon by all its applications, is the way it slices away semantic sub-components of the original program. Amorphous slicing is thus slicing in the semantic sense but not in the syntactic sense.

1.1.7 Software Architectural Slice

Software architecture is receiving increasing attention as a critical design level for software systems. As software architectural design resources (in the form of architectural specifications) are going to be accumulated, the development of techniques and tools to support architectural understanding, testing, reengineering, maintenance, and reuse will become an important issue. Jianjun Zhao [101] introduced a new form of slicing, named architectural slicing, to aid architectural understanding and reuse. In contrast to traditional slicing, architectural slicing is designed to operate on the architectural specification of a software system, rather than the source code of a program. Architectural slicing provides knowledge about the high-level structure of a software system, rather than the low-level implementation details of a program. In order to compute an architectural slice, he presents the architecture information flow graph, which can be used to represent information flows in software architecture. Based on the graph, he gives a two-phase algorithm to compute an architectural slice. Kim et al. [217] introduced the notion of dynamic software architecture slicing (DSAS). A dynamic software architecture slice represents the run-time behavior of those parts of the software architecture that are selected according to a particular slicing criterion such as a set of resources and events. They also described a methodology for using the notion, and an algorithm to generate dynamic software architecture slices. The feasibility and the expected benefits of the approach were demonstrated through a study of part of an electronic commerce system and a run-time execution of its architecture using a tool.

1.1.8 Real-Time Program Slice

Gerber et al. [196] described an approach to scheduling which relies upon slicing to identify the observable parts of a real-time system (those that interact with the environment through I/O operations) and the unobservable components (those which do not perform I/O). The unobservable computations need not fit within the time frame allocated to the task and are moved to the end of the task’s computation, increasing its schedulability. The system mixes
slicing and the single transformation code motion, but it does simply to re-order computation and does not delete computations. Therefore, the system does not produce amorphous slices.

1.1.9 Conditioned Slice

The conditioned slice was introduced by Canfora et al. [38]. A conditioned slice consists of a subset of program statements, which preserves the behaviour of the original program with respect to a set of program executions. The set of initial states of the program that characterizes this execution is specified in terms of first order predicate logic formula on the input variable. Conditioned slicing allows a better decomposition of the program giving the maintainer the possibility to analyze code fragments with respect to different perspectives.

1.1.10 UML Model Slice

Lallchandani et al. [118] have proposed an approach to slice UML model. He described a model slice as a selection of model elements from a model, or a part of a model selected based on a particular slicing criterion.

1.1.11 Aspect-Oriented Program Slice

Aspect-oriented programs work by providing an explicit mechanism for capturing the structure of crosscutting concern in a software system. Specific aspect-oriented features such as join points, advice, and the aspect that are different from existing procedural or object-oriented programming language. Zhao [117] describes how to slice such large aspect-oriented software. Various researchers have put their effort in slicing variant of aspect-oriented programs can be found in the literature [63, 126, 164].

1.2 Issues in Program Slicing

In this section, we discuss some of the major issues in the program slicing of service-oriented software (SOS).

- **Accuracy**

  An accurate slice as defined by Agrawal [87] is one which includes a statement of a program only when the value of the given variable at the given location is affected by
an occurrence of that statement. Although the concept of an accurate slice is simple in its description, determining whether or not a slice is accurate is not trivial. Furthermore, testing that an algorithm always computes accurate slices would require the elaboration of an infinite test set. No formal proof has been provided by the research community regarding the accuracy of slices.

- **Intermediate Representation**

In order to slice a service-oriented architecture (SOA)-based software, first, the program should be represented by a suitable intermediate representation. This intermediate representation should correctly represent the service-oriented architecture (SOA) features such as higher level abstraction, interoperability, composability, loose coupling, autonomy, reusability, statelessness, extensibility, testability and dynamic binding. A suitable intermediate representation in Chapter 4 for addressing higher level abstraction feature using service call dependency and composite dependency is developed. This intermediate representation is extended in Chapter 5 to represent interoperability and composibility features of service-oriented architecture (SOA)-based software by introducing intra-service dependency and inter-service dependency. In Chapters 6, suitable techniques for addressing testability features using semantic elements are developed.

- **Memory Requirement**

The memory requirement for both the intermediate representation and the static or dynamic slicing algorithms should be as small as possible. Otherwise, the stored data will run out of memory due to the large sizes of service-oriented architecture (SOA)-based software. Our intermediate representations and both static and dynamic slicing algorithms are equivalent to the existing ones with respect to memory space requirement.

- **Time Requirement**

The time requirement for any static or dynamic slicing algorithm should also be as small as possible as the algorithm will be generally used in interactive applications such as testing & debugging. Otherwise, the response time will be too large. Since we use program slicing as the basis for testing, we do consider time requirement as the performance criterion to measure the efficiency of our static and dynamic slicing algorithms. Even in the worst case, our algorithms perform equivalently to the existing ones.

- **Correctness**

The slicing algorithms should compute correct static or dynamic slices with respect to any given slicing criterion. A slice is said to be correct if it contains all the statements
that affect the slicing criterion. We claim that each of the proposed static or dynamic slicing algorithms computes correct static or dynamic slices with respect to any given slicing criterion.

- **Scalability**

The static or dynamic slicing algorithms should be developed in such a way that the algorithms can easily be extended to handle large scale software as the size of service-oriented software (SOS) are very large, involving multiple partner web services. Our static or dynamic slicing algorithms can easily be extended to handle such large and complex software.

### 1.3 Overview of Service-Oriented Architecture (SOA)

The emergence of service-oriented architecture (SOA) as an approach for integrating business applications that expose services, introduces many new challenges to organizations resulting in significant risks to their business. Particularly important among those risks are failures to effectively address QoS requirements such as performance, reliability, availability, security, testability and modifiability. Because the risk and impact of SOA are distributed across applications, it is critical to perform an architectural or model testing early in the software development life cycle.

Service-oriented architecture (SOA) is a very popular architecture paradigm for designing and developing distributed systems. The SOA solutions have been crafted to satisfy business goals that include easy and flexible integration with legacy systems, streamlined business processes, reduced costs, innovative service to customers, agile adaptation and reaction to opportunities and competitive threats.

One of the most important software engineering principles is phase containment error, which introduces checkpoints into the software development lifecycle phases. Software architecture testing is particularly important because architecture is the bridge between business goals and the software systems. Testing architecture or model that satisfies functional as well as non-functional (QoS) requirements (e.g., availability, security, and performance) is vital to the success of the system. Early testing of the requirements and the architecture save time and money.

In this section, we define service-oriented architecture (SOA), and service. Also we discuss the important service characteristics along with various advantages associated with SOA.
1.3.1 What is Service-Oriented Architecture (SOA)?

There are many definitions of SOA but none is universally accepted. We define a few important ones as below.

- Erl [219] defines service-oriented architecture as a model in which automation logic is decomposed into smaller, distinct units of logic. Collectively, these units comprise a larger piece of business automation logic. Individually, these units can be distributed.

- OASIS [176] defines service-oriented architecture as a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. It provides a uniform means to offer, discover, interact with and use capabilities to produce desired effects consistent with measurable preconditions and expectations.

- Opengroup [209] defines service-oriented architecture as an architectural style that supports service orientation. Service-orientation is a way of thinking in terms of services and service-based development and the outcomes of services.

- Microsoft [154] defines service-oriented architecture as an architectural pattern in computer software design in which application components provide services to other components via a communications protocol, typically over a network. The principles of service-orientation are independent of any vendor, product or technology.

Service-oriented architecture (SOA) is a novel way to design, develop, deploy systems, in which services are reusable business functions exposed via well-defined interfaces, a separation of concern between the interface and its implementation, and service clients use service functionalities from available service descriptions. The SOA infrastructure enables dynamic discovery, dynamic composition, and dynamic invocation of services. From a technical point of view, SOA is an architectural style or design paradigm. Systems that are built based on the SOA principles are called service-oriented systems.

1.3.2 What is Service?

The central pillar of service-oriented architecture (SOA) is the services. For an SOA, service

- is a reusable component.
- is self-contained, highly modular, autonomous business task.
• is a distributed component.
• has a published service interface.
• stresses interoperability.
• is discoverable.
• is dynamically bound.

1.3.3 Important Service Characteristics

• Service Encapsulations

The services encapsulate logic within a distinct context. This context can be specific to a business task, a business entity, or some other logical grouping. The concern addressed by a service can be small or large. Therefore, the size and scope of the logic represented by the service can vary. Further, service logic can encompass logic provided by other services. In this case, one or more services are composed into a collective.

As shown in Fig. 1.7, when building an automation solution consisting of services, each service can encapsulate a task performed by an individual step or a sub-process comprised of a set of steps. A service can even encapsulate the entire process logic. In the latter two cases, the larger scope represented by the services may encompass the logic encapsulated by other services.

![FIGURE 1.7: Service encapsulation](source)

• **Service Relations**

Within SOA, services can be used by other services or other programs. Regardless, the relationship between services is based on an understanding that for services to interact, they must be aware of each other. This awareness is achieved through the use of service descriptions. A service description in its most basic format establishes the name of the service and the data expected and returned by the service. The manner in which services use service descriptions results in a relationship classified as *loosely coupled*. For example, Fig. 1.8 illustrates that service A is aware of service B because service A has acquired B’s service description.

![Service relations diagram](image)

**FIGURE 1.8:** Service relations

• **Service Communications**

![Service communications diagram](image)

**FIGURE 1.9:** Service communications

After a service sends a message on its way, it loses control of what happens to the message thereafter. That is why we require messages to be fitted with enough intelligence to
self-govern their parts of the processing logic. Hence, messages, like services, should be autonomous and self-governing. Fig 1.9 shows message existing as an independent unit of communication.

1.3.4 What is a Business Process?

Business processes are a defined set of tasks that produce specific capabilities or products. Each task is composed of one or more services that are integrated to form the process [64].

1.4 Advantages of Service-Oriented Architecture (SOA)

This list of common advantages of SOA is generalized and certainly not complete. It is merely an indication of the potential that SOA can offer [219].

- **Improved integration (and intrinsic interoperability)**
  SOA can result in the creation of solutions that consist of inherently interoperable services. Because of the vendor-neutral communications framework established by web services-driven SOAs, the potential is there for enterprises to implement highly standardized service descriptions and message structures. The net result is intrinsic interoperability, which turns a cross-application integration project into less of a custom development effort, and more of a modeling exercise.

- **Inherent reuse**
  Service-orientation promotes the design of services that are inherently reusable. Designing services to support reuse from the get-go opens the door to increased opportunities for leveraging existing automation logic. Building service-oriented solutions in such a manner that services fulfill immediate application-level requirements while still supporting the degree of reuse by future potential requesters establishes an environment wherein investments into existing systems can potentially be leveraged and re-leveraged as new solutions are built.

- **Streamlined architectures and solutions**
  The concept of composition is another fundamental part of SOA. It is not, however, limited to the assembly of service collections into aggregate services. The WS-* platform is based in its entirety on the principle of composability. The reduced performance requirements mentioned previously only refer to the fact that SOA extensions are composable and therefore allow each application-level architecture to contain extensions only relevant to its solution requirements.
• **Leveraging the legacy investment**
  The industry-wide acceptance of the web services technology set has spawned a large adapter market, enabling many legacies environments to participate in service-oriented integration architectures. This allows IT departments to work toward a state of federation, where previously isolated environments now can interoperate without requiring the development of expensive and sometimes fragile point-to-point integration channels.

• **Establishing standardized XML data representation**
  On its most fundamental level, SOA is built upon and driven by XML. As a result, an adoption of SOA leads to the opportunity to fully leverage the XML data representation platform. A standardized data representation format (once fully established) can reduce the underlying complexity of all affected application environments. For examples the XML documents and accompanying XML schema definition (XSD) passed between applications or application components fully standardize format and typing of all data communicated. The result is a predictable and therefore easily extensible and adaptable communications network. XML’s self-descriptive nature enhances the ability for data to be readily interpreted by architects, analysts, and developers. The result is the potential for data within messages to be more easily maintained, traced, and understood.

• **Focused investment on communications infrastructure**
  Because web services establish a common communications framework, SOA can centralize inter-application and intra-application communication as part of standard IT infrastructure. This allows organizations to evolve enterprise-wide infrastructure by investing in a single technology set responsible for communication.

• **“Best-of-breed” alternatives**
  Because SOA establishes a platform-neutral communications framework, it frees IT departments from being chained to a single proprietary development and/or middleware platform. For any given piece of automation that can expose an adequate service interface, you now have a choice as to how you want to build the service that implements it.

• **Organizational agility**
  Agility is a quality inherent in just about any aspect of the enterprise. Much of service-orientation is based on the assumption that what you build today will evolve over time. One of the primary benefits of a well-designed SOA is to protect organizations from the impact of this evolution. When accommodating change becomes the norm in distributed solution design, qualities such as reuse and interoperability become commonplace. The predictability of these qualities within the enterprise leads to a reliable
level of organizational agility. However, all of this is attainable through proper design and standardization. Regardless of what parts of service-oriented environments are leveraged, the increased agility with which IT can respond to a business process or technology-related changes is significant.

1.5 Research Problem Definition

The research problem that we are going to address in our work can be formulated by raising various questions as mentioned below:

- Research Hypothesis:
  “Can program slicing be applied to test service-oriented software using service oriented architecture modeling language (SoaML) or business process modeling notation (BPMN) models”?

- Research Assumption(s):
  Nil

- What is the problem?
  The research problem can be defined by raising various research questions (RQs) and defining the expected outcome as given below:

  – Research Questions:
    * RQ1: What are the problems related to the testing service-oriented architecture (SOA)-based software?
    * RQ2: What are the benefits related to the testing service-oriented architecture (SOA)-based software?
    * RQ3: What will be the process of testing it?
    * RQ4: How effective will the proposed testing approach be for an organization in terms of addressing the problems?

  – Expected outcomes:
    In general, the expected outcome of our research work will be a new technique or approach, and knowledge acquisition which address the issues of testing service-oriented architecture (SOA)-based software. In particular, the final research work will include the following outcomes:
* Problems associated with testing service-oriented architecture (SOA)-based software.
* Advantages related to the testing service-oriented architecture (SOA)-based software.
* Development of some techniques for testing service-oriented architecture (SOA)-based software.
* Implementing the proposed testing approach/technique in an organization.
* The evaluation results of implementing the proposed test approach/technique in an organization.

• Where is the problem?
  The problem lies in testing service-oriented architecture (SOA)-based software.

• How to solve the problem?
  The problem can be addressed by devising suitable intermediate representations using SoaML or BPMN models and testing through program slicing technique.

• Why you want to solve the problem?
  It helps the tester or IT vendors to test service-oriented architecture (SOA)-based software early in the design phase, so the cost of detecting and correcting errors will be less compared to error detection at coding phase.

• Is the problem current?
  Yes, and can affect other parent software like cloud-based software too.

• Will the problem continue in the future if it is not solved? No.

• Who is suffering from that problem?
  Software testers, IT vendors and other stakeholders.

• Will this improve or disprove the existing knowledge?
  Yes, it will improve the existing knowledge.

1.6 Motivation for Our Research Work

SOA has started gaining importance in this competitive world of business and cloud-based applications. The research community along with IT vendors have started substantially contributing to the fast adoption of SOA in the business world.
Between 2005 and 2007, multiple surveys were conducted by organizations such as Forrester, Gartner, and IDC, which showed that the top drivers for SOA adoption were mainly internally focused: these top drivers generally included application integration, data integration, and internal process improvement. This is changing. Forrester [73] showed that the number of organizations currently using SOA for external integration is approximately one-third of the surveyed organization [73]. While the percentage of externally focused SOA applications is still a minority, this percentage has been growing and the trend will continue as organizations look at SOA adoption for supply-chain integration, access to real-time data, and cost reduction through the use of third-party services via the cloud or software-as-a-service (SaaS).

All major IT vendors, a few names, such as IBM, Tibco, Software AG, Sun Oracle, SAP and so on have made huge investments into SOA in recent years, making up a global estimated budget of $2 billion in 2007, which is further expected to rise and reach $9.1 billion by 2014 [95]. Even, corporate giants like Microsoft, IBM, Sun Oracle, SAP, Infosys have already proposed their own SOA solutions and related software products. They have also rolled out many success stories by implementing SOA in varied organizations. Even there are major SOA failures reported across the globe.

The current year 2016 is a big boost for SOA as many e-commerce companies adopted SOA for their business processes. This has helped immensely to improve agility and business integration. This also gives an indication that we need more reliable SOA-based systems and their testing is inevitable.

But still, the challenges are live like standardization of architectures, increased business agility, flexibility, reuse, data rationalization, integration, testing, error handling, security, costs reduction, and so on. Different SOA frameworks being used in the enterprise sector face mainly integration challenges to put together data, business process, security, and stakeholders. All these raise a vital issue of quality of SOA-based software. There is a need for SOA-based software to be testable across different enterprise applications giving emphasize on quality.

Web services can be offered within organizational borders in private SOA deployments, as well as across organizational borders by third party providers. Recent years have seen an increasing number of web services made available over the internet by various service providers. As a consequence, the issues of trust and dependability on third-party providers have been receiving increasing importance. Service requesters need to ensure that provided web services satisfy their requirements in different aspects and that they have also been correctly implemented, before integrating them into their systems.

Web services are a concrete implementation of the SOA concept. They incorporate appli-
lications that provide a set of operations accessed by other applications through the Internet, using XML standards such as SOAP, WSDL, and UDDI. This technology enables to interconnect different enterprises and to build complex business applications which are used in safety critical environments. SOA and web services add new factors that need to be considered during software development. Hence, it goes without saying that web services that support SOA implementation opens up a lot of scope for testing.

One problem with the current standard for web service description language (WSDL) is that it lacks support for service interface descriptions beyond the external interface of operation signatures. WSDL descriptions lack the means to specify the testing support of a web service so that requesters are aware of the exact behaviour expected from the consumed service. Therefore, different standards or languages are required to describe the additional functional and non-functional web service aspects, including its behaviour to support testability.

To the best of our knowledge, a few SOA testing methodologies have been reported. For example according to Canfora et al. [75], SOA testing must span several levels, from individual services to inter-enterprise federations of systems and must cover functional and non-functional aspects. SOA has a unique combination of features, such as the run-time discovery of services, ultra-late binding, QoS-aware composition, and SLA automated negotiation [76]. Run-time discovery and ultra-late binding entail that the actual configuration of a system is known only during the execution and this may make many existing integrations testing techniques inadequate [76].

Last but not the least, distribution, lack of control and observability, dynamic integration with other applications, and XML standards usage are key features of this new type of software. In this context, software testing is essential to guarantee a high degree of service quality and reliability.

So, there is a pressing necessity to devise a testing approach for SOA-based software. We also notice that testing approaches have been changed from traditional applications to object-oriented applications and must be changed for SOA-based software too. We have given kind attention to this aspect.

With this motivation for developing techniques for testing SOA-based software, in the next section, we identify the major goals of this thesis.
1.7 Goals and Objectives of Our Research Work

The main goal of our research work is to test SOA-based software through program slicing technique during design, testing and maintenance phases of software development lifecycle. To address this broad objective, we identify the following goals:

1. We first wish to compute static slices of SOA-based software using SoaML models, for this we want to develop:
   - Suitable intermediate representations for SOA-based software on which the slicing algorithms can be applied.
   - Static slicing algorithms for SOA-based software using the proposed intermediate representation.

2. Then, we wish to extend this approach for computing dynamic slices of SOA-based software, as dynamic slices are useful for interactive applications such as testing & debugging.

3. Next, we aim to test web services from its WSDL using program slicing artifacts i.e dependence graph.

4. At last, we aim to propose an approach to test SOA-based software using BPMN and SoaML models.

5. In addition to investigating our slicing algorithms and testing algorithms, theoretically, we wish to implement all the proposed algorithms experimentally for verifying their performance, correctness, and preciseness.

6. Finally, we plan to carry out an analysis of our experimental studies to draw broad conclusions about the realized work for time and space requirements.

Fig. 1.10 presents a conceptual schema of our work targeted towards meeting the aforementioned objectives. It shows that a key component of our work focuses on representation of SoaML models that is suitable for model slicing. The conceptual schema depicted in Fig. 1.10 comprises four blocks viz., input, algorithm, implementation, and output respectively. The input block depicts the different inputs to be given, namely, a SoaML model and a slicing criterion, that can be given to the algorithm block. Moreover, a SoaML system design is supposed to be given input using the interfaces, contracts, participants, choreographies, and architectures models. Next, the box titled “proposed technique” represents the
algorithm block depicting that this thesis deals with proposing model slicing algorithms. Further, the implementation block indicates that the experimental studies presented in this thesis are carried out through the development of different prototype tools. This is represented using a box titled “prototype tool” for the corresponding technique proposed in the algorithm block. Subsequently, the output block shows the expected outcome from the thesis work viz., intermediate representation, static and dynamic model slices. In the context of the above-identified objectives for our work, the block marked algorithm in the conceptual schema corresponds to the Objectives 1, 2 while the implementation block corresponds to the Objectives 6, 7.

Next, Fig. 1.11 presents a conceptual schema of our work targeted towards meeting the aforementioned objectives. It shows that a key component of our work focuses on web service description language (WSDL) that is suitable for black-box testing. The conceptual schema depicted in Fig. 1.11 comprises four blocks viz., input, technique/approach, implementation, and output respectively. The input block depicts the different inputs to be given, namely, a WSDL, XSD, and a test case, that can be given to the technique/approach block. Moreover, an SOA-based software is supposed to be given input using the SOA artifacts such as XML, XSD, WSDL, SOAP, and UDDI. Next, the box titled “proposed technique” represents the algorithm block depicting that this thesis deals with proposing an extension to WSDL using XSD. Further, the implementation block indicates that the experimental studies presented in this thesis are carried out through the testing tools. This is represented using a box titled “open source testing tool” for the corresponding technique proposed in the technique/approach block. Subsequently, the output block shows the actual results from executing test cases. In the context of the above-identified objectives for our work, the block
FIGURE 1.11: Conceptual schema of the planned work referencing objectives 3, 5 and 6

marked technique/approach in the conceptual schema corresponds to the Objectives 3 while the implementation block corresponds to the Objectives 5, 6.

FIGURE 1.12: Conceptual schema of the planned work referencing objectives 4, 5 and 6

Further, Fig. 1.12 presents a conceptual schema of our work targeted towards meeting the aforementioned objectives. It shows that a key component of our work focuses on SoaML and BPMN models. The conceptual schema depicted in Fig. 1.12 comprises three blocks viz., input, algorithm & implementation, and output respectively. The input block depicts the different inputs to be given, namely, a SoaML service interface diagram and BPMN diagram that can be given to the algorithm&implementation block. Next, the box titled “proposed
technique” represents the algorithm block depicting that this thesis deals with proposing CFG generation, test path generation, XSD schema generation, XSD schema instance generation, and test case execution respectively for each model. Subsequently, the output block shows the test result from this technique. In the context of the above-identified objectives for our work, the block marked algorithm&implementation in the conceptual schema corresponds to the Objectives 4, 5, and 6.

1.8 Scope of Our Research Work

The scope of our work will be limited up to service-oriented software testing. It includes which aspects of SOA-based software should be tested (specification/model/code), the elements to be tested (web service, web service description language (WSDL), XML schema (XSD), universal description discovery and integration (UDDI), simple object access protocol (SOAP)) and other XML artifacts produced during the testing effort. Also, it includes testing strategies for the unit, integration, and systems-level testing of SOA based web services (for both functional and non-functional attributes).

This means that instead of covering small pieces of testing SOA software development process, we preferred to advance as far as possible on slicing and testing SOA-based software with the aim of being able to present convincing solutions to the SOA challenges discussed in Section 2.4. Without a doubt, slicing and testing service-oriented software are complex problems as well and deserve to be investigated intensively, which has been done in the scope of this thesis.

1.9 Original Contributions of Our Research Work

In this research work, we have made progress in the field of slicing and testing service-oriented software. We have developed novel techniques for solving the previously listed challenges and implemented software prototypes to prove the applicability of our concepts. The most significant contributions and achievements are as follows:

- A simple technique for static slicing of SOA-based software, based on an intermediate representations.
- A technique for dynamic slicing of SOA-based software, using an intermediate representations.
• An approach for testing SOA-based software, based on WSDL which provides business flexibility.

• Some approaches for testing SOA-based software using BPMN and SoaML models.

Moreover, we have published the software prototype as open-source, as a contribution to the research community.

1.10 Evaluation of Our Contributions

The presentation of novel concepts always requires an evaluation in order to prove their applicability, usefulness, and correctness. Depending on the type of concepts, different types of evaluations make sense to be applied. For this work, however, the evaluation was not trivial.

We have performed a comparative evaluation, by matching our approach to other available ones, in order to prove applicability of our proposed concepts. This is mainly due to the novelty of our work and the lack of direct competitors. Also, we have not done a precise performance evaluation, as our contribution is not about performance issues nor does it prove the quality of our approach in any case. It would merely assess the applicability of our prototype implementation, which is primary importance being a proof of concept.

Without a doubt, a real-world evaluation, where our concepts and prototypes are applied in real SOA development projects would make the most sense and give valuable insights into how much the testing process got improved by our contribution. Unfortunately, this was not possible as (i) we did not have access to test a significant number of real-world SOA projects and (ii) it would have been not easy to convince the testers to apply our prototype implementation in their work.

Instead, we evaluated our concepts in a selective manner, choosing what we regarded as reasonable and realizable case studies. For instance, we included a test case generation technique which deals with generating test cases at design time from SoaML service interface diagrams. In contrast to that, we tested SOA-based software at the run-time, which has a significant effect on test results. Moreover, we applied our approach to several standardized case study projects for an internal assessment and as a proof of usability of the prototype implementation.
1.11 Research Methodology

The term *methodology* refers to the overall approaches and perspectives of the research process as a whole. A preliminary total of four research questions is proposed for our research work. The answer to each research question is associated with proper selection of research methods.

- **Qualitative Methods**
  The qualitative part of the research deals with the participatory knowledge claims, open-ended interviewing and narrative design. First, the knowledge participates for finding the related problems through literature and interviews along with designing the narrative solution for the problem.

  The RQ1, RQ2, and RQ3 will be answered through interviews and literature review, which will identify the problems and benefits related to testing SOA-based software. In addition, the process will be defined after analyzing the results of RQ1, RQ2, and RQ3.

- **Quantitative Methods**
  The quantitative part of the research deals with the experimental strategy of testing SOA-based software, results, and their comparisons. First, the study defines the testing process and then collects/creates the data to observe the testing process. To evaluate the defined testing approach in the qualitative part, experimentation is needed for the evaluation of its effectiveness in order to answer RQ3 and RQ4. The design of the experiment has been finalized after having a discussion with the supervisor. Experiments have been performed and demonstrated to the supervisor. Even the results are produced for comparison with related work.

- **Research Question Methodology**
  - RQ1: Literature Review
  - RQ2: Literature Review
  - RQ3: Interview/Experiment
  - RQ4: Experiment

  Finally, we announce our data formulation process required for our work by raising various questions as mentioned below:

  - Why did you collect certain data?
• What data you collected?
• From where you collected it?
• How you collected it?
• How you analyzed it?

The main research methodologies to generate data for our testing purpose can be linked to two approaches named positivistic research and phenomenological research. The positivistic research approach includes surveys, experimental studies, longitudinal studies, and cross-sectional studies. The phenomenological research approach includes case studies, action research, ethnography, participative inquiry, feminist perspectives, and grounded theory.

For our research work, we have followed the following specific methodologies which have been selected from both positivistic and phenomenological research approaches:

• Descriptive survey is concerned with identifying and counting the frequency of a particular response among the survey group.

• Analytical survey is used to analyze the relationship between different elements (variables) in a sample group.

• Experimental studies are carried out in carefully controlled and structured environments and enables the causal relationships of phenomena to be identified and analyzed. The variables can be manipulated or controlled to observe the effects on the subjects studied.

• Longitudinal studies are carried out over an extended period to observe the effect that time has on the situation under observation and to collect primary data (data collected at first hand) of these changes.

• Case studies offer the opportunity to study a particular subject and usually involves gathering and analyzing information. Information may be both qualitative and quantitative. Case studies can be used to formulate theories or, be:

  – Descriptive (e.g. where current practice is described in detail)

  – Illustrative (e.g. where the case studies illustrate new practices adopted by an organization)

  – Experimental (e.g. where difficulties in adopting new practices or procedures are examined)
– Explanatory (e.g. where theories are used as a basis for understanding and explaining practices or procedures)

1.12 Outline of the Thesis

The rest of the thesis is organized into chapters as follows.

Chapter 2 discusses the basic concepts related to program slicing, service-oriented architecture (SOA), and software testing which are used in the rest of the thesis. We discuss some graph-theoretic concepts and definitions which will be used later in our algorithms. Then, we discuss some intermediate program representation concepts which are used in slicing techniques. Then, we discuss the concepts of precision and correctness of a dynamic slice. Further, we discuss service-oriented architecture (SOA) layers and its architectural principles. Finally, we discuss software testing.

Chapter 3 provides a brief review of the related work relevant to our contribution. In this chapter, we first discuss the work on static slicing of service-oriented software. Then, we discuss the work on dynamic slicing of service-oriented software. Further, we discuss the related work on white-box testing of web services. Finally, we discuss the work reported on WSDL based testing and model-based test case generation technique.

Chapter 4 presents our static slicing algorithm for service-oriented software using service oriented architecture modeling language (SoaML) service interface diagram. We first discuss some definitions related to static slicing. This is followed by the proposal of an intermediate representation for service-oriented software named SIDG (service interface dependency graph) and then a static slicing algorithm SSSIM (static slicing of service interface model) along with its pseudocode. A complexity analysis for SSSIM is next presented. Finally, we present an implementation of our prototype tool and discuss the experimental results obtained using the prototype tool.

Chapter 5 presents our dynamic slicing algorithm for service-oriented software using service oriented architecture modeling language (SoaML) choreography diagram. We introduce few basic concepts and definitions pertinent to the dynamic slicing of service-oriented software. We first develop an intermediate representation of service-oriented software called SOSDG (service-oriented software dependence graph) and then present the dynamic slicing algorithm MBGDS (marking based global dynamic slicing) along with its pseudocode. The complexity analysis of MBGDS is next discussed. Subsequently, based on our MBGDS algorithm, we describe a prototype tool implementation named SOSDS (service-oriented software dynamic slicer). We also present the experimental results obtained in slicing sev-
eral SoaML models using SOSDS. We claim that our algorithm MBGDS computes correct dynamic slices.

**Chapter 6** reports our approaches for carrying out black-box testing of service-oriented software. In the first technique of black-box testing, we first present fundamentals of web service description language (WSDL) document, which describes the service interface and is of prime focus for black-box testing. Next, we present our proposal of extending the WSDL document. Then, we describe the proof of our extension by carrying out black-box testing using various WSDL data sets. Finally, we claim that our extension increases testability of web services without violating the architectural principles of services.

Secondly, we present novel approaches to test service-oriented software using BPMN and SoaML models. First, we describe our testing algorithm for service-oriented software using BPMN (business process modeling notation) diagram. In this approach, we present a CFG generation algorithm and its implementation. Subsequently, we discuss the test path and test case execution for the generated CFG. In the second approach, we present another algorithm for testing service-oriented software using SoaML (service oriented architecture modeling language) service interface diagram. In that context, we discuss the XML schema generation and its instance generation process. Finally, we discuss the test case execution of XML schema instance.

**Chapter 7** concludes the thesis with a summary of our contributions. Finally, we discuss the possible future extensions to our work.

The work flow diagram of the complete thesis is shown in Fig. 1.13.

**FIGURE 1.13: Thesis outline**