CHAPTER - 5
Dynamic Slicing of Service-Oriented Software

One major drawback of the static slices computed in the last chapter was that they were found to be highly abstracted. This was due to two main reasons. First, the computed static slices were small in terms of number of service interface nodes. And, the slicing criterion considered service interface nodes based on static analysis rather than any particular business scenario. Such a slice computation exercise decreases the utility of those slices, especially when the requirement arises to identify specific services based on a certain business scenario execution. Though such static slices can be used in applications such as understanding, but when we need to develop a slicing-based technique for test case generation, the utility of small static slices becomes limited. Therefore, there arises a need to compute slices considering a specific business scenario execution along with a particular input. These slices are generally termed as dynamic slices. A major goal of any dynamic slicing technique is efficiency since the slicing results may be used during interactive applications such as program debugging and testing. Efficiency is especially an important concern for slicing SOA-based software since their sizes are typically very large. Large software results in very large intermediate graphs and can result in response times of several hundreds of seconds.

Many of the real life SOA-based software services are distributed which run on different machines connected to a network. It is usually accepted that testing and debugging of services are much harder compared to those of sequential programs. The non-deterministic nature of distributed services, statelessness, non-availability of source code, unsynchronized interactions among services, and dynamic nature are some reasons for this difficulty. An increasing amount of resources are being spent in understanding, debugging, testing and maintaining these software. Slicing techniques promise to come in handy at this point. However, research results in slicing SOA-based software have not been reported in the literature. It is the objective of this chapter to present our work concerning development of a dynamic slicing algorithm for SOA-based software.
SoaML (Service oriented architecture modeling language) diagrams are the basic modeling artifacts for service-oriented architecture (SOA). These SoaML models can be used for testing service-oriented software (SOS). Testing can be planned at the design phase of software development life cycle. With this motivation, in this chapter we propose a new global dynamic slicing algorithm for computing slices of SOA-based software. Only the interoperability, compositability, and non-availability of source code issues are addressed here. With this context, we present a novel technique to compute global dynamic slices for service-oriented software (SOS) based on SoaML Sequence Diagram. In our technique, we first map each message in sequence diagram with the corresponding web service messages. This mapping is observable. After that, we construct an intermediate representation of SoaML sequence diagram which we called service-oriented software dependence graph (SOSDG) which is an intermediate representation that needs to be stored and traversed to get a global dynamic slice as and when web services get executed. This SOSDG identifies data, control, intra-service and inter-service dependencies from SoaML sequence diagram and from web service execution. For a given slicing criterion our algorithm computes global dynamic slice from SOSDG and identifies the affected service. The novelty of our work lies in computing of global dynamic slices based on SOSDG, its dependencies induced within or across organizations and small slices.

The rest of this chapter is organized as follows. Section 5.1 introduces a service-oriented software choreography example of buying a product from the online retailer and its mapping with web services. Section 5.2 presents basic definitions relevant to our proposed algorithm and extensions to dynamic slicing technique. Our intermediate representation SOSDG is discussed in Section 5.3. Section 5.4 presents our MBGDS (marking based global dynamic slicing) algorithm for computing global dynamic slice of service-oriented software, its working, correctness proof and theoretical complexity analysis. Section 5.5 presents the implementation and experimental result covering our tool SOSDS’s design, implementation, data sets and results obtained. Section 5.6 compares our work with some existing work. And finally, Section 5.7 concludes the chapter.

5.1 Service-Oriented Software Example: Online Shopping System

Let’s take a real-world example to demonstrate service-oriented software. Buying a product from the online retailer is a good example of service-oriented software choreography. Generally, a product seller registers its product with an online retailer. Other service providers may also register for shipping and logistic service with retailer too. This registration ser-
vice interface is being provided by the online retailer. When a customer wants to buy a product, customer searches the product at online retailer. Thus, searching a product by customer represents a web service. Once the product is found, the customer goes for buying it. The buying service is a composite service which first checks for user authentication through login. Again this login service is being provided by the retailer for their registered users or users may signup or else use the third-party login service like www.facebook.com or www.gmail.com. Once the customer successfully logged on, the customer can go to make payment or add the product to shopping cart for later payment or can use the cash on delivery (COD) options. The payment service interface is being provided by various banks. The user selects the type of card, the bank and other information and proceeds for payment. Once the transaction is successful, the order is being confirmed and the product is being sent to the customer address. This service-oriented software choreography is best described using the SoaML sequence diagram shown in Fig. 5.1

The `product_registration()` service accepts product name, quantity, price, seller name and contact address as input or arguments and returns successful or unsuccessful messages depending on the service computations. Similarly, the `couriercomapny_registration()` service accepts shipping company name, address and charges as input or arguments and returns successful or unsuccessful messages depending on the service computations. These service providers (product seller, courier company etc.) may register the products or services successfully at the begining so that their product can be found at an online retailer. Next, the `login()` service accepts the username and password and authenticate the user’s credential. The `signup()` service signs users with mobile no, username and password etc. The `search()` service accepts product name, pin code, and quantity and lists the results if the product is found. The `search()` service is a composed service which checks product availability at particular address using a pin code. The `add_to_cart()` service accepts product name and quantity and puts the product into user cart. The `add_to_cart()` needs prior execution of the services `login()` or `signup()` or `thirdparty_login()`. The `make_payment()` service accepts card type, card no, expiry date, PIN, OTP etc. for online payment through registered payment gateways. These payment gateways are being provided by banks for their registered customers when requested. Next, the `make_courier()` service accepts the customer name, address, contact no, product name, and quantity, and send the product with assigned tracking ids.

In this service choreography, we can see that the `product_registration()`, `couriercompany_registration()`, `login()`, `signup()`, `search()`, and `add_to_cart()` are the services being provided by online retailer, while `make_payment()`, `make_courier()` and `thirdparty_login()` services are being provided by other service providers or parties.
FIGURE 5.1: The SoaML sequence diagram for buying a product

1. Buy Product Service

Customer
- Payment Company
- Online Resolver
- Third-party Login
- Payment Service Provider
- Shipping and Counter Service Provider

1. Get Username and Password
2. Login Success
3. Login Unsuccessful
4. Enter Username and Password
5. Get Username and Password
6. Login Success
7. Login Unsuccessful
8. Enter Mobile No
9. Add Product to Cart
10. View Cart
11. Buy Now
12. Enter Delivery Address
13. Address Added for Delivery
14. Make Payment
15. Enter Card No, Expiry Date and PIN
16. Payment
17. Enter OTP
18. Transaction Successful
19. Order No and Expected Delivery Date
20. Notify Regarding Order
21. Product Received at warehouse
22. Track ID and Other Information
23. Product Received Successfully
24. Transaction UnSuccessful
25. Notify Regarding Transaction Failure
26. Pay by Cash/On Delivery
27. Notify Regarding Cash/On Delivery Order
28. Product Received Successfully
29. Product Received Successfully
30. Product Received Successfully
The successful registration of sellers or courier companies depends on service level agreement (SLA) defined by mutual understanding between parties. This SLA defines interfaces, choreography and any terms and conditions. The service level agreement (SLA), binds both the parties about how a service is to be provided and consumed. Customer works with SLA and contracts as well. Typically these are negotiated via human intervention between the customer and the service provider. The expectations of the customer and the reputation of the service provider are key parts of those negotiations. The service contract needs to be documented to formalize the required processing resources by the individual party. These contracts can be best modeled as SoaML service contract diagram as shown in Fig. 5.2.

![FIGURE 5.2: A SoaML service contract diagram involving multiple parties](image)

### 5.1.1 Mapping of Service Choreography Message With Web Service

By looking up the choreography messages one can carry out a generic mapping of service choreography messages with web services. Fig. 5.3 shows this mapping along with choreography message fragments. It maps the input message to the corresponding simple object access protocol (SOAP) request message of web service and output messages with simple object access protocol (SOAP) response messages. A static analysis reveals that the message set \{1, 1.1, 1.2\} corresponds to the `product_registration()` service, message set \{2, 3, 3.1\} corresponds to the `couriercompany_registration()` service, message set \{4, 4.1, 4.2\} corresponds to the `login()` service, message set \{5, 5.1, 6, 6.1, 7.1\} corresponds to the `third-party-login()` service, message set \{8, 8.1, 8.1.1, 8.1.1.1, 8.1.1.2\} corresponds to the `signup()` service, message set \{8.1.1.2.1, 8.1.1.2.1.1, 8.1.1.2.1.1.1, 8.1.1.2.1.1.1.1\} corresponds to the `add_to_cart()` service, message set \{11, 12, 12.1, 12.1.1, 12.1.1.1, 14, 15, 15.1, 15.1.1, 16, 17, 17.1, 18, 19, 20\} corresponds to the `make_payment()` service, and message set \{25, 25.1, 26, 26.1, 27, 27.1, 28, 29, 29.1\} corresponds to the `make_courier()` service. This mapping is useful to run and analyze the web service with corresponding SOAP messages.
FIGURE 5.3: A static mapping of service choreography messages with web services
5.2 Basic Definitions

Before presenting our global dynamic slicing algorithm, we introduce a few definitions that would be used in our algorithm. Also, we extend the definitions pertaining to compute dynamic slice.

**Definition 5.1: def(m)**

Let \( m \) be a message node in service-oriented software choreography (SOSC). A node \( n \) is said to be \( \text{def}(m) \) node if \( n \) defines (assigns) values to variables of message node \( m \).

Consider the service-oriented software choreography (SOSC) of Fig. 5.1 and its service-oriented software dependency graph (SOSDG) of Fig. 5.5. In the SOSDG of Fig. 5.5, node 1 is the \( \text{def}(\text{product name}) \) node.

**Definition 5.2: use(m)**

Let \( m \) be a message node in SOSC. A node \( n \) is said to be \( \text{use}(m) \) node if \( n \) uses the values of variables assigned by message node \( m \).

In the SOSDG of Fig. 5.5, nodes 1.1 and 1.2 are the \( \text{use}(\text{product name}) \) node.

**Definition 5.3: recentDef(m)**

For each message \( m \), \( \text{recentDef}(m) \) represents the node corresponding to the most recent definition of the message variable \( \text{def}(m) \) in particular service execution history (SEH).

In Fig. 5.5, \( \text{recentDef}(\text{product name}) \) is same at any message nodes, since there is no message node which changes the value of message variable \( \text{product name} \). Hence, \( \text{recentDef}(m) \) is same as \( \text{def}(m) \).

**Definition 5.4: Web Service Control Flow Graph (WSCFG)**

A web service control flow graph (WSCFG) of a service-oriented software choreography \( SOSC \) is a directed graph \( (N, E, \text{Start, Stop}) \), where each node \( n \in N \) represents message of service choreography \( SOSC \), while each edge \( e \in E \) represents control transfer among nodes. Nodes \( \text{start} \) and \( \text{stop} \) are two unique nodes representing entry and exit of the service-oriented software choreography, respectively. There is a directed edge from node \( a \) to node \( b \) if control may flow from node \( a \) to node \( b \). The WSCFG of the service-oriented software given in Fig. 5.1 (through a SoaML sequence diagram) is shown in Fig. 5.4.

**Definition 5.5: Service Execution Case**

A test case is a triplet \([I,S,O]\) where \( I \) is the input to the system at state \( S \) and \( O \) is the
expected output. It consists of run-time input values read by the sequential program. This definition is insufficient for SOA-based software which gives various challenges due to its inherent features like dynamic binding, agility and many others.

We define service execution case (SEC) as the set of information required to guarantee repeatability. By repeatability, we mean that each service executes same message and execution of each message sees the same values for each of the service variables.

**Definition 5.6: Service Execution Point**
In the sequential program, the execution point is a point in that flow of execution. An execution point is defined for a process and for the occurrence of statement that has been executed by this process. Because of multiple flow of execution in service-oriented software (SOS) we define a service execution point (SEP) by a pair \((S_k, M^j_i)\) where \(M^j_i\) is the \(j\)th occurrence of message \(M_i\) executed by the web service \(S_k\).

This definition imposes that the service has already been executed that occurrence of the message or currently executing it. It allows the same messages to be executed by various services.

**Definition 5.7: Service Execution History**
An execution history is defined for specific execution case. A service execution history (SEH) is the sequence of service execution points (SEPs) in the order in which they are executed by the services.

**Definition 5.8: Service Choreography Execution History**
It merges the service execution history (SEH) of each service into a single set. Sorting such set is difficult due to distributed environment. But it is important to maintain the order in which one SEP has influenced the execution of another. It is possible to have more than one service choreography execution history (SCEH) for a given service execution case.

**Definition 5.9: Slicing Criterion for Service-Oriented Software Choreography**
We define the slicing criterion for service-oriented software choreography (SOSC) as the triplet \((var, (S_k, M^j_i), SEC)\) where \(var\) is the message variable used at \(m\), and \((S_k, M^j_i)\) is the service execution point (SEP) with input service execution case (SEC).
FIGURE 5.4: The WSCFG of the service choreography given in Fig. 5.1
Now, we define some new dependencies like intra-service and inter-service dependencies which arise in case of service-oriented software. Then, we define local dynamic slice and global dynamic slice.

**Definition 5.10: Intra-Service Dependency**
In a sequential program, the occurrence of the statements on which the current statement is dependent has already been executed. This is not necessarily correct for *service-oriented software (SOS)*. Intra-Service Dependencies are used to indicate that the state of service at that point depends on the execution of a message by another service. We define *intra-service dependence edge* \((n_i,n_j)\) as an edge denoting that:

1. the two nodes \((n_i,n_j)\) are being executed by two distinct services, \(S_i\) and \(S_j\) respectively; and
2. the state of service \(S_i\) at node \(n_i\) directly depends on the execution of the node \(n_j\) by service \(S_j\).
3. These services \(S_i\) and \(S_j\) are being provided by a single service provider.

Intra-service dependencies can also reflect data or control dependencies.

**Definition 5.11: Inter-Service Dependency**
We define *inter-service dependence edge* \((n_i,n_j)\) as an edge denoting that:

1. the two nodes \((n_i,n_j)\) are being executed by two distinct services \(S_i\) and \(S_j\) respectively; and
2. the state of service \(S_i\) at node \(n_i\) directly depends on the execution of the node \(n_j\) by service \(S_j\).
3. These services \(S_i\) and \(S_j\) are being provided by more than one service provider.

Inter-service dependencies may also reflect data or control dependencies.

The global dynamic slice computation is based on all the types of dependencies that have been defined earlier. We define two types of dynamic slices *global dynamic slice* and *local dynamic slice*.

**Definition 5.12: Global Dynamic Slice of Service-Oriented Software Choreography**
We define *global dynamic slice* of *service-oriented software choreography (SOSC)* with respect to the slicing criterion \((\text{var},(S_k,M^j_i),\text{SEC})\) as the subset of *service-oriented software*.
choreography (SOSC) messages whose execution really affected the value of message variable \( \text{var} \), as observed at the service execution point \( (S_k, M^j_i) \), for the service execution case \( SEC \).

**Definition 5.13: Local Dynamic Slice of Service-Oriented Software Choreography**

We define *local dynamic slice* for a service \( S_l \) with respect to a slicing criterion \( (\text{var}, (S_k, M^j_i), SEC) \) as the subset of service \( S_l \) messages whose execution really affect the value of the given message variable \( \text{var} \), as observed at the service execution point \( (S_k, M^j_i) \), for the service execution case \( SEC \). It is a way to filter out messages other than executed by service \( S_l \).

### 5.3 Service-Oriented Software Dependence Graph (SOSDG): Our Intermediate Representation of Service-Oriented Software

This section introduces a method for efficient representation of service-oriented software in SOA environment. This representation is later used to compute dynamic slices. We named this representation *service-oriented software dependence graph* (SOSDG). Each message in a sequence diagram is represented as a node along with their number in SOSDG. This message node also maps with corresponding input or output message of web service. This SOSDG captures control dependencies from static analysis of sequence diagram. It also captures data, intra-service and inter-service dependencies from run-time analysis of corresponding web service execution. The inter-service dependencies may cross organizational boundaries. We also depict web service nodes for simplifying the SOSDG along with mapped numbers. The web service node may belong to more than one service provider. Fig. 5.5 shows the SOSDG for Fig. 5.1.

Further, *service-oriented software dependence graph* (SOSDG) can be defined for a *service choreography execution history* (SCEH). It is a tuple \( (S, G) \), where \( S \) is the set of service nodes and \( G \) is the set of *web service control flow graph* (WSCFG). The web service control flow graph (WSCFG) is a tuple \( (M, A) \), where \( M \) is the set of message nodes and \( A \) is the set of dependence edges that we defined earlier. The web service control flow graph (WSCFG) shows the control dependencies of message occurrence within that graph. In addition data, intra-service and inter-service dependence connect occurrence of messages of distinct graphs. This set of web service dependence graphs and the edges between them form the *service-oriented software dependence graph* (SOSDG). For any instance of service choreography execution history (SCEH), it contains all the web services which have executed at least one message in that \( SCEH \). If a web service is contained in that set, its
A web service control flow graph (WSCFG) is a subgraph of the service-oriented software dependence graph (SOSDG). When a web service executes its first message, it is added to $S$ and this message forms the first node of its WSCFG. When a web service executes other messages, its WSCFG is updated.

FIGURE 5.5: The SOSDG of the service choreography given in Fig. 5.1
5.4 Marking Based Global Dynamic Slicing (MBGDS) Algorithm

In this section, first, we briefly describe our MBGDS algorithm. Then, we present the pseudo-code of the algorithm. Subsequently, we discuss the complexity of our algorithm.

5.4.1 Overview of the MBGDS Algorithm

We first provide a brief overview of our global dynamic slicing algorithm. Before execution of a service-oriented software choreography (SOSC) and its services, WSCFG and SOSDG are constructed statically. We permanently mark the control dependence edges as they don’t change during the execution of services. We consider all the data dependence edges, intra-service dependence edges and inter-service dependence edges for marking and unmarking during run-time. During execution of the SOSC and services, we mark an edge when its associated dependence exists, and unmark when its associated dependence ceases to exist.

After each message $m$ is executed, we unmark all incoming marked dependence edges excluding the control dependence edges, associated with the service $S_i$, corresponding to the previous execution of the node $m$. Then, we mark the dependence edges corresponding to the present execution of the node $m$.

During the execution of the service-oriented software choreography (SOSC), let $Global\_Dynamic\_Slice(s,m)$ with respect to the slicing criterion $(s,m)$ denotes the global dynamic slice with respect to the most recent execution of the node $m$, for given SEC. Let $\{(z_1,m), (z_2,m), \ldots, (z_k,m)\}$ be all the marked incoming dependence edges of $m$ in the updated SOSDG after the execution of message $m$. Then, it is clear that global dynamic slice with respect to the present execution of the node $m$, for the service $S_i$, with input $SEC$, is given by

\[
Global\_Dynamic\_Slice(s,m) = \{z_1, z_2, \ldots, z_k\} \cup Global\_Dynamic\_Slice(z_1,m) \cup Global\_Dynamic\_Slice(z_2,m) \cup \ldots \cup Global\_Dynamic\_Slice(z_k,m)
\]

Let $\{m_1, m_2, \ldots, m_k\}$ be all the message variables used or defined as node $m$. Then, we define global dynamic slice of the message $m$ as

\[
Global\_Dynamic\_Slice(s,m) = Global\_Dynamic\_Slice(m_1,m) \cup Global\_Dynamic\_Slice(m_2,m) \cup \ldots \cup Global\_Dynamic\_Slice(m_k,m)
\]
Our slicing algorithm works in three main phases:

**Phase 1:** Construction of the intermediate representation graph,

**Phase 2:** Managing the SOSDG at run-time, and

**Phase 3:** Computing and displaying the global dynamic slice.

In Phase 1 of our MBGDS algorithm, the WSCFG is constructed from a static analysis of the SOSC. Also at this stage, using the WSCFG the static SOSDG is constructed. The Phase 2 of the algorithm is responsible for maintaining the SOSDG during run-time. The maintenance of the SOSDG at run-time involves marking and unmarking the different dependencies such as data dependencies, control dependencies, intra-service dependencies and inter-service dependencies as they arise and ceases. Phase 3 is responsible for computing the global dynamic slice for a given slicing criterion using updated SOSDG. It computes the global dynamic slice through table look up. So, when a request for a slice is made, it is obtained immediately, as it is already available in a table. Thus, after statically constructing the SOSDG of a given service \( S_i \) in a service-oriented software choreography (SOSC), our global dynamic slicing algorithm can compute global dynamic slice with respect to any given slicing criterion.

We now present our MBGDS algorithm for service-oriented software choreography (SOSC) in the form of pseudo-code.

**Algorithm** : Marking Based Global Dynamic Slicing (MBGDS) Algorithm.

**Input** : Service-Oriented Software Choreography (SOSC) and Slicing Criterion \((\text{var}, (S_k,M^i_k), \text{SEC})\) // Slicing criterion given during run-time

**Output** : Global Dynamic Slices // Global dynamic slices are extracted during run-time

**Phase 1: Constructing Static Graphs WSCFG and SOSDG**

1. WSCFG Construction
   
   (a) Node Construction
      
      i. Create two special nodes start and stop.

      ii. For each message \( m \) of a service-oriented software choreography (SOSC) do the followings:
A. create a node $m$.
B. Initialize the node with message variables used or defined.

(b) Add control flow edges

for each node $n_i$ do the following

for each node $n_j$ do the following

Add control flow edge $(n_i, n_j)$ if control flow from node $n_i$ to node $n_j$.

2. SOSDG Construction

(a) Add control dependence edges

for each test(predicate) node $n_i$ do the following

for each node $n_j$ in the scope of $n_i$ do the following

Add control dependence edge $(n_i, n_j)$ and mark it.

(b) Add data dependence edges

for each node $n_i$ do the following

for each message variable used at $n_i$ do the following

for each reaching definition $n_j$ of message variable do the following

Add data dependence edge $(n_i, n_j)$ and unmark it.

(c) Add intra-service dependence edges

for each node $n_i$ in service $S_i$ do the following

for each node $n_j$ in service $S_j$ do the following

Add intra-service dependence edge $(n_i, n_j)$ if edge is either data or control dependence edge and the state of service $S_i$ at node $n_i$ directly depends on the execution of the node $n_j$ by service $S_j$ and both services are provided within organization. Unmark it.

(d) Add inter-service dependence edges

for each node $n_i$ in service $S_i$ do the following

for each node $n_j$ in service $S_j$ do the following
Add *inter-service dependence edge* \((n_i, n_j)\) if edge is either data or control dependence edge and the state of service \(S_i\) at node \(n_i\) directly depends on the execution of the node \(n_j\) by service \(S_j\) and both services are provided by more than one service providers. Unmark it.

**Phase 2: Managing SOSDG at run-time**

1. **Initialization:** Do the following before execution of each message \(m\) of services \(S_i\) of the service-oriented software choreography (SOSC), consisting of set \((S_1, S_2, \ldots, S_k)\).
   
   (a) Set \(\text{Global\_Dynamic\_Slice}(s, m) = \phi\) for every message \(m\) used or defined at every node \(m\) of the SOSDG.
   
   (b) Set \(\text{recentDef}(m) = \text{NULL}\) for each message variables in service \(S_i\).

   // end of initialization

2. **Run-time Updation of SOSDG:** Run the web services and carry out the following after each message \(m\) of the service corresponding to the SOSC \(s\), and SEC for each \(S_i\) of SOSC gets executed.

   (a) Unmark all incoming marked dependence edges excluding the control dependence edges, if any, associated with message \(m\) of the service \(S_i\), corresponding to the previous execution of the node \(m\).

   (b) Update data dependencies: For every message variable used at node \(m\), mark the incoming data dependence edge corresponding to the most recent definition \(\text{recentDef}(m)\) of the service \(S_i\) in SEH.

   (c) Update intra-service dependencies: If \(m\) is \(\text{use}(m)\) node, then mark the incoming intra-service dependence edge, if any, corresponding the associated \(\text{def}(m)\) node which belongs to same party.

   (d) Update inter-service dependencies: If \(m\) is \(\text{use}(m)\) node, then mark the incoming inter-service dependence edge, if any, corresponding the associated \(\text{def}(m)\) node which belongs to other party.

   (e) Update the global dynamic slice for different dependencies:

      i. **Handle data dependency:** Let \(\{(d_1, m), \ldots, (d_j, m)\}\) be the set of marked incoming data dependencies to \(m\). Then,
Global Dynamic Slice(m) = \{d_1, d_2, \ldots, d_j\} \cup \text{Global Dynamic Slice}(d_1) \cup \text{Global Dynamic Slice}(d_2) \cup \ldots \cup \text{Global Dynamic Slice}(d_j),
where \( d_1, d_2, \ldots, d_j \) are the initial nodes of the corresponding marked incoming edges of \( m \).

ii. Handle control dependency: Let \((c, m)\) be the marked control dependence edge. Then,
\[
\text{Global Dynamic Slice}(m) = \text{Global Dynamic Slice}(m) \cup \{c\} \cup \text{Global Dynamic Slice}(c).
\]

iii. Handle intra-service dependency: Let \( m \) be a \( \text{use}(m) \) node and \((x, m)\) be the marked intra-service dependence edge associated with corresponding \( \text{def}(m) \) node \( x \) within organization. Then,
\[
\text{Global Dynamic Slice}(m) = \text{Global Dynamic Slice}(m) \cup \{x\} \cup \text{Global Dynamic Slice}(x).
\]

iv. Handle inter-service dependency: Let \( m \) be a \( \text{use}(m) \) node and \((y, m)\) be the marked inter-service dependence edge associated with corresponding \( \text{def}(m) \) node \( y \) across an organization. Then,
\[
\text{Global Dynamic Slice}(m) = \text{Global Dynamic Slice}(m) \cup \{y\} \cup \text{Global Dynamic Slice}(y).
\]

Phase 3: Computing and displaying the global dynamic slice

1. Global Dynamic Slice Computation:

   (a) For every message variable \( m \) used at node \( m \) do the following:
       Let \((d, m)\) be a marked data dependence edge corresponding to the most recent definition of the message variable, \((c, m)\) be the marked control dependence edge, \((x, m)\) be the marked intra-service dependence edge, and \((y, m)\) be the marked inter-service dependence edge. Then,
       \[
       \text{Global Dynamic Slice}(s, m) = \{d, c, x, y\} \cup \text{Global Dynamic Slice}(d) \cup \text{Global Dynamic Slice}(c) \cup \text{Global Dynamic Slice}(x) \cup \text{Global Dynamic Slice}(y)
       \]
   
   (b) For message variable defined at node \( m \), do
       \[
       \text{Global Dynamic Slice}(s, m) = \text{Global Dynamic Slice}(m) \cup \{m\}.
       \]

2. Global Dynamic Slice Look Up

   (a) If a slicing command \((s, m)\) is given for a service-oriented software choreography \( \text{SOSC}, \text{SEC} \) and for particular message variable \( \text{var} \), carry out the followings:
i. Look up $Global\_Dynamic\_Slice(s, m)$ for the content of the slice.

ii. Display the resulting slice.

(b) If the services of $SOSC$ has not terminated, go to Step 2 of Phase 2.

### 5.4.2 Working of MBGDS Algorithm

We are interested in computing the global dynamic slice of service-oriented software choreography shown in Fig. 5.1 with respect to the slicing criterion $(order \, no, (S_8, M 11^1))$. {seller name= XY, contact address= China, product name=AB, quantity=1000, price=12999, user name= YZ, password= temp_123456}). The updated $SODG$ after applying Phase 2 of our algorithm is shown in Fig. 5.6. All the marked edges are shown in bold lines and the message nodes included in the dynamic slice are shown as red colored vertices in Fig. 5.6. We will explain how our algorithm for computing global dynamic slices of service-oriented software works. For the input given in SEC, our MBGDS algorithm executes the message nodes \{1, 1.1, 8.1.1.2.1, 8.1.1.2.1.1, 8.1.1.2.1.1.1, 8.1.1.2.1.1.1.1, 9, 9.1, 4, 4.1, 9.2, 11\}.

Now, $(Global\_Dynamic\_Slice(S_8, M 11^1)) = Global\_Dynamic\_Slice(9.2) \cup \{11\}$. On evaluating the expression recursively, we obtain the global dynamic slices as follows:

$Global\_Dynamic\_Slice(9.2) \cup \{11\} = \{1, 1.1, 8.1.1.2.1, 8.1.1.2.1.1, 8.1.1.2.1.1.1, 9, 9.1, 4, 4.1, 9.2, 11\}$

$Global\_Dynamic\_Slice(9.2) = Global\_Dynamic\_Slice(9.1) \cup Global\_Dynamic\_Slice(9.2) \cup \{9.2\} = \{1, 1.1, 8.1.1.2.1, 8.1.1.2.1.1, 8.1.1.2.1.1.1, 9, 4, 4.1, 9.2\}$

$Global\_Dynamic\_Slice(4.1) = \{4, 4.1\} \cup Global\_Dynamic\_Slice(9.1) = \{1, 1.1, 8.1.1.2.1, 8.1.1.2.1.1, 8.1.1.2.1.1.1, 9, 4, 4.1\}$

$Global\_Dynamic\_Slice(9.1) = Global\_Dynamic\_Slice(4.1) \cup Global\_Dynamic\_Slice(9) \cup \{9\} = \{1, 1.1, 8.1.1.2.1, 8.1.1.2.1.1, 8.1.1.2.1.1.1, 9\}$

$Global\_Dynamic\_Slice(9) = \{8.1.1.2.1.1.1\} \cup Global\_Dynamic\_Slice(8.1.1.2.1.1.1) \cup Global\_Dynamic\_Slice(9) = \{1, 1.1, 8.1.1.2.1, 8.1.1.2.1.1, 8.1.1.2.1.1.1\}$

$Global\_Dynamic\_Slice(8.1.1.2.1.1.1) = \{8.1.1.2.1.1\} \cup Global\_Dynamic\_Slice(8.1.1.2.1.1.1) = \{1, 1.1, 8.1.1.2.1, 8.1.1.2.1.1\}$
Global_Dynamic_Slice(8.1.1.2.1.1) = Global_Dynamic_Slice(8.1.1.2.1) \cup \{8.1.1.2.1\} = \{1, 1.1, 8.1.1.2.1\}

Global_Dynamic_Slice(8.1.1.2.1) = Global_Dynamic_Slice(1.1) \cup Global_Dynamic_Slice(8.1.1.2.1) \cup \{1.1\} = \{1, 1.1, 8.1.1.2.1\}

Global_Dynamic_Slice(1) = Global_Dynamic_Slice\{1.2\} \cup Global_Dynamic_Slice\{1\} \cup \{\phi\} = \{1, 1.1\}

Thus, the Global_Dynamic_Slice(S_b, M_{11^1}) consists of the message node set \{1, 1.1, 8.1.1.2.1, 8.1.1.2.1.1, 8.1.1.2.1.1.1, 8.1.1.2.1.1.1.1, 9, 9.1, 4, 4.1, 9.2, 11\}.

FIGURE 5.6: The updated SOSDG of the service choreography given in Fig. 5.1
5.4.3 Correctness of MBGDS Algorithm

In this section, we sketch the proof of correctness of our MBGDS algorithm.

**Theorem 5.1** MBGDS algorithm always finds a correct global dynamic slice with respect to a given slicing criterion.

**Proof.** The proof is given through mathematical induction. Let \( S \) be a service-oriented software choreography (SOSC) for which a global dynamic slice is to be computed using MBGDS algorithm. For any given set of input values to \( S \), the global dynamic slice with respect to the first executed message is certainly correct, according to the definition. From this, we can argue that, the global dynamic slice with respect to the second executed message is also correct. During execution of the service-oriented software choreography (SOSC) \( S \), assume that the algorithm has computed correct global dynamic slices prior to the execution of a message \( m \). To complete the proof, we need only to show that the global dynamic slice computed after execution of the message \( m \) is correct. Note that the messages that affect the execution of the message \( m \) must have been executed prior to this execution of the message \( m \). It is obvious that the global dynamic slice \( \text{Global Dynamic Slice}(s,m) \) contains all those messages which have affected the current value of the message variables \( \text{var} \) used at \( m \), since our MBGDS algorithm has marked all the incoming edges to \( m \) only from those nodes on which node \( m \) is dependent. The Steps 2(b), 2(c) and 2(d) of Phase 2 of the MBGDS algorithm ensure that the node \( m \) is dependent (with respect to its present execution) on a node \( n \) if and only if the edge \( (m,n) \) is marked in the SODG of the service-oriented software choreography (SOSC) \( S \). If a node has no effect on the value of the message variables \( \text{var} \), then it will not be included in the global dynamic slice \( \text{Global Dynamic Slice}(s,m) \). So, \( \text{Global Dynamic Slice}(s,m) \) is a correct global dynamic slice. In other words, we can say that the global dynamic slices computed prior to this execution of the message \( m \) are correct. Therefore, the Steps 2(e(i)), 2(e(ii)), 2(e(iii)), 2(e(iv)) of Phase 2 and Steps 1(a) and 1(b) of Phase 3 of the MBGDS algorithm ensure that the global dynamic slices computed after execution of the message \( m \) are correct. Further Step 2(b) of Phase 3 of the MBGDS algorithm guarantees that the algorithm stops when execution of the program \( P \) terminates. This establishes the correctness of the algorithm.

5.4.4 Salient Features of MBGDS Algorithm

The important features of the MBGDS algorithm are listed below.

- It computes correct global dynamic slices with respect to any valid slicing criterion.
• It can traverse inter-service dependency, intra-service dependency and extracts composite service like buying service in Section 5.1 using MBGDS.

• No trace files are generated. All information are maintained and updated dynamically for all services and are discarded at run-time of a SOSC on termination of services.

• It does not create any additional message nodes during run-time. This saves the expensive message node creation steps.

• When a request for a slice is made, it is easily available through slicer service.

• No serialization of the events of the services are required due to slicing based on the sequence diagram.

• It can be easily extended to accommodate dynamic slices of cloud-based programs.

### 5.4.5 Complexity Analysis of MBGDS Algorithm

In the following section, we analyze the time and space complexities of our MBGDS algorithm.

**Time complexity:**
To determine the time complexity of our MBGDS algorithm, we have considered barometer instructions which significantly contributes for the computation of the slice. The first instruction is related to the time required for running web services and updation of SOSDG. The second instruction corresponds to the time required to look up the data structure to retrieve the slice. Let \( n \) be the total number of messages of the web services. Then, \( O(n^2) \) time is required to compute and update SOSDG. Let \( m \) be the length of service choreography execution history (SCEH) involved in SOSC. Then, the run-time complexity of the MBGDS algorithm would be \( O(n^2m) \). The algorithm takes constant amount of time i.e., \( O(1) \) for slice look up, which is negligible.

**Space complexity**
Let a service-oriented software choreography (SOSC) have \( n \) messages. The space complexity for step 1 of Phase 1 would be \( O(n^2)+O(n)+O(3) \). The required space for step 2 of Phase 2 would be \( O(n^2)+O(n^3)+O(n^2) \).

For \( n \) messages in SOSC, the space required to store global dynamic slice would be \( O(n^3) \) and \( O(n^2) \) space is required to store the recentDef. So, the space complexity of our MBGDS algorithm would be \( O(n^3) \).
5.5 Implementation and Experimental Results

In this section, we present a tool for SOSDG construction and discuss the experimental results obtained using our tool.

5.5.1 SOSDS: An SOS Dynamic Slicing Tool

In this section, we present a brief description of a tool which we have developed to implement our global dynamic slicing algorithm for service-oriented software (SOS). We have named our tool Service-Oriented Software Dynamic Slicer (SOSDS). Our tool can compute the global dynamic slice of a service-oriented software with respect to any given slicing criterion. Currently, the SOSDS supports inter-service communication and intra-service communication using WSDL. In the following, we briefly discuss the design, implementation, and working of our slicing tool.

5.5.2 Design of SOSDS

The high-level design of our implementation SOSDS has been depicted in Fig. 5.7. The SOSDS takes a SoaML sequence diagram comprising the parties and their interactions, in XMI format as input. This is parsed by the DOM parser module, which gathers information regarding parties participating in interactions along with the messages exchanged between them. The DOM parser reads the entire input XMI file and creates a tree structure in memory. When the DOM parser encounters XML tags in the XMI, it parses the tag to describe what type of tags were encountered. The information obtained using the DOM parser module is then used to initialize all the data structures needed to construct the static WSCFG as stated in Phase 1 of our MBGDS algorithm.

Further, SOAP messages are generated during run-time execution of services and its clients serve as input to our SOSDS. These inputs help the interceptor module to intercept SOAP messages exchanged among services during run-time. The intercepted SOAP messages provide information like time-stamp, request encoding, request preamble, request length, response encoding, response preamble, response length and more importantly, SOAP message body along with HTTP headers. This runtime information helps in initializing the data structure needed to construct dynamic SOSDG as stated in Phase 2 of our MBGDS algorithm.

The GUI module constructs WSCFG and SOSDG of the SoaML model in consultation
with DOM parser and Interceptor module. The *slicer service* modules takes the slicing criterion as input from the *slicer service client* and outputs the computed global dynamic slice. The GUI module updates the graph to reflect the computed global dynamic slices.

FIGURE 5.7: Schematic design of SOSDS
5.5.3 Implementation

We have implemented our MBGDS algorithm for web services written in Java. Our dynamic slicing tool is coded in Java and uses interceptor module of WS monitor [236]. When the web service and their clients are made to run our slicing tool SOSDS, first the interceptor module intercepts the SOAP messages and stores the data in a hashmap called message\_info. Meanwhile, the DOM parser module analyzes the XML tags and construct the WSCFG from a SoaML sequence diagram (XMI representation) given as input. Using the WSCFG and the message\_info, the SOSDG is constructed statically. While constructing the SOSDG, we store the data in a hashmap called service\_info. Each of this service\_info contains information: time-stamp, request encoding, request preamble, request length, response encoding, response preamble, response length, SOAP message body, HTTP headers. For constructing the SOSDG, we have used the following flags: data\_flow\_flag, control\_flow\_flag, intra-service\_flag, inter-service\_flag etc. For storing the SOSDG we have used the hashmap: Map sosdg = new HashMap(); If there is an edge from message node i to j, then execute sosdg.put("i","j", 1).

After constructing the SOSDG statically, we run the services along with their clients. After execution of each message, we invoke the update\_global\_dynamic\_slice() method, which marks and unmarks the edges of SOSDG appropriately and updates the global dynamic slice. When the global dynamic slice of a message node and service is requested our slicer SOSDG provides the global dynamic slice for the given slicing criterion and also visualizes the SOSDG. The visualization of both the graphs, WSCFG, and SOSDG was carried out by using JGrpahT library [108].

5.5.4 Data Sets Used in Our Experiments

As per our knowledge there does not exist any benchmark data sets to validate service-oriented software (SOS). Due to non-availability of such benchmark models, we use example available from the assignment submissions of the service-oriented computing course of my department. The case study was implemented in a laboratory. A batch of 20 students taking a laboratory assignment in their service-oriented computing course at VGEC, Chandkheda were considered. They were first asked to carry out analysis and design a SoaML sequence diagram for a system description given in exercises of Chapter 6 of the software engineering book [194]. They were instructed to implement each of the systems as a service-oriented software system and to use JAX-WS API to create web services. Even, we gave flexibility to choose systems based on their own choice from any real-world examples. Such real-world SOA examples have been described in Appendix-A. The SoaML sequence diagrams were
constructed using visual paradigm [230], and were exported in corresponding XMI files. For the experiment, we selected important SoaML sequence diagrams (XMI representations), web services along with associated clients submitted by them. The maximum XMI file size was up to $15708$ lines of tag (LOT) involving $11$ services in service choreography with maximum up to $2145$ lines of code (LOC). However, currently, SOSDS can handle only intra-service and inter-service dependencies. We will extend it to handle the composition and exception handling dependencies, in our future work.

5.5.5 Experimental Results

We have tested the working of MBGDS algorithm using the case study examples as stated in [194] with inter-service and intra-service dependencies using SOSDG.

The system configuration used to run MBGDS algorithm is windows 7 professional service pack 1, intel(R) core(TM) i3-3240 CPU@ 3.40GHz running at 3.40 GHz, with 4.00 GB RAM. We studied the run-time requirements of our MBGDS algorithm for these case studies and for several runs. All measured times reported in this section are overall times, including parsing and building of the both WSCFG and SOSDG representation. Table 5.1 summarizes the average run-time requirements of MBGDS algorithm. As we are not aware of the existence of any algorithm for dynamic slicing of service-oriented programs, so we have not presented any comparative results. We have presented only the results obtained from our experiments. Since we computed the global dynamic slices at different messages of a services, we have calculated the average run-time requirements of the MBGDS algorithm.

From the experimental results, it can be observed that the average run-time increases sublinearly as the number of services increase in a service choreography as shown in Fig. 5.8. Average run-time is linear after 1K LOC, also in Fig. 5.8 average run-time is linear after number of services reach 6. This is because of the hash table data structure we used for implementation, and the program characteristics. Fig. 5.8 shows good response time for small no. of services and LOCs but for large no. of services and LOCs average run-time increases significantly. In Table 5.1, as the LOC increases the run-time increases linearly. Similarly, when the service and XMI (LOT) increase, average run-time increases linearly. The performance results of our implementation agree with the theoretical analysis.
<table>
<thead>
<tr>
<th>Sl. No.</th>
<th>Name of Case-Study</th>
<th>XMI (#LOT)</th>
<th># Services</th>
<th>#LOC</th>
<th>Average Run-Time (in Sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Travel Reservation System (TRS) [Appendix-A]</td>
<td>20701</td>
<td>20</td>
<td>3122</td>
<td>54.42</td>
</tr>
<tr>
<td>2</td>
<td>Online Shopping System (OSS) [Section 5.1]</td>
<td>15708</td>
<td>11</td>
<td>2145</td>
<td>37.45</td>
</tr>
<tr>
<td>3</td>
<td>Bookshop Automation Software (BAS)</td>
<td>9150</td>
<td>6</td>
<td>1189</td>
<td>21.06</td>
</tr>
<tr>
<td>4</td>
<td>Library Information System (LIS)</td>
<td>9240</td>
<td>6</td>
<td>1190</td>
<td>21.09</td>
</tr>
<tr>
<td>5</td>
<td>Restaurant Automation System (RAS)</td>
<td>5789</td>
<td>5</td>
<td>986</td>
<td>15.01</td>
</tr>
<tr>
<td>6</td>
<td>Student’s Auditorium Management Software (SAMS)</td>
<td>5333</td>
<td>5</td>
<td>988</td>
<td>15.00</td>
</tr>
<tr>
<td>7</td>
<td>Municipality Garbage Collection Automation Software (MGCAS) [Appendix-A]</td>
<td>5278</td>
<td>5</td>
<td>989</td>
<td>15.00</td>
</tr>
<tr>
<td>8</td>
<td>Hotel Automation Software (HAS)</td>
<td>5712</td>
<td>4</td>
<td>795</td>
<td>13.90</td>
</tr>
<tr>
<td>9</td>
<td>Road Repair and Tracking System (RRTS)</td>
<td>5812</td>
<td>4</td>
<td>810</td>
<td>14.10</td>
</tr>
<tr>
<td>10</td>
<td>Judiciary Information System (JIS)</td>
<td>5101</td>
<td>4</td>
<td>790</td>
<td>13.80</td>
</tr>
<tr>
<td>11</td>
<td>Supermarket Automation Software (SAS)</td>
<td>5119</td>
<td>4</td>
<td>789</td>
<td>13.75</td>
</tr>
<tr>
<td>12</td>
<td>Medicine Shop Automation Software (MSAS)</td>
<td>5219</td>
<td>4</td>
<td>794</td>
<td>13.89</td>
</tr>
<tr>
<td>13</td>
<td>Railway Reservation Software (RRS) [Appendix-A]</td>
<td>5318</td>
<td>4</td>
<td>809</td>
<td>14.09</td>
</tr>
<tr>
<td>14</td>
<td>Software Component Cataloguing Software (SCCS)</td>
<td>5400</td>
<td>3</td>
<td>589</td>
<td>13.01</td>
</tr>
<tr>
<td>15</td>
<td>Motor Parts Shop Software (MPSS)</td>
<td>5139</td>
<td>3</td>
<td>580</td>
<td>12.95</td>
</tr>
<tr>
<td>16</td>
<td>House Rental Software (HRS) [Appendix-A]</td>
<td>2411</td>
<td>2</td>
<td>397</td>
<td>11.80</td>
</tr>
</tbody>
</table>
Section 5.6 Comparison With Related Work

To the best of our knowledge, no algorithm for dynamic slicing of service-oriented software has been proposed so far. We, therefore, compare the performance of our algorithm with the existing algorithms for static or dynamic slicing of object-oriented software.

Static slicing of software architecture has drawn the attention of many researchers [92, 100, 101]. Also, dynamic slicing of software architecture and UML models has been addressed by many researchers [99, 131]. Zhao [101] was the first one to introduce the concept of architectural slicing based on architectural description language (ADL) ACME. He defined component-connector dependency, connector component dependency, and additional dependencies. He proposed a two-phase algorithm to compute architectural slice based on software architectural dependency graph (SADG). He extended his previous work [100] by introducing architectural information flow graph with information flow arcs like component-connector, connector-component, and internal flow arcs based on Wright ADL.

Kim et al. [217] introduced dynamic software architecture slicing (DSAS), in which slices represented the run-time behavior of those parts of the software architecture that were selected according to the particular slicing criterion (e.g., a set of resources and their values) provided by the software architect. They defined a slicing criterion for (ADLs) such as ACME, RAPIDE, Aesop, UniCon, and Wright.

All these approaches [92, 100, 101] slices were computed in forward or backward manner. These approaches did not consider the static and dynamic aspects of systems. A shortcoming of these approaches is being addressed by Lallchandani [99, 131]. Lallchandani et al. [99, 131] have used generic class diagram and generic sequence diagram and integrated
them to generate *model dependency graph* (MDG). They proposed an algorithm *architectural model slicing through MDG traversal* (AMSMT) to produce the static and dynamic architectural model slices. The algorithm traversed the edges of *model dependency graph* (MDG) according to the slicing criterion. They developed a tool which computed a dynamic slice from UML architectural models.

Mohapatra [55] computed dynamic slices for object-oriented programs using Java language. They defined dynamic slice *Dynamic Slice*(u; var) with respect to the slicing criterion < s; var> for the most recent execution of the statement s, where var is the variable defined or used at the statement s. They proposed a dynamic slicing algorithm which computed correct dynamic slices.

A comparison with the related work is presented in Table 5.2. In Table 5.2, the comparisons are presented based on slicing entities, slice types and slice output. There is no direct related work available using quantitative evaluation parameter. The only quantitative evaluation parameter we found is the average run-time for computing the slices.

**TABLE 5.2: Comparison with related work**

<table>
<thead>
<tr>
<th>Sl. No.</th>
<th>Related Work</th>
<th>Model/Language Used</th>
<th>Slicing Entities</th>
<th>Slice Type</th>
<th>Slice Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Zhao [101]</td>
<td>Wright ADL</td>
<td>Software Architecture</td>
<td>Static</td>
<td>Sliced Components</td>
</tr>
<tr>
<td>2</td>
<td>Kim et al. [217]</td>
<td>ACME and RAPIDE ADL</td>
<td>Software Architecture</td>
<td>Dynamic</td>
<td>Sliced Components</td>
</tr>
<tr>
<td>3</td>
<td>Mohapatra [55]</td>
<td>Object-Oriented Program</td>
<td>Java Program</td>
<td>Dynamic</td>
<td>Sliced objects</td>
</tr>
<tr>
<td>4</td>
<td>Lallchanandani et al. [131]</td>
<td>UML</td>
<td>Class and Sequence diagram</td>
<td>Dynamic</td>
<td>Sliced objects</td>
</tr>
<tr>
<td>5</td>
<td>Our MBGDS Algo.</td>
<td>SoaML</td>
<td>Services and Sequence diagram</td>
<td>Dynamic</td>
<td>Sliced messages</td>
</tr>
</tbody>
</table>
However, our MBGDS algorithm for global dynamic slicing of service-oriented software incorporates several new things as compared to other work reported in the literature. One new thing is the computation of a global dynamic slice based on both the sequence diagram and web services. The computed slice is based on the dependencies existing among different services that are distributed across the various sites. Slicing based on both model and services can efficiently correlate different services during run-time, and help to understand how changing any one of the services will impact the rest of the service choreography.

5.7 Conclusion

In this chapter, we have proposed a novel algorithm for computing global dynamic slices of service-oriented programs. We have named our algorithm markings based global dynamic slicing (MBGDS). We considered SoaML sequence diagram and services in our work. Our algorithm used service-oriented software dependence graph (SOSDG) as the intermediate representation. The MBGDS algorithm is based on marking and unmarking the edges of the SOSDG as and when the dependencies arise and cease at run-time.

The main advantage of our algorithm is that it does not use any trace file to store the execution history. Also, it does not create additional message nodes during run-time. This saves the expensive file I/O and node creation steps. Another advantage of our approach is that when a request for a slice is made, it is easily available through a mere table lookup. We have developed a slicer to verify the proposed algorithm.

The unavailability of web service code is a major challenge for testing, as web services are just provided as interfaces to the users and systems. This constraint makes black-box testing the only viable solution for testing web services. Therefore, in the next chapter, we are going to carry out black-box testing of web services.