ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)

Vol 5, Issue 3, March 2018

Dynamic Slicing of Service-Oriented Software

[1] Kaushik Rana, [2]Jalpa Ramavat, [3]Durga Prasad Mohapatra
[1][2] Computer Engineering Department, Vishwakarma Government Engineering College, Gujarat, India.
[3] Department of Computer Science and Engineering, NIT Rourkela, Orissa, India.

***Abstract* - SoaML (Service oriented architecture Modeling Language) diagrams are the basic modeling artifacts for Service-Oriented Architecture (SOA). These SoaML models can also be used for testing Service-Oriented Software (SOS). Testing can be planned at design phase of software development life cycle. With this context, we present a novel technique to compute 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 as Service-Oriented Software Dependence Graph (SOSDG) which is an intermediate representation that needs to be stored and traversed to get dynamic slice as and when web service gets 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 computation of global dynamic slice based on SOSDG, it's dependencies induced within or across organizations and small slices.**

Keywords: Progam Slicing, Service-Oriented Architecture(SOA), Testing, Web Service.

I. INTRODUCTION

Nowadays many organizations are shifting their focus from technology oriented business process to Service-Oriented Architecture (SOA) based business process. This paradigm shift occurs due to major advantages offered by SOA over the traditional manual business process. SOA enables organizations to align their businees process with changing customer needs,creating flexible, agile business environments. Understanding such dynamic SOA enables one to have great insight and comprehension of a business process. But it is often a difficult task. Design models helps organizations to simplify, visualize business processes. UML has been widely used a general purpose modeling language for object-oriented systems which is not intended for modeling distributed systems like SOA. SoaML (Service oriented architecture Modelling Language), a language which models SOA, specified by OMG and being supported by major IT vendors like IBM, Visual Paradigm , open source organization Eclipse etc, is an emerging standards for modeling distributed systems.

The rest of this paper is organized as follows. The next section presents related work. Section 3 introduces a service-oriented software choreography example of buying a product from online seller and its mapping with web services. Section 4 presents basic definitions relevant to our proposed algorithm and extentions to dynamic slicing technique. Our intermediate representation SOSDG is discussed in Section 5. Section 6 presents our algorithm MBGDS for computing global dynamic slice of service-oriented software, it's working and theoretical complexity analysis. Section 7 presents proof of concept covering our tool SOSDS's design, implementation, data sets and results obtained. Section 8 compare our work with related work. And finally, Section 9 concludes the work.

II. RELATED WORK

In this section we, briefly presents the reported work on program slicing, architectural and UML model based slicing. Most of the work reported in the literature are focused on development of technique for slicing UML models like class diagram, sequence diagram, use case diagram, activity diagram etc.

A natural way of localizing an error is to consider only those statements of a program, which tends to the erroneous behaviour being observed. Often this results in find-ing the statements of a program relevant to the value of a chosen variable at a given location of that program. This approach is called program slicing. The given variable and location form the slicing criterion. The original concept of program slicing was proposed by Mark Weiser [22] as another approach for debugging sequential programs. He claims that program slicing corresponds to the mental ab-straction performed by programmers while debugging pro-grams. Agrawal et al. [9] have presented a uniform ap-proach to compute dynamic slices of programs that may in-volve unconstrained pointers, composite variables, and pro-cedures. Program slicing is an active area of research, and this is reflected in various surveys as Frank Tip [8].

Zhao [14] was the first one to introduce the concept of architectural slicing based on Architectural Description Language (ADL) ACME. He define component-connector dependency, connector component dependency, and addi-tional dependency. He proposed an algorithm to com-pute architectural slice based on SADG (Software Ar-chitectural Dependency Graph). He extended his previ-ous work [13] 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. [36] have introduced Dynamic Software Architecture Slicing (DSAS) as set of architect components and connectors that are relevent to the particular variable and events of interest at some point dur-ing the execution of software architecture. Korel et al.[3] have presented deterministic and nondeterministic slicing based on EFSMs (Extended Finite State Machines). They also develop a tool to demonstrate their slicing technique. Kagdi [10] have introduces concept of model slice, which extracts slice from a class diagram.

Samuel et al. [26] have presented a technique to test object-oriented software using dynamic slice technique on UML sequence diagram. They used message guards on se-quence diagram to generate dynamic slice with respect to each conditional predicates. Slices are generated from the dependencies graph for all the variables at each message point in sequence diagram. Their approach suggest that they computed static slice.

Samuel et al. [27] have generated test case from UML communication diagram. They generated communication tree from communication diagram (CD), performs post-order traversal on conditional predicates and apply function minimization technique to generate test data.They imple-mented a tool named UTG (UML behavioral Test case Gen-erator) to demonstrate generation of test cases.

Soldal et al.[23] defines a test generation scheme based on a conformance testing and a formal operational semantics for sequence diagrams, which takes input as sequence diagrams that may contain the operators assert and neg and that produces tests in the form of sequence diagrams. The algorithm is based on a formal operational semantics for se-quence diagrams and is an adaption of J. Tretmans et al.[11]. The operational semantics and the test generation al-gorithm are implemented in term rewriting language Maude defined by M. Clavel [19].

Lallchandani et al. [12, 17] have used generic class diagram and generic sequence diagram and integrated them to generate Model Dependency Graph (MDG). They proposed an algorithm AMSMT (Architectural Model Slicing through MDG Traversal) to produce the static and dynamic architectural model slices. The algorithm traverses the edges of Model Dependency Graph (MDG) according to slicing criterion. They developed a tool which computes a dynamic slice from UML architectural models.

NODA et al. [16] have extended their own work on dynamic slicing of sequence diagram. They generated Behavior Model (B-model), defines various dependencies and calculates slice. They support their work by incorporat-ing exceptions and multi-threading programs. They imple-mented a tool as Eclipse plug-in to demonstrate their pro-posed method.

Prasanna et al. [20] have presented a model based approach for automated generation of test cases in object-oriented systems. They consider UML object diagram and genetic algorithms tree crossover operator to generate new generation tree. The new generation of trees are converted into binary trees. Then, depth first search traversal is per-formed on binary trees to generte test cases.

Sharma et al. [24] have presented Use case Diagram Graph (UDG) and Sequence Diagram Graph (SDG) for generating test cases from use case and sequence diagrams respectively. They integrated UGD and SDG into Sys-tem Testing Graph (STG). The STG is traversed to get test cases.They uses a PIN authentication scenario of an ATM system. Also they implemented a tool with the help of Mag-icDraw and Rational Rose.

Kundu et al. [5] have presented an approach for generating test cases from UML 2.0 activity diagrams with use case scope. They define an activity path coverage criterion. Ad-ditionally, the generated test cases are capable of detecting various types of faults which may occur.

Swain et al. [31] have illustrated a method to derive test cases from analysis artifacts such as use cases, and their cor-responding sequence diagrams. They generated test cases from Use Case Dependency Graph (UDG) and Concurrent Control Flow Graph (CCFG) from use case diagram and corresponding sequence diagram respectively. They gen-erated test case using full predicate coverage criteria. They used Library Information System (LIS) to demonstrate their work.

Nayak et al. [2] have proposed an approach of synthesizing test data from information available from class diagrams, sequence diagrams and Object Constraint Language (OCL) constraints. They generated a Structured Composite Graph (SCG), incorporating system wide information from which test case are generated

Shanthi et al. [1] have focused on test case genera-tion from UML sequence diagram using genetic algorithm. They extract information from sequence diagram and creates a Sequence Dependency Table (SDT). With the help of SDT test path are generated. And then Genetic Algorithm are used to prioritize test cases.

Kosindrdecha [25] have proposed a technique to derive and generate tests from state chart diagram. They car-ried out an extensive literature survey and classified vari-ous test generation technique as specification-based techniques, sketch diagram-based techniques, and source code-based techniques.

Grigorjevs et al. [15] presented a testing technique for model generation from UML sequence diagram to UML Testing profile. They discusses principles of model transformation to generate test cases from sequence diagram and shows practical approach to specific model.

Patnaik et al. [4] have put an effort to represent dead-lock situations with the help of graph in a distributed environment. They proposed an algorithm to detect deadlock with the help of real time banking system and generated test cases for it.

Panthi et al. [39, 38] have generated test cases from sequence diagram automatically. They named their approach as Automatically Test Sequences Generation from Sequence Diagram (ATGSD). In ATGSD, they first convert the sequence diagram into Sequence graph then, the graph is traversed to select the predicate functions. Next they transform the predicate into source code. Then, they construct the Extended Finite State Machine (EFSM) from the code. Finally, they generate the test data corresponding to the transformed predicate functions and store the gener-ated test data for future use. They demonstrate their work based on bank ATM system and tool based on ModelJunit library.

Sumalatha et al. [37] have presented a new technique to generate test case from integrating UML activity and sequence diagram. They uses breadth first search on activity sequence graph which is generated from merging activity and sequence diagram. Priya et al. [30] have generated test path from UML sequence diagram of a medical consulta-tion system. Swain et al. [28] uses condition slicing and generate test cases from UML interaction diagrams. They generated test cases from message flow dependence graph from UML sequence diagram and then applies conditioned slicing on a predicate node of the graph.

Meena [6] have proposed an approach to generate test cases from UML sequence diagram and interaction overview diagram. They transforms the sequence diagram and interaction overview diagram to an intermediate form called UML interaction graph (UIG) using XMI code. Then, UIG is traversed to find the all valid path and generate test cases. Kaur et al. [21] have developed a methodology to derive test cases using conditional predicate.

III. SERVICE-ORIENTED SOFTWARE EXAMPLE: ONLINE SHOPPING SYSTEM

Let’s take a real world example to demonstrate service-oriented software choreography. Buying a product from the online retailer is a good example of service-oriented soft-ware choreography. Generally, a product seller registers its product with online retailer. Other service providers may also register for shipping and logistic service with retailer too. These registration service interface is being provided by online retailer. When a customer wants to buy a product, he(she) searches it over the online retailer. Thus, searching a product by customer represents a web service. Once the product is found the customer goes for buying it. The buy-ing service is a composite service which checks for user login. Again this login service is being provided by re-tailer for their registered users or users may sign up or else uses the third-party login service like www.facebook.com or www.gmail.com. Once the customer successfully logged on, he(she) can go for make payment or adding the product to shopping cart for later payment or uses the Cash On Delivery (COD) service options. The payment service interface is being provided by various banks. 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 chore-ography is best described using SoaML sequence diagram shown as in Fig. 1.

In this service choreography, we can see that the *prod-uct 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.

The successful registration of sellers or courier companies depends on Service Level Aggrement (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. These service contracts can be best modeled as SoaML service contract diagram as shown in Fig. 2.

III. A STATIC 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 service. Fig. 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. This mapping is useful to run and analyze the web service with corresponding SOAP messages.

IV. EXTENTIONS TO DYNAMIC SLICING TECHNIQUE

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

Definition 1: def(m)
let m be a message node in Service-Oriented Software Choreography (SOSC). A node n is said to be def(m) node if n it defines(assigns) values to variables of message m.

Definition 2: use(m)
let m be a message node in SOSC. A node n is said to be use(m) node if n uses the values of variables assigned by message m.

Definition 3: recentDef(m)
For each message m, recentDef(m) represents the node corresponding to the most recent definition of the message variable def(m) in particular Service Execution History (SEH).

Definition 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, Start, Stop), where each node n 2 N represents message of service choreography SOSC, while each edge e 2 E represents control transfer among nodes. Nodes Start and Stop are two unique nodes representing entry and exit of the program P, 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 a service-oriented software given in Fig. 1 is shown in Fig. 4.

Definition 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

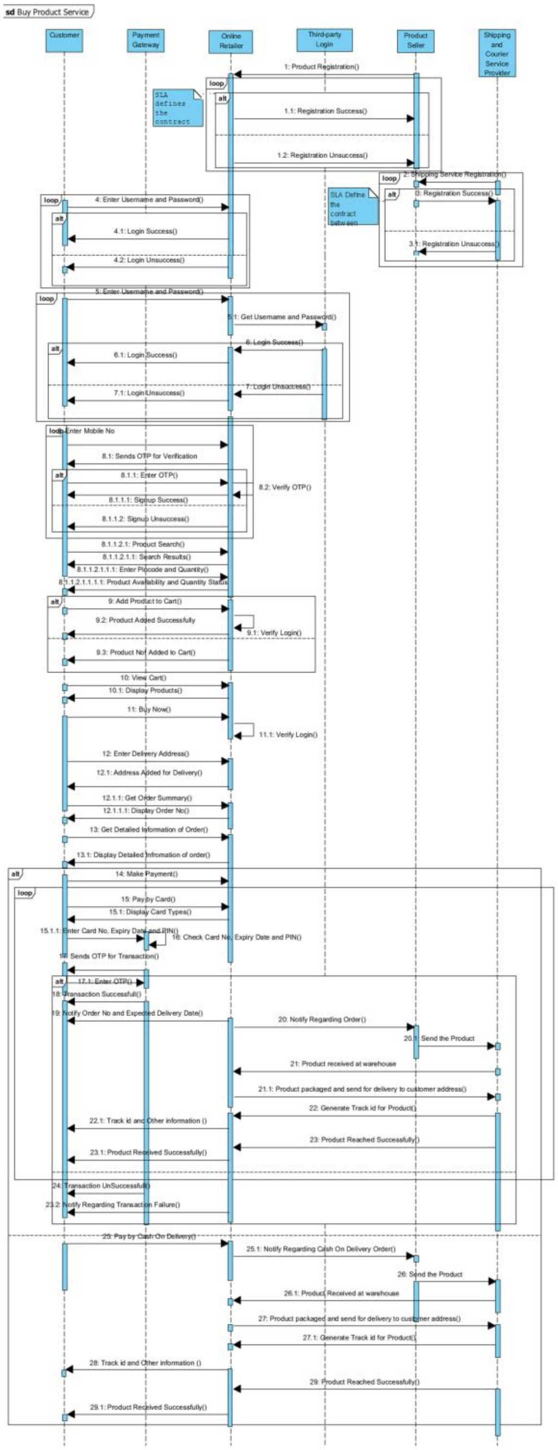

sd Buy Product Service
Customer
Payment Gateway
Online Retailer
Third-party Login
Product Seller
Shipping and Courier Service Provider
1: Product Registration()
loop
SLA defines the contract
alt
1.1: Registration Success()
1.2: Registration Unsuccess()
2: Shipping Service Registration()
loop
alt
3: Registration Success()
SLA Define the contract between
3.1: Registration Unsuccess()
loop
4: Enter Username and Password()
alt
4.1: Login Success()
4.2: Login Unsuccess()
5: Enter Username and Password()
loop
5.1: Get Username and Password()
6: Login Success()
alt
6.1: Login Success()
7: Login Unsuccess()
7.1: Login Unsuccess()
loop Enter Mobile No
8.1: Sends OTP for Verification
alt
8.1.1: Enter OTP()
8.2: Verify OTP()
8.1.1.1: Signup Success()
8.1.1.2: Signup Unsuccess()
8.1.1.2.1: Product Search()
8.1.1.2.1.1: Search Results()
8.1.1.2.1.1.1: Enter Pincode and Quantity()
8.1.1.2.1.1.1.1: Product Availability and Quantity Status
alt
9: Add Product to Cart()
9.2: Product Added Successfully
9.1: Verify Login()
9.3: Product Not Added to Cart()
10: View Cart()
10.1: Display Products()
11: Buy Now()
11.1: Verify Login()
12: Enter Delivery Address()
12.1: Address Added for Delivery()
12.1.1: Get Order Summary()
12.1.1.1: Display Order No()
13: Get Detailed Information of Order()
13.1: Display Detailed Infromation of order()
alt
14: Make Payment()
loop
15: Pay by Card()
15.1: Display Card Types()
15.1.1: Enter Card No, Expiry Date and PIN()
16: Check Card No, Expiry Date and PIN()
17: Sends OTP for Transaction()
alt
17.1: Enter OTP()
18: Transaction Successfull()
19: Notify Order No and Expected Delivery Date()
20: Notify Regarding Order()
20.1: Send the Product
21: Product received at warehouse
21.1: Product packaged and send for delivery to customer address()
22: Generate Track id for Product()
22.1: Track id and Other information ()
23: Product Reached Successfully()
23.1: Product Received Successfully()
24: Transaction UnSuccessfull()
23.2: Notify Regarding Transaction Failure()
25: Pay by Cash On Delivery()
25.1: Notify Regarding Cash On Delivery Order()
26: Send the Product
26.1: Product Received at warehouse
27: Product packaged and send for delivery to customer address()
27.1: Generate Track id for Product()
28: Track id and Other information ()
29: Product Reached Successfully()
29.1: Product Received Successfully()

Figure 1. An SoaML sequence diagram for buying a product

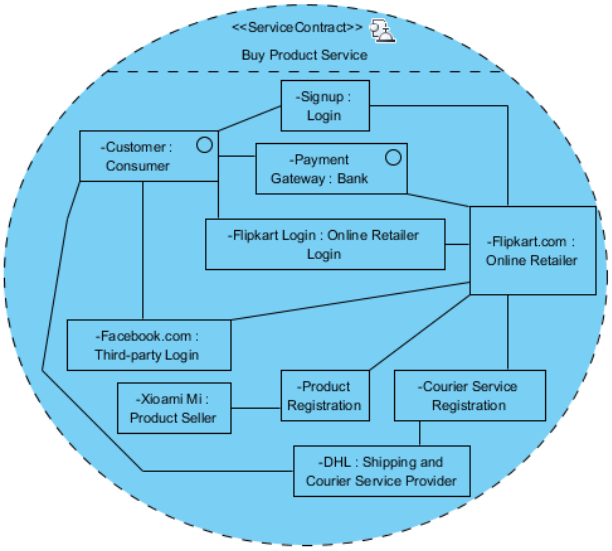

<<ServiceContract>>
Buy Product Service
-Signup : Login
-Customer : Consumer
-Payment Gateway : Bank
-Flipkart Login : Online Retailer Login
-Flipkart.com : Online Retailer
-Facebook.com : Third-party Login
-Xioami Mi : Product Seller
-Product Registration
-Courier Service Registration
-DHL : Shipping and Courier Service Provider

Figure 2. An SoaML service contract diagram involving multiple parties

dynamic binding, agility and many others.

We define Service Execution Case (SEC) as being 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. A Service Execution Case must consider non-determinism.

Definition 6: Service Execution Point

In 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 $(S_k\ , M_i^j)$ where Mij is the jth occurrence of message Mi 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 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 8: Service Choreography Execution His-tory
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 the 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 given service execution case.

Definition 9: Slicing Criterion for Service-Oriented Software Choreography
We define the slicing criterion for the Service-Oriented Soft-ware Choreography (SOSC) as the triplet (var, (S_k , M_i^j), SEC) where var is the message variable used at m, and (S_k , M_i^j) is Service Execution Point (SEP) with input Ser-vice Execution Case (SEC).

IV.I EXTENTIONS TO DEPENDENCY

In this subsection we define some new dependencies like intra- service and inter- service dependencies which arises due to service-oriented software and then we define local dynamic slice and global dynamic slice.

Definition 10: Intra-Service Dependency
In a sequential program the occurrence of the statements on which the current statement is dependent have 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 being an edge denoting that:

1. the two nodes (n_i, n_j) are being executed by two dis-tinct 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 nj by service S_j .

3. These service S_i and S_j are being provided by a single service provider.

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

Definition 11: Inter-Service Dependency

We define inter-service dependence edge (n_i, n_j) as being an edge denoting that:

1. the two nodes (n_i, n_j) are being executed by two dis-tinct services S_i and S_j respectively; and

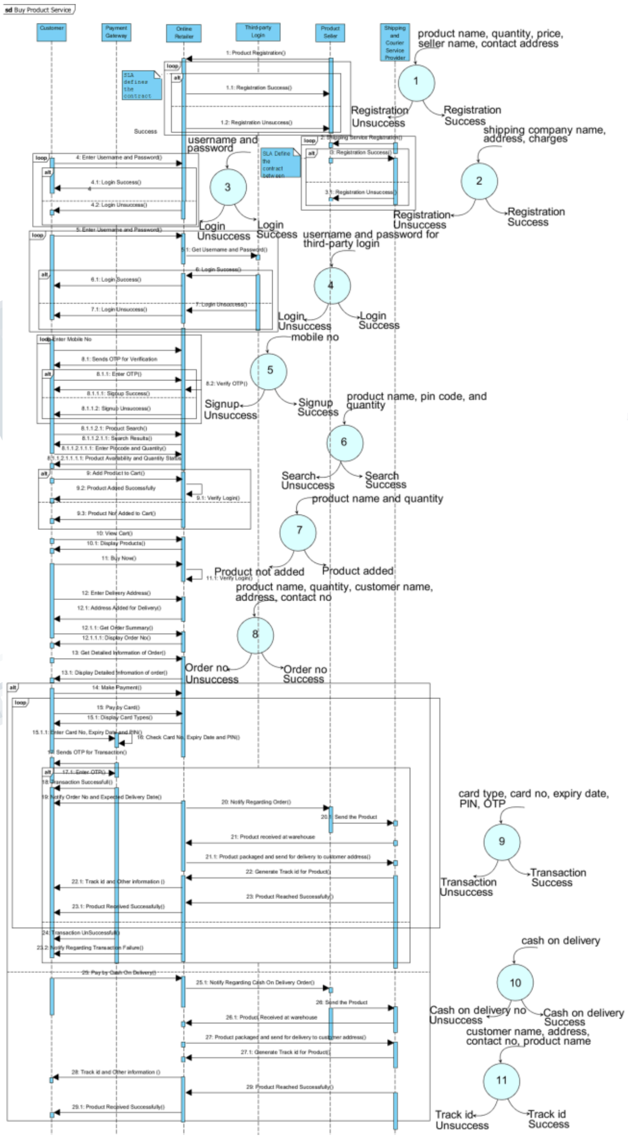


Figure 3. A static mapping of service chore-ography messages

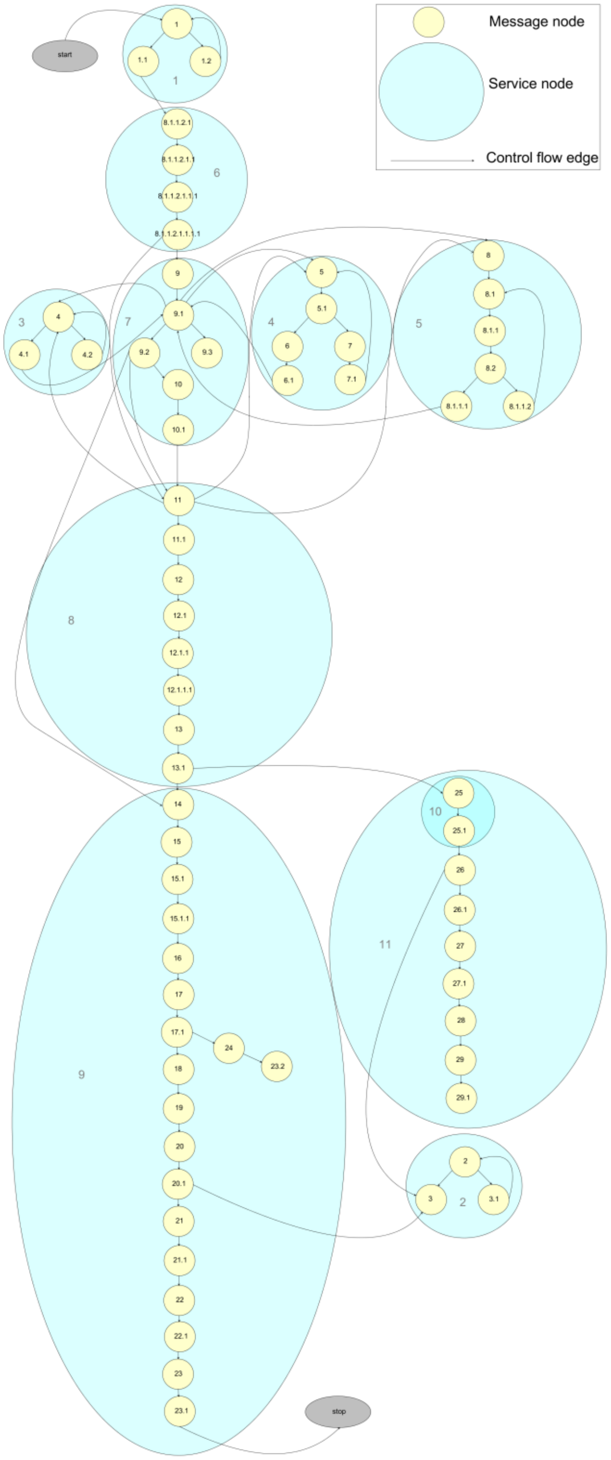


Figure 4. The WSCFG of the service choreography of Fig 1

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 service 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 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 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 (var, $(S_k\ , M_i^j)$, SEC) as the subset of Service Choreography SOSC messages whose execution really affected the value of message variable var, as observed at the service execution point $(S_k\ , M_i^j)$, for the service execution case SEC.

Definition 13: Local Dynamic Slice of Service-Oriented Software Choreography

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

V. 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 name this representation Service-Oriented Software De-pendence Graph (SOSDG). Each message in a sequence di-agram is represented as a node along with their number, in SOSDG. This message node also maps with correspond-ing input or output message of web service. This SOSDG captures control dependencies from static analysis of se-quence diagram. It also captures data, intra-service and inter-service dependencies from run time analysis of cor-responding web service execution. The inter-service depen-dencies may cross organizational boundaries. We also de- pict 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 shows the SOSDG for Fig. 1.

Further, Service-Oriented Software Dependence Graph (SOSDG) can be defined for a Service Choreography Ex-ecution History (SCEH). It is a two-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 two-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 mes-sage occurance within that graph. In addition data, intra-service and inter-service dependence connects 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 His-tory (SCEH) it contains all the web services which have ex-ecuted at least one message in that SCEH. If a web service is contained in that set, its 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.

VI. MARKING BASED GLOBAL DYNAMIC SLICING (MBGDS) ALGORITHM

In this section, first we briefly describe our MBGDS al-gorithm. Then, we present the pseduo-code of the algo-rithm. Subsequently, we discuss the complexity of our al-gorithm.

VI.I OVERVIEW OF THE MBGDS ALGO-RITHM

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

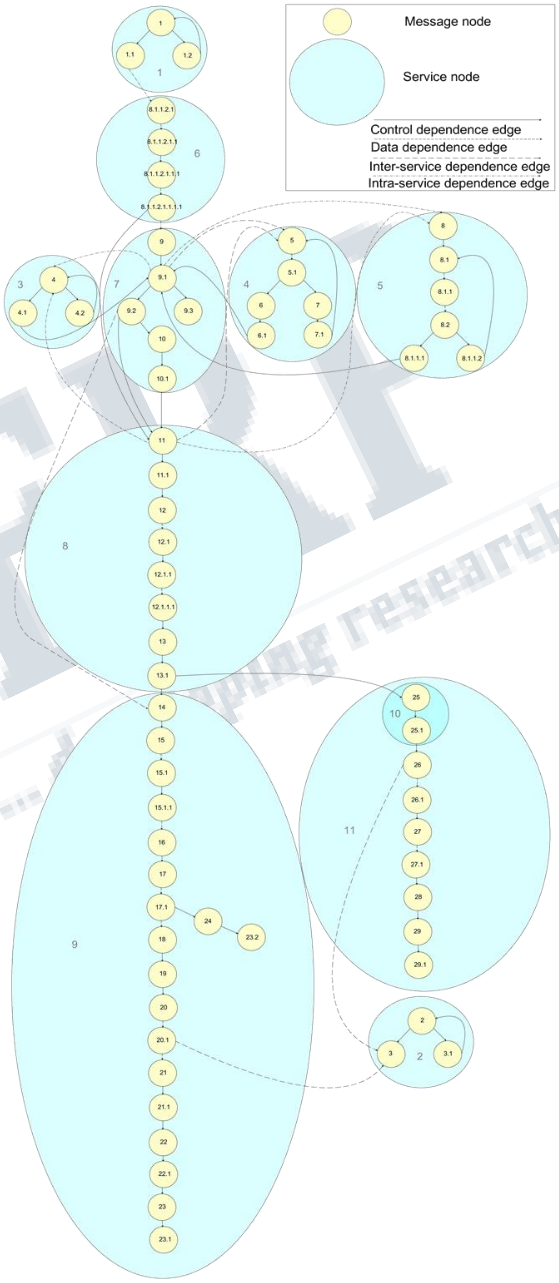


Figure 5. The SOSDG of the service choreog-raphy of Fig. 1

all incoming marked dependence edges excluding the control dependence edges, associated with the service Si, 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)* denote 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),….(z$_k$,m)}*g 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 Si, with input SEC, is given by

Global Dynamic Slice *(s,m) = {z$_1$, z$_{2,}$….. z$_k$}*
Global Dynamic Slice (z$_1$,m)
Global Dynamic Slice (z$_2$,m) ∪ … … … ∪
Global Dynamic Slice (z$_k$,m)

Let *{m$_1$, m$_{2,}$….. 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) ∪
Global Dynamic Slice (m$_2$,m) ∪ … ∪
…*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 dy-namic 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 unmark-ing the different dependencies such as data dependencies, control dependencies, intra-service dependencies and inter-service dependencies as they arise and ceases. The phase 3 is responsible for computing the global dynamic slice for a given slicing criterion using updated SOSDG. It also lookups the global dynamic slice during run-time. So, when a request for a slice is made, it is obtained quickly. 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 in the form of pseudo-code.

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

Input : Service-Oriented Software Choreography (SOSC) and Slicing Criterion (var, $(S_k\,, M_i^j)$, SEC) // Slicing criterion given during run-time

Output : Computed Global Dynamic Slice // Global dynamic slices 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 the following:
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 con-trol 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 fol-lowing
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 Si of the Service-Oriented Software Choreography (SOSC), con-sisting of set *(S_1,S_2,...,S_k).*

(a) Set Global Dynamic Slice*(s,m)* = Ø for every message m used or defined at every node m of the SOSDG.

(b) Set *recentDef(m)* = NULL for each message variables in service S_i .

// end of initialization

2. Runtime Updation of SOSDG: Run the web ser-vices and carry out the following after each mes-sage m of the service corresponding to the SOSC s, and SEC for each Si 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 recentDef(m) of the service S_i in SEH.

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

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

(e) Update the global dynamic slice for different de-pendencies:

i. Handle data dependency: Let

f(d_1 ,m),...,(d_j,m)gbe the set of marked incoming data dependencies to m. Then,
$=\{d_1, d_2, \ldots, d_j\} \cup Global\_Dynamic\_Slice(d_1) \cup Global\_Dynamic\_Slice(d_2) \cup \ldots \cup Global\_Dynamic\_Slice(d_j),$
where *{d_1, $d_{2,}$..... d_k}* are the initial nodes of the corresponding marked incoming edges of m.

ii. Handle control dependency: Let (c, m) bethe marked control dependence edge. Then,
Global Dynamic Slice(m) =
$Global\_Dynamic\_Slice(m) \cup \{c\} \cup Global\_Dynamic\_Slice(c).$

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

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

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,

Global Dynamic Slice(s, m) = $\{d, c, x, y\} \cup Global\_Dynamic\_Slice(d) \cup Global\_Dynamic\_Slice(c) \cup Global\_Dynamic\_Slice(x) \cup Global\_Dynamic\_Slice(y)$

(b) For message variable defined at node m, do

Global Dynamic Slice (s, m) = $Global\_Dynamic\_Slice(m) \cup \{m\}$.

2. Global Dynamic Slice Look Up

(a) If a slicing command (s,m) is given for a service-oriented software choreography SOSC, SEC and for particular message vari-able var carry out the following:

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.

VI.II WORKING OF MBGDS ALGORITHM

We are interested in computing the global dynamic slice of service-oriented software choreography shown in Fig. 1 with respect to the slicing criterion (order no, (S8,M111), fseller name= XY, contact address= China, product name=AB, quantity=1000, price=12999, user name= YZ, password= temp 123456g). The updated SOSDG after applying Phase 2 of our algorithm is shown in Fig. 6. We will explain how our algorithm computes the slice. To the input given in SEC, our algorithm MBGDS will executes the message nodes f1, 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, 11g. The global dynamic slice is computed as below:

Global Dynamic Slice $(S_8, M11^1)$ = Global Dynamic Slice (9:2) {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}

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

$Global\_Dynamic\_Slice(4.1) = \{4, 4.1\} \cup Global\_Dynamic\_Slice(9.1)$ ={ 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}

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

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

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

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

$\mathrm{Global\_Dynamic\_Slice}(8.1.1.2.1) = Global\_Dynamic\_Slice(1.1) \cup Global\_Dynamic\_Slice (8.1.1.2.1) \cup \{1.1\} = \{1, 1.1, 8.1.1.2.1\}$

$Global\_Dynamic\_Slice(1) = Global\_Dynamic\_Slice\{1.2\} \cup Global\_Dynamic\_Slice\{1\} \cup \{\phi\}$ ={ 1, 1.1}

VI.III SALIENT FEATURE 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 handle inter-service communication and intra service communication by using WSDL.
- 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 a service.

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 sequence diagram. It can be easily extended to accommodate dynamic slices of cloud based programs.

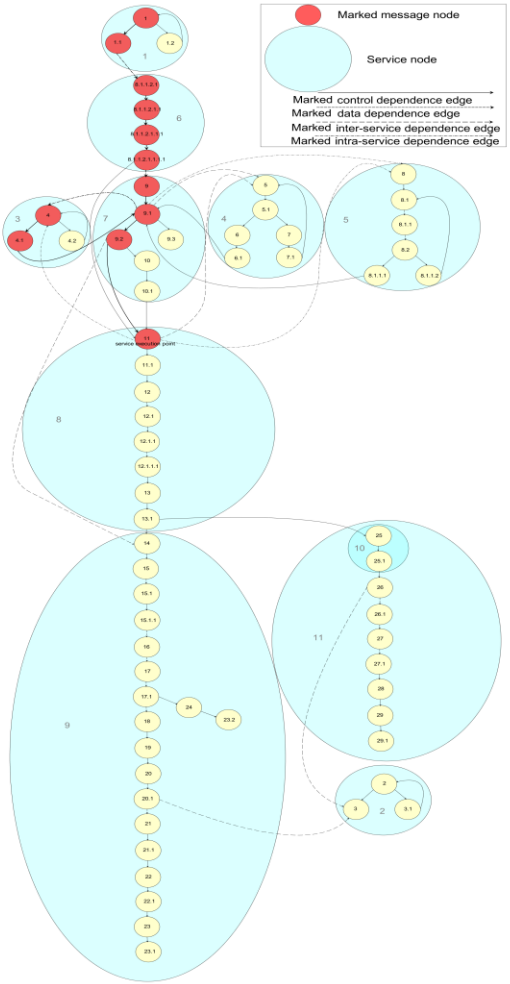

Marked message node
Service node
Marked control dependence edge
Marked data dependence edge
Marked inter-service dependence edge
Marked intra-service dependence edge
service execution point

Figure 6. The updated SOSDG of the service choreography of Fig. 1

VI.IV 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 considers barometer instructions which significantly contributes for the computation of the slice. The first instruction is related with 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(n2) time is required to compute and update SOSDG. And n will be the length of execution of web services involved in SOSC, then the run-time complexity of the MBGDS algorithm would be O(n^3). We consider 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 dynamic slice would be O(*n3*) and O(*n2*) space is required to store the recentDef. So,the space complexity of our MBGDS algorithm would be O(*n3*).

VII. PROOF OF CONCEPT

In this section, we present a tool for SOSDG construc-tion and discuss the experimental results obtained using the tool.

VII.I 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 as 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 communica-tion using WSDL. In the following, we briefly discuss the design, implementation, and working of our slicing tool.

VII.II DESIGN OF SOSDS

The high level design of our implementation SOSDS has been depicted in Fig. 7.

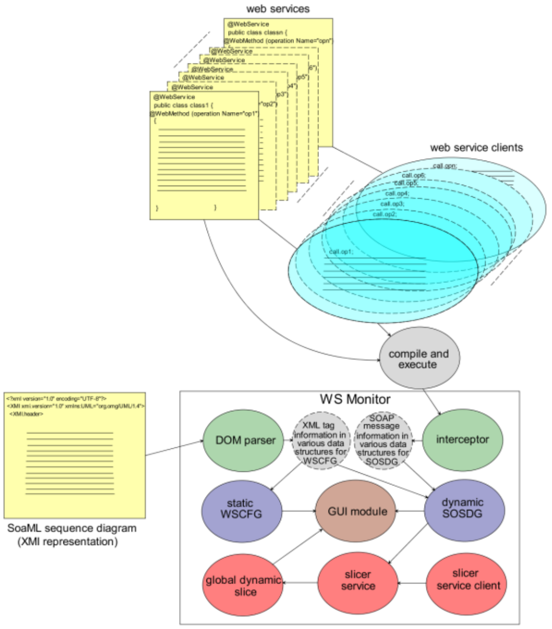

web services
@WebService
public class classn {
@WebMethod (operation Name="opn")
@WebService
public class class1 {
@WebMethod (operation Name="op1")
web service clients
call.opn;
call.op6;
call.op5;
call.op4;
call.op3;
call.op2;
call.op1;
compile and execute
WS Monitor
<?xml version="1.0" encoding="UTF-8"?>
<XMI xmi.version="1.0" xmlns:UML="org.omg/UML/1.4">
<XMI.header>
SoaML sequence diagram (XMI representation)
DOM parser
XML tag information in various data structures for WSCFG
SOAP message information in various data structures for SOSDG
interceptor
static WSCFG
GUI module
dynamic SOSDG
global dynamic slice
slicer service
slicer service client

Figure 7. Schematic design of SOSDS

The SOSDS takes input an SoaML sequence diagram comprising the parties and their interactions, in XMI format. This is parsed by the DOM Parser module, which gathers information regarding parties participating in inter-actions along with the messages exchanged among them. The DOM parser reads the entire input XMI file and cre-ates a tree structure in memory. When the DOM parser encounters an XML tags in the XMI, it parses the tag to de-scribe what type of tags was 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 generated during run-time execution of services and it's clients serves as input to our SOSDS. These inputs helps the interceptor module to intercept SOAP messages exchanged among services during run-time. The intercepted SOAP messages provides 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. These 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 construct WSCFG and SOSDG of the SoaML model in consultant 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.

VII.III 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 [41]. When the web service and their clients are made to run our slicing tool SOSDS, first the interceptor module inter-cepts the SOAP messages and stores the data in a hashmap called message info. Meanwhile the DOM parser module analyze the XML tags and consturct the WSCFG from an SoaML sequence diagram (XMI representation) given as in-put. 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 en-coding, 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 SOSDS provides the global dynamic slice for the given slicing criterion and also visualize SOSDG. The visualization of both the graphs, WSCFG and SOSDG was carried out by using JGrpahT Library [18].

VII.IV DATA SET USED IN 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 analysis and design an SoaML sequence diagram for a system description given in exercises of chapter 6 of a software engineering book [29]. Also instructed to implement each of the system as service-oriented software system and to use JAX-WS API to create web services. Even, we gave flexibility to choose system based on their own choice from any real SOA world examples. The SoaML sequence diagrams were constructed using Visual Paradigm [40], and had exported in corresponding XMI files. For 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 are extending it to handle the composition and exception handling dependencies.

VII.V EXPERIMENTAL RESULTS

We have tested the working of SOSDS using case study examples as stated in [29] with inter-service and intra-service dependencies using WSDL.

The system configuration used to run SOSDS 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. All measured times reported in this section are overall times, including parsing and building of the both WSCFG and SOSDG representation. We studied the run-time requirements of our MBGDS algorithm for these case studies and for several runs. Table 1 summarizes the aver-age run-time requirements of MBGDS algorithm. As we are not aware of 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 dynamic slices at different messages of a services, we have calculated the average run-time requirements of the MBGDS algorithm. The performance results of our implementation agree with the theoretical analysis. From the experimental results, it can be observed that the average run-time increases sub-linearly as the no. of service increases in a service choreography.

Table 1. Average runtime of MBGDS algorithm

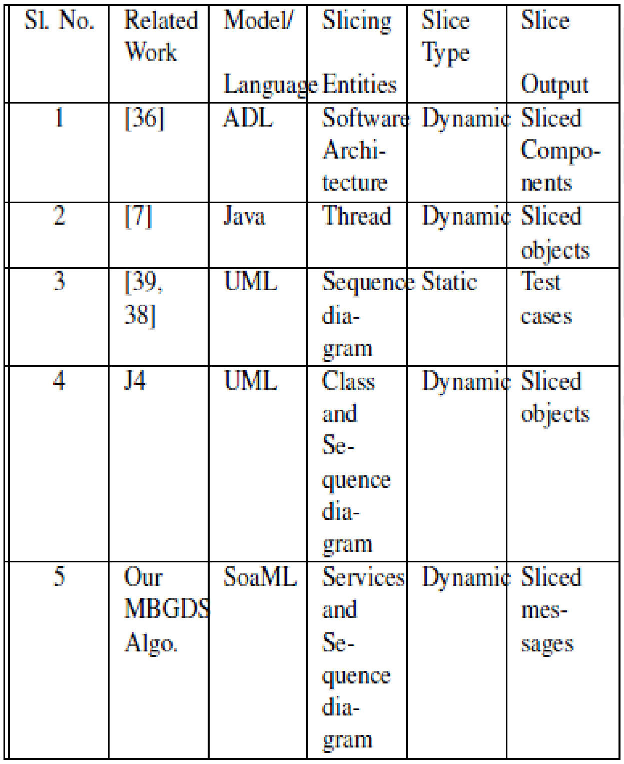
Sl. No.	Name of Case-Study	XMI (#LOT)	# Ser-vices	#LOC	Average Run-Time (in Sec)
1	OSS	15708	11	2145	37.45
2	HAS	5712	4	795	14.04
3	BAS	9150	6	1189	21.06
4	RRTS	5812	4	810	13.10
5	RAS	5789	5	986	12.22
6	JIS	5101	4	790	12.11
7	LIS	9240	6	1190	20.09
8	SCCS	5400	3	589	13.87
9	SAS	5119	4	789	14.78
10	MPSS	5139	3	580	12.45
11	SAMS	5333	5	988	13.89
12	MSAS	5219	4	794	14.12
13	RRS	5318	4	809	14.09
14	MGCAS	5278	5	989	13.18
15	HRS	2411	2	397	13.18

VIII. COMPARISION WITH RELATED WORK

To our best of 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 and dynamic slicing of models/languages. A comparison between the related work is presented in Table 2

However, our MBGDS algorithm for dynamic slicing of service-oriented software incorporates several newer things as compared to other work reported in the literature. One new thing is in the computation of a 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 various site. Slicing based on both model and services can efficiently correlate different services during run-time, and help understand how changing any one of service will impact the rest of the service choreography.

Table 2. Comparison with related work



Sl. No.	Related Work	Model/ Language	Slicing Entities	Slice Type	Slice Output
1	[36]	ADL	Software Architecture	Dynamic	Sliced Components
2	[7]	Java	Thread	Dynamic	Sliced objects
3	[39, 38]	UML	Sequence diagram	Static	Test cases
4	J4	UML	Class and Sequence diagram	Dynamic	Sliced objects
5	Our MBGDS Algo.	SoaML	Services and Sequence diagram	Dynamic	Sliced messages

IX. CONCLUSION

In this paper, we have proposed a novel algorithm for computing dynamic slices of service-oriented programs. We have named our algorithm Marking Based Global Dynamic Slicing (MBDS) algorithm. We consider the SoaML sequence diagram and services. Our algorithm uses Service-Oriented Software Dependency 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.

Our algorithm 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 avail-able. We have developed a slicer to verify the proposed algorithm.

REFERENCES

[1] A.V. K. Shanthi and G. Mohan Kumar, ”Automated Test Cases Generation From UML Sequence Dia-gram”, International Conference on Software and Computer Applications (ICSCA 2012),Volume 41, 2012.

[2] Ashalatha Nayak and Debasis Samanta, ”Automatic Test Data Synthesis Using UML Sequence Diagrams ”, Journal of Object Technology, Volume 9, No.2, March-April 2010.

[3] Bogdan Korel, Inderdeep Singh, Luay Tahat, and Boris Vaysburg, ”Slicing of State Based Models”, In the Proceeding of International Conference of Soft-ware Maintenance, pp 34-43.2003.

[4] Debashree Patnaik, Arup Abhinna Acharya, Durga Prasad Mohapatra, ”Generation of Test Cases Using UML Sequence Diagram in a System With Commu-nication Deadlock”, International Journal of Com-puter Science and Information Technologies, Volume 2(3) , 2011.

[5] Debasish Kundu and Debasis Samanta, ”A Novel Approach to Generate Test Cases From UML Activ-ity Diagrams”, Journal of Object Technology, Volume 8, No.3, May-June 2009.

[6] Deepak Kumar Meena, ”Test Case Generation From UML Interaction Overview Diagram and Sequence Diagram”, A Master Thesis, June 2013.

[7] Durga Prasad Mohapatra, ”Dynmic Slicing of Object-Oriented Programs”, [PhD. thesis]. IIT Kharagpur, May 2005.

[8] Frank Tip, ”A Survey of Program Slicing Techniques”, Journal of Programming Languages ,Vol-ume 3, No 3, pp 121-189, 1995.

[9] Hiralal Agrawal, R. A. DeMillo, and E. H. Spafford, ”Dynamic Slicing in the Presence of Pointers, Ar-rays and Records”, In the Proceeding of the Fourth Symposium on Testing, Analysing and verification (TAV4), pp 60-73, ACM/IEEE-CS, October 1991.

[10] Huzefa Kagdi, Jonathan I. Maletic, and Andrew Sut-ton, ” Context-free Slicing of UML Class Models”, In the Proceeding of 21st IEEE International Confer-ence on Software Maintenance (ICSM’ 05) pp. 635-638, Washington, DC, USA, 2005.

[11] J. Tretmans, ”Testing Concurrent Systems: A For-mal Approach”. In 10th International Conference on Concurrency Theory (CONCUR99), No 1664 in LNCS, pp. 4665,Springer-Verlag, 1999.

[12] Jaiprakash T. Lallchandani, R. Mall, ”Static Slicing of UML Architectural Models”, Journal of Object Technology, Volume 8, No 1, pp.159-188. 2009.

[13] Jianjun Zhao, ”Applying Slicing Technique to Soft-ware Architectures”, In Fourth IEEE International Conference on Engineering of Complex Computer Systems ,pp 87 -98,1998.

[14] Jianjun Zhao, ”Slicing Software Architecture”, A Technical Report of Information Processing Society of Japan,97-SE-117, pp 85-92, Nov 1997.

[15] Jurijs Grigorjevs, ”Model-Driven Testing Approach Based on UML Sequence Diagram”, Scientific Jour-nal of Riga Technical University Computer Science. Applied Computer Systems, Volume 47, 2011.

[16] Kunihiro NODA, Takashi KOBAYASHI, Kiyoshi AGUSA, Shinichiro YAMAMOTO, ”Sequence Dia-gram Slicing”, 16th IEEE Asia-Pacific Software Engineering 2009 Conference, 2009.

[17] Lallchandani and R. Mall, A Dynamic Slicing Tech-nique for UML Architectural Models, IEEE Trans-action on Software Engineering, Volume 37, No 6, 2011.

[18] JGraphT Library, [online]. Available: jgrapht.org, [accessed on 18/03/2014].

[19] M. Clavel, F. Duran, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer, and C. Talcott, ”Maude Manual (Version 2.1.1)”, SRI International, Menlo Park, Apr. 2005.

[20] M. Prasanna and K.R. Chandran, ”Automatic Test Case Generation for UML Object Diagrams Using Genetic Algorithm”, International Journal of Ad-vance Software Computer Application” Volume 1, No. 1, July 2009.

[21] Manpreet Kaur and Rupinder Singh, ”Generation of Test Cases From Sliced Sequence Diagram”, International Journal of Computer Applications, Volume 97, No.5, July 2014.

[22] Mark Weiser, ”Programmers Use Slices When Debugging”,j-CACM, Volume 25, No 7, pp 446-452, 1982.

[23] Mass Soldal Lund and Ketil Stlen, ”Deriving Tests From UML 2.0 Sequence Diagrams With neg and as-sert”, AST06, Shanghai, China, May 23 2006.

[24] Monalisa Sharma and Rajib Mall, Automatic Test Case Generation From UML Models, 10th International Conference on Information Technology, pp.196-201, 2007.

[25] Nicha Kosindrdecha, Jirapun Daengdej, ” A Test Generation Method Based on State Diagram”, Jour-nal of Theoretical and Applied Information Technol-ogy, 2010.

[26] Philip Samuel, Rajib Mall and Sandeep Sahoo, ”UML Sequence Diagram Based Testing Using Slic-ing”, IEEE Indicon 2005 Conference, Chennai, In-dia, 11-13 Dec.2005.

[27] Philip Samuel, Rajib Mall, Pratyush Kanth, ”Auto-matic Test Case Generation From UML Communication Diagram”, Information and Software Technol-ogy(49), Elsevier, 158-171, 2007.

[28] Ranjita Kumari Swain, Vikas Panthi, Prafulla Kumar Behera, ”Test Case Design Using Slicing of UML In-teraction Diagram”,2nd International Conference on Communication, Computing and Security (ICCCS-2012),Elsevier,2012.

[29] Rajib Mall, Fundamentals of Software Engineering, PHI Learning Private Limited, second edition, February 2009.

[30] S. Shanmuga Priya, P. D. Sheba Keizia Malarchelvi, ”Test Path Generation Using Uml Sequence Dia-gram”, International Journal of Advanced Research in Computer Science and Software Engineering, Vol-ume 3, Issue 4, April 2013.

[31] Santosh Kumar Swain, Durga Prasad Mohapatra, and Rajib , ”Test Case Generation Based on Use Case and Sequence Diagram”, International Journal of Soft-ware Engineering , Volume 3, Issue 2,pp 21-52, July 2010.

[32] SOA Testing Technique, [online]. Available: http://www.blog.soatetsting.com, [accessed on 18/03/2014].

[33] Srikant Inaganti, Sriram Arvamudan, ”Testing SOA Application”, BPTrends, April 2008, [online]. Avail-able: www.bptrends.com., [accessed on 18/03/2014].

[34] Sun, J., Liu, Y., Dong, J. S., Pu, G., and Tan, T. H, ” Model-based Methods For Linking Web Service Choreography and Orchestration”, In the Proceeding of the 17th Asia Pacific Software Engineering Con-ference (2010), pages 166-175, 2010.

[35] SOA testing tool survey, [online]. Available: ”http://soatestingresearch.blogspot.in/2008/10/soa-testing-tool-survey.html”, [accessed on 18/03/2014].

[36] Taeho Kim, Yeong-Tae Song, Lawrence Chung, and Dung T. Huynh, ”Dynamic Software ArchitectureSlicing”, 23rd International Computer Software and Applications Conference, COMPSAC ’99, pp. 61-66, Washington, DC, USA, 1999.

[37] V.Mary Sumalatha, G.S.V. P Raju, ”UML Based Automated Test Case Generation Technique Us-ing Activity-Sequence Diagram”, The International Journal of Computer Science and Applications (TI-JCSA), Volume 1, No 9, November 2012.

[38] Vikash Panthi and Durga Prasad Mohapatra, ”Au-tomatic Test Case Generation Using Sequence Di-agram”, In the Proceedings of ICAdc, AISC 174, Springer India, pp 277-284, 2013.

[39] Vikash Panthi, Durga Prasad Mohapatra, ”Automatic Test Case Generation Using Sequence Diagram”, In-ternational Journal of Applied Information System (IJAIS), Volume 2, No.4, May 2012.

[40] Visual Paradigm for UML Enterprise Edition, [on-line]. Available: http://www.visual-paradigm.com, [accessed on 18/03/2014].

[41] WS Monitor Tool,[online]. Available: https://java.net/projects/wsmonitor, [accessed on 18/03/2014].