CHAPTER - 4

Static Slicing of Service-Oriented Software

Nowadays many organizations are shifting their focus from technology-oriented business process to SOA (service-oriented architecture) based business process. This paradigm shift occurs due to major advantages offered by SOA over the traditional manual business processes. SOA enables organizations to align their business processes with changing customer needs creating flexible, and agile business environments. Understanding such dynamic SOA enables one to have great insight and comprehension of a business process. Since it is a difficult task, a better understanding of such processes can be obtained through models.

Design models help organizations simplify and visualize business processes. UML has been widely used as a general purpose modeling language for object-oriented systems which is not intended for modeling distributed systems like SOA [96]. Service oriented architecture modeling language (SoaML), an emerging standard for modeling SOA, is specified by object management group (OMG) and is supported by major IT vendors like IBM, Visual Paradigm, Eclipse and many other organizations. With this motivation, we define our objective and scope of our work for this chapter as follows:

• Research Question and Objective
  Our main research question is “are we able to compute the static slices of SOA-based software?” This work partly answers the question by computing static slices for service-oriented software (SOS) using SoaML service interface diagram.

• Scope of Work
  The scope of our work is limited up to SOS testing.

With above defined objective and scope, we present a novel technique for static slicing of service-oriented software (SOS) using SoaML (service oriented architecture
modeling language) service interface diagram. In this technique, a service interface
diagram is converted into an intermediate representation which we have named *service interface dependency graph (SIDG)*. The SIDG identifies *service call dependency* and *composite dependency* from service interface diagram. Giving slicing criterion as input, our algorithm traverses the SIDG and identifies the affected service interface nodes. The novelty of our work lies in the computation of slices based on SIDG and its dependencies induced within SIDG.

We first present some basic concepts and definitions that will be used in our algo-

rithm. Then, we discuss the intermediate program representation: *service interface dependency graph* (SIDG). Next, we present our marking based static slicing (SS-
SIM) algorithm for service-oriented software. Also, we present a brief description of

a slicing tool we have developed to implement our proposed static slicing algorithm. Then, we compare our algorithm with some related algorithms. Finally, we present the

conclusion of this chapter.

### 4.1 Basic Concepts and Definitions

In this section, we discuss service architecture and its web service implementation. We then, define some important definitions which are essential for understanding static

slicing of service interface diagram discussed subsequently.

#### 4.1.1 Service Architecture and Web Services

![Diagram of Service Architecture](https://via.placeholder.com/150)

**FIGURE 4.1: Class diagram of service architecture**

The purpose of service architecture is to specify the service-oriented architecture (SOA) of

some organization, community, component or process to provide mutual values [175]. As
shown in the Fig. 4.1, the class diagram of service architecture is composed of service interface diagram, service choreography diagram, service participant diagram and service contract diagram. The service interface diagram describes service and participant’s interactions. The service interface diagram is a kind of simple or service interface based on discriminator communication protocol. The simple interface focuses on one-way communication that does not require the caller identity while the service interface requires two-way communication among the participants. The service interface may also describe the choreography of the service which specifies information exchange and its sequence among the participants. In short, the service choreography diagram is a kind of UML sequence diagram representing certain scenarios among the participants. The service participant diagram models the participants that play various roles in service completion. Service participants may be people, organizations, software or individuals that provide and/or consume service(s). The service contract diagram defines the agreement among the participants specifying how a service is to be provided and consumed. The participants must obey the agreement while the service is being invoked. The majority of our work is focused on service interface diagram.

From the implementation perspective of SOA, web services are the only available platform to realize SOA. A generic class diagram of web service has been depicted in Fig. 4.2. A generic web service includes three basic classes service registry, service consumer and service provider. The service provider designs WSDL document and publishes in the service registry using publish() method. The Service consumer looks up the service registry for possible match against business needs using discover() method and finally calls the matched web service using request() method and accordingly, the service provider responds with response() method.

![Class diagram of web service](image)

**FIGURE 4.2:** Class diagram of web service
4.1.2 Some Important Definitions

A slice can be computed by finding different participants and the dependencies among them from service interface model. These participants can be identified based on a slicing criterion.

Definition 4.1: Slicing Criterion
For a service interface dependency graph (SIDG) $G$ of a service-oriented software having a service interface (SI) node $SI(n)$ with an edge symbolized with method $M()$ from SI node $SI(n)$ in $G$ during the execution of a certain business scenario $S$, a slicing criterion is a triplet $[SI(n),M(),S]$. The slicing criterion represented by $[SI(n),M(),S]$ is said to involve the participant of $SI(n)$, and a method $M()$ using composite and service call dependence edges. It should be noted that $M()$ and $S$ is ignored when static slices are computed, and the slicing criterion becomes $[SI(n)]$, only.

Definition 4.2: Service Interface Model Slice
A service interface model slice is a part of service-oriented software consisting of the set of all participants with their properties affecting directly or indirectly in method call $M()$ based on slicing criterion $[SI(n),M(),S]$.

Definition 4.3: Static Slices of a Service Interface Model
Let a static slice of service interface dependency graph (SIDG) $G$ with respect to slicing criterion $[SI(n)]$ be $SSSIM(SI(n))$. Let $((SI(n_1),SI(n_2)),\ldots,(SI(n_1),SI(k)))$ be the set of all dependence edges that can be traversed from $SI(n)$ in the SIDG $G$. Then the static slice of the service interface model can be expressed as

$$SSSIM(SI(n))=((SI(n_1),SI(n_2))\cup\ldots\cup(SI(n_1),SI(k))\cupSSSIM(SI(n_2))\cup\ldots\cupSSSIM(SI(k)))$$

4.2 Service Interface Dependency Graph (SIDG): Our Proposed Intermediate Representation of Service Interface Model

The SIDG represents an intermediate form of service interfaces and uses information available from service interface model. It consists of interface nodes and different types of edges like composite dependence and service call dependence edges. The composite dependence
edge represents the existence of services within the organization. The service call dependence edge between consumer and provider represents service invocation from consumer to provider.

4.2.1 Properties of Service Interface Dependency Graph (SIDG)

A class diagram of service interface dependency graph (SIDG) shown in Fig. 4.3 indicates various elements in structuring SIDG. Each object of SIDG is composed of one or more nodes along with one or more edges representing various types of dependencies as shown in Fig. 4.3. A node in the class diagram represents participants and an edge represents their association while participants play certain roles. As participants play various roles, we cannot differentiate the node types. A composite edge exists between a provider and the organization which owns services. A service call edge exists between consumer and provider. A service call is being followed by service response.

![Class diagram of service interface dependency graph (SIDG)](image)

**FIGURE 4.3:** Class diagram of service interface dependency graph (SIDG)

4.3 Computing Static Slices of Service-Oriented Software from SIDG

In this section, first we describe our proposed algorithm for computing static slices of service-oriented software (SOS). Then we explain the working of our algorithm through an example. Below we discuss an example: TLS’s purchase order service (POS) [219] to explain our proposed approach.
4.3.1 The SSSIM Algorithm

Now, we present a novel static slicing algorithm for service interface model. We named our algorithm static slicing of service interface model (SSSIM). The SSSIM algorithm takes the service interface model and slicing criterion as input and produces static slices. The SSSIM algorithm works in two Phases, Phase 1 construct the SIDG and Phase 2 traverse the SIDG to compute and display the static slices.

In Phase 1, the SIDG is constructed from the service interface diagram. The SIDG construction involves creation, initialization of service interface nodes and representing service calls or drawing composite dependence edges among the nodes. In second Phase, the constructed SIDG is traversed for a given slicing criterion. The SIDG traversal identifies different service interface nodes forming the slices. The resulted slices are being displayed in the SIDG itself which helps SSSIM algorithm to save storage space as the resulted slices are not stored anywhere. The SSSIM algorithm assumes service interface diagram to be in XML metadata interchange (XMI) format.

Algorithm : SSSIM (Static Slicing of Service Interface Model) Algorithm.

Input : Service Interface Diagram in XMI format, and a Slicing Criterion

Output : Static Slices

**Phase 1 : Construction of SIDG from service interface diagram**
1. For each node of service interface diagram do the followings:
   (a) Create a node $Sl(n)$.
   (b) Initialize the node.
2. Add service call and composite dependence edges
   for each node $Sl(i)$, do followings:
   for each node $Sl(j)$, do followings:
   (a) Add service call dependence edge $(Sl(i), Sl(j))$, if node $Sl(i)$ calls service of node $Sl(j)$.
   (b) Add composite dependence edge $(Sl(i), Sl(j))$, if node $Sl(i)$’s functionality is composed by node $Sl(j)$.

**Phase 2: Computation of static slices**
For each node $Sl(n)$ with slicing criterion $[Sl(n)]$, traverse and do the followings:
(a) Mark the service call dependence edge $(Sl(i), Sl(j))$, if node $Sl(i)$ calls service of node $Sl(j)$.
(b) Mark composite dependence edge \((SI(i), SI(j))\), if node \(SI(i)\)'s functionality is composed by node \(SI(j)\).
(c) Remove unmarked nodes from SIDG.
(d) Display the marked nodes as sliced nodes.

Given the slicing criterion as input, the SSSIM algorithm computes the static slices by executing Phase 2 of the algorithm.

### 4.3.2 The SIDG Example

The service interface model for TLS’s purchase order service (POS) [219] is shown in Fig. 4.4. TLS’s purchase order service submits electronic purchase orders (POs) that are received by RailCo’s order fulfillment service (OFS). It defines TLS as a consumer and RailCo as a provider. The TLS’s POS uses the POSOrder() operation of RailCo’s OFS and RailCo’s OFS uses the OFSConfirmed() operation of TLS’s POS. Corresponding to Fig. 4.4, an intermediate representation SIDG has been shown in Fig. 4.5. Since service interface may play various roles depending on business scenarios, we cannot distinguish the node types. Also, it shows three SI nodes, namely consumer, provider, and purchase order service. It defines service call dependence edge between consumer and provider and composite dependence edge between provider and purchase order service.

![FIGURE 4.4: Service interface diagram of POS](image-url)
4.3.3 Complexity Analysis of SSSIM Algorithm

In this section, we analyze the time and space complexity of our proposed SSSIM algorithm. The space complexity SSSIM is $O(n^2)$, where $n$ is the number of service interface nodes. Phase 1 takes $O(n)$ memory space to construct SIDG from XMI and Phase 2 takes $O(n^2)$ space to traverse, accumulate and to display slices. So, the computed space requirement is $O(n^2)$. The time complexity of SSSIM is $O(n)$, where $n$ is the number of service interface nodes. Phase 1 and Phase 2, each requires $O(n)$ time. Hence, the time complexity of our SSSIM algorithm is $O(n)$.

4.3.4 Working of SSSIM Algorithm

We will explain the working of our SSSIM algorithm using the example of SIDG given in Fig. 4.5 which will be obtained at the end of Phase 1 of the algorithm. Let us compute static slices for the slicing criterion Consumer. We have shown the SIDG obtained after the Phase 2 of SSSIM in Fig. 4.6(a) displaying the service interface nodes forming the slices. As the consumer, provider and purchase order service are being traversed from service call dependence edge and composite dependence edges they form the static slices. Similarly, for the slicing criterion Provider, the composite dependence edge is being traversed and the sliced nodes are shown in Fig. 4.6(b).
4.3.5 Correctness of SSSIM Algorithm

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

**Theorem 4.1** SSSIM algorithm always finds a correct dynamic slice with respect to a given slicing criterion.

**Proof.** The proof is given through mathematical induction. Let $S$ be a service-oriented software for which a static slice is to be computed using SSSIM algorithm. For any given set of input values to $S$, the static slice with respect to the first node $SI(n)$ is certainly correct, according to the definition. From this, we can argue that, the static slice with respect to the second node $SI(m)$ is also correct. During static analysis of the service-oriented software $S$, assume that the algorithm has computed correct static slices prior to the node $SI(u)$. To complete the proof, we need only to show that the static slice computed after node $SI(u)$ is correct. Note that the nodes that affect nodes $SI(u)$ must have been affected prior to the node $SI(u)$. It is obvious that the static slice $SSSIM(SI(n))$ contains all those nodes which have affected the node $SI(u)$, since our SSSIM algorithm has marked all the incoming edges to $SI(u)$ only from those nodes on which node $u$ is dependent. The Steps (a), and (b) of Phase 2 of the SSSIM algorithm ensure that the node $SI(u)$ is dependent (with respect to its slicing criterion) on a node $SI(v)$ if and only if the edge $(u, v)$ is marked in the SIDG of the program $S$. If a node does not affect node $SI(v)$, then it will not be included in the static slice $SSSIM(SI(n))$. So, $SSSIM(SI(n))$ is a correct static slice. In other words, we can say that the static slices computed for the node $SI(u)$ are correct. Therefore, the Steps (a), (b), and (c) of Phase 2 of the SSSIM algorithm ensure that the static slices computed for the node $SI(u)$ are correct. This establishes the correctness of the algorithm.
4.4 Implementation and Experimental Results

In this section, we present an implementation of our concept. We have named our slicing tool \textit{SOS Static Slicing} (SOSSS). The design of our tool SOSSS has been shown in Fig. 4.7. It takes SoaML service interface diagram in XMI form as input, which is being parsed by extensible stylesheet language transformations (XSLT) [251]. The XSLT extracts and gathers information related to nodes and generates SIDG. The graphical user interface (GUI) accepts slicing criterion as input, which is being used to traverse and compute static slices. We have used visual paradigm professional edition [230] for modeling and exporting SoaML service interface diagram into XML metadata interchange (XMI). We have used Java script to input the slicing criterion. Java applet and JGraph [109] are used to display SIDG. According to Kundu et al. [52], it is difficult to construct control flow graph from XMI representation. So, we have used XSLT to parse XMI and to traverse the SIDG. Static SIDG will be generated when the user clicks on Static Graph button. Fig. 4.8 shows the resultant slices, with respect to the slicing criterion Consumer which is supplied as input. Similarly, an alternative slicing criterion Provider is given as input and static slices are being computed.

![FIGURE 4.7: Design of our tool](image)

4.4.1 Experimental Results

We have tested the working of SSSIM algorithm using the case study examples as stated in [194] with a service call and composite dependencies using SIDG.

The system configuration used to run SSSIM 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 SSSIM algorithm for these case studies and for several runs. All the measured times reported in this section are overall times, including parsing and building of SIDG representation. Table 4.1 summarizes the \textit{average run-time and memory space requirements} of SSSIM algorithm. From Table 4.1 the following observations can be made:
1. The SIDG (Service Interface Diagram) is a connected and acyclic graph which is used as an intermediate representation for computing the slices.

2. The SIDG is a connected, acyclic and \( n = e + 1 \).

3. Every two service interface nodes of SIDG are joined by a unique path.

As we are not aware of the existence of any algorithm for static 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 static slices at different service interface nodes of a service interface diagram, we have calculated the average run-time requirements of the SSSIM algorithm.

From the experimental results, it can be observed that the average run-time increases sublinearly as the number of service interface nodes increases in a service interface diagram as shown in Fig. 4.9. Average run-time increases linearly as the number of service interface nodes in SIDG increase. In Fig. 4.9, for 13 nodes the average run-time is 0.60 Seconds. The experiments have been performed for graphs having maximum number of nodes 13. SSSIM algorithm is scalable to handle the real situation where large numbers of nodes arise. But, the time complexity and space complexity will increase significantly. The performance results of our implementation agree with the theoretical analysis.
FIGURE 4.9: Average run-time of SSSIM algorithm w.r.t number of nodes

TABLE 4.1: Average run-time and memory space requirements of SSSIM algorithm

<table>
<thead>
<tr>
<th>Sl. No.</th>
<th>Benchmark Model</th>
<th># Nodes in SIDG</th>
<th># Edges in SIDG</th>
<th>Slicing Criterion ((SI(n)))</th>
<th>Avg. Run-Time (in Sec)</th>
<th>Memory Space Reqs. (in KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Online Shopping System (OSS)</td>
<td>13</td>
<td>12</td>
<td>Consumer</td>
<td>0.60</td>
<td>6.9</td>
</tr>
<tr>
<td>2</td>
<td>Hotel Automation Software (HAS)</td>
<td>9</td>
<td>8</td>
<td>Consumer</td>
<td>0.50</td>
<td>6.8</td>
</tr>
<tr>
<td>3</td>
<td>Medicine Shop Automation Software (MSAS)</td>
<td>9</td>
<td>8</td>
<td>Provider</td>
<td>0.50</td>
<td>6.9</td>
</tr>
<tr>
<td>4</td>
<td>Bookshop Automation Software (BAS)</td>
<td>9</td>
<td>8</td>
<td>Consumer</td>
<td>0.50</td>
<td>6.8</td>
</tr>
<tr>
<td>5</td>
<td>Road Repair and Tracking System (RRTS)</td>
<td>8</td>
<td>7</td>
<td>Provider</td>
<td>0.49</td>
<td>6.8</td>
</tr>
<tr>
<td>6</td>
<td>Restaurant Automation System (RAS)</td>
<td>8</td>
<td>7</td>
<td>Consumer</td>
<td>0.50</td>
<td>6.9</td>
</tr>
<tr>
<td>7</td>
<td>Student’s Auditorium Management Software (SAMS)</td>
<td>8</td>
<td>7</td>
<td>Provider</td>
<td>0.49</td>
<td>6.8</td>
</tr>
<tr>
<td>8</td>
<td>Library Information System (LIS)</td>
<td>8</td>
<td>7</td>
<td>Consumer</td>
<td>0.50</td>
<td>6.8</td>
</tr>
<tr>
<td>9</td>
<td>Software Component Cataloguing Software (SCCS)</td>
<td>7</td>
<td>6</td>
<td>Consumer</td>
<td>0.40</td>
<td>6.8</td>
</tr>
<tr>
<td>10</td>
<td>Supermarket Automation Software (SAS)</td>
<td>7</td>
<td>6</td>
<td>Consumer</td>
<td>0.40</td>
<td>6.9</td>
</tr>
<tr>
<td>11</td>
<td>Judiciary Information System (JIS)</td>
<td>6</td>
<td>5</td>
<td>Consumer</td>
<td>0.20</td>
<td>6.5</td>
</tr>
<tr>
<td>12</td>
<td>Municipality Garbage Collection Automation Software (MGCAS) [Appendix-A]</td>
<td>5</td>
<td>4</td>
<td>Consumer</td>
<td>0.10</td>
<td>6.5</td>
</tr>
<tr>
<td>13</td>
<td>Motor Parts Shop Software (MPSS)</td>
<td>4</td>
<td>3</td>
<td>Consumer</td>
<td>0.10</td>
<td>6.3</td>
</tr>
<tr>
<td>14</td>
<td>Railway Reservation Software (RRS) [Appendix-A]</td>
<td>4</td>
<td>3</td>
<td>Provider</td>
<td>0.09</td>
<td>6.3</td>
</tr>
<tr>
<td>15</td>
<td>House Rental Software (HRS) [Appendix-A]</td>
<td>4</td>
<td>3</td>
<td>Provider</td>
<td>0.09</td>
<td>6.3</td>
</tr>
</tbody>
</table>
4.5 Comparison with Related Work

To the best of our knowledge, no work has been done on static slicing of SOA-based software. The only comparable work to our work is the work of Lallchandani et al. [99, 110]. With respect to asymptotic analysis both perform equally. Lallchandani et al. [99, 110] have merged class diagram and sequence diagram in order to get better information of the system, while we used SoaML service interface diagram. From the implementation perspective, Lallchandani et al. [99, 110] used document object model (DOM) parser, while we used XSLT parser.

4.6 Conclusion

In this chapter, we have proposed a novel technique for static slicing of SOA-based software using SoaML service interface diagram. Such slicing based on SoaML service interface diagram is difficult due to distributed participants with implicit dependencies among them in SOA-based software. We first construct SIDG, an intermediate representation for service interfaces. Then, our SSSIM algorithm uses SIDG information to compute static slices. Such static slices can be used for understanding change impact analysis, and SOA comprehension etc.

The algorithm in this chapter is not suitable to be applied to SOA-based web services running on several nodes connected through a network. Even, such static slices are small at higher-level of abstractions as in case of service interface diagram, but at lower-level of implementation, it may contain large number of web service statements affected by the values of variables. In the next chapter, we will extend our framework to compute dynamic slices of service-oriented software in which services run on several nodes, as is common in Client-Server applications.