Chapter 7

Architectural Aspect-Oriented Dynamic Slicing

This chapter proposes a technique for dynamic slicing of aspect-oriented based UML 2.0 sequence diagrams. To represent the classes, aspects, pointcuts and advices in a single intermediate graph is quite difficult and complex in nature. We first extract all the relevant information and interaction between the classes, aspects, pointcuts, join points and advices from a UML sequence diagram specifying the aspect-orientation, into an intermediate representation, which we call Aspect Model Dependency Graph (AMDG). The concept of program slicing in UML models is introduced as a mean to support software maintenance through the understanding, querying, and analysis of software models. For a given slicing criterion, our proposed dynamic slicing algorithm traverses the constructed AMDG to identify the parts that are directly or indirectly affected during the execution, by marking and unmarking the edges of the AMDG when the associated dependencies arise and cease during the run-time, in a specified scenario. The novelty of our approach is that, it eliminates the use of trace files and no new nodes are created during run-time. Also, our approach captures and represents all the necessary constructs among the classes and aspects correctly.

Over a decade ago, Aspect-Oriented Programming (AOP) [37, 39, 54, 60, 72] an emerging programming paradigm has been proposed by Gregor Kiczale’s team from Xerox Palo Alto Research Centre (PARC), which allows software developers to modularize cross cutting concerns (whose implementations would otherwise have been scattered throughout the program, because of the limited abstractions). A concern in mean of software development can be understood as “a specific requirement that must be addressed in order to satisfy the overall system goal”. In the mean of software development process, the non-functional requirements can be considered as cross cutting concerns. Some
common examples of cross cutting concerns are logging, transactions, auditing and security. Developments in aspect-oriented programming languages, such as JBoss [3], AspectJ [28, 56, 61, 73] and Spring AOP [4] and programming mechanisms such as composition, monitoring and refactoring, have been the prominent reasons for adopting AOP. While the emphasis has been on program implementation, it has been argued that applying aspect orientation at the design level can also be beneficial [9, 25, 40].

During software development process, in the design perspective, a software product is ready to be implemented in some programming languages. Usually, the slices are computed from the program source code i.e. during the coding phase of software development life cycle. An alternative approach is to compute slices from specifications like UML models [64, 95]. With this, the slices are derived during the analysis or design stage itself. This has the advantage of allowing slices to be available early in the software development cycle, thereby making the design components more reusable. It makes automatic code generation possible for AOP applications with higher levels of separation of concerns at the generated code [9, 25, 40].

Slicing aspect-oriented based UML models presents some new challenges and difficulties, as the information about the system is distributed across several model views captured through a large number of diagrams. The existing traditional slicing approaches cannot be applied directly to AO based UML diagrams, due to the presence of special features in aspect-oriented programs like aspects, pointcuts, join-points, advices etc. which need a lot of investigation, studies and new ideas in order to achieve a high-level of accuracy in formulating a suitable intermediate representation and then computing the dynamic slices.

To the best of our knowledge, no work has been reported in the literature that describes about computation of dynamic slices from aspect-oriented based UML models. In this chapter, we propose a dependence based intermediate representation to represent aspect-oriented based UML sequence diagrams, which captures the foundational issues surrounding AOP such as join points, pointcuts, aspects and advices and a dynamic slicing algorithm called to compute the dynamic slices from the intermediate representation.

The rest of the chapter is organized as follows. Section 7.1 discusses the basic concepts of aspect-oriented modeling. Section 7.2 describes some definitions and terminologies including construction of the intermediate representation aspect model dependency graph (AMDG). Section 7.3 presents the pseudo code, working, complexity analysis and the correctness proof of our proposed AAODS algorithm. An implementation model for AAODS is pre-
sented in Section 7.4. The comparison with some existing work is presented in Section 7.5. Section 7.6 discusses the summary of the chapter.

7.1 Basic concepts

In this section, we present some basic concepts of UML 2.0 that are relevant to our work and an overview of aspect-oriented modeling.

7.1.1 UML 2.0 Sequence Diagrams

"The Unified Modeling Language (UML) is a standard language for writing software blueprints and can be defined as a modeling language for visualizing, specifying, constructing and documenting the artifacts of a software-intensive system". Modeling is a proven and well-accepted engineering technique as it helps the users to represent the desired structure and behaviour of the system, visualizing and controlling the system’s architecture, exposing opportunities for simplification, reusing and to managing risks.

UML sequence diagrams are used to show the flow of functionality through a use case. They emphasize the ordering of messages and show the chronological sequence of messages, their names and responses and their possible arguments. A sequence diagram is formed by placing the objects that participate in the interaction at the top of the diagram, across the X axis. Typically, the object that initiates the interaction is placed at the left, and increasingly more subordinate objects are placed to the right. The messages that these objects send and receive are placed along the Y axis, in order of increasing time from top to bottom. This gives the users a clear visual cue to the flow of control over time.

In our work, we have considered the aspect-oriented based UML 2.0 sequence diagrams to compute the dynamic slices. An example of aspect-oriented based UML 2.0 sequence diagram is shown in Fig. 7.1. The vertical dashed line in the diagram is called a lifeline that represents the existence of an object over a period of time. Most objects that appear in a sequence diagram will be in existence for the duration of the interaction, so these objects are all aligned at the top of the diagram, with their lifelines drawn from the “top of the diagram to the bottom”. Arrows between the lifelines denote communication between object instances using messages. In the context of sequence diagram, a message can be defined as a request to the receiver object to perform an operation. Synchronous call messages are shown with filled arrow head. They represent operation calls, i.e. first the message is sent and then the execution is suspended, until a response is acknowledged. Asyn-
chronous call messages are shown with open arrow heads. In asynchronous call, first a message is sent and the next message is also sent without receiving a response of the first sent message. The activation (focus of control) is a tall, thin rectangle that shows the period of time during which an object is performing an action, either directly or through a subordinate procedure. The top of the rectangle is aligned with the start of the action and the bottom of rectangle is aligned with its completion. The bottom of the rectangle can be marked with a return message.

UML 2.0 allows an element called note, for adding additional information to the sequence diagram. Generally note is represented in a dog-eared shaped rectangle that is linked to the dashed lines of the lifelines. A note may contain pre-conditions and post-conditions and constraints. In UML 2.0, more complex interactions can be created with “combined fragments”. A combined fragment consists of one or more interaction operands. An interaction operator specifies the purpose of the fragment. Interaction constraints can guard each interaction operand. Messages on their own cannot cross the boundaries of combined fragments, they need a gate which links the two parts of the message. Through the use of combined fragments, the understanding of the number of traces will be more easier. A combined fragment with an operator alt (for alternative) is shown in Fig. 7.1.

7.1.2 Aspect-Oriented Modeling

With the advancements in AOP applications, there is a need for addressing the concerns that are scattered throughout the programs and also the weaving mechanisms in the early phases of aspect-oriented software development. It is important that the system architectural design is represented in a methodology that is easily conveyed to the software practitioners. Therefore, the use of Unified Modeling Language (UML) in aspect-oriented programs specifies that aspect-oriented design models should be used to develop both the architectural design as well as the software specifications. UML is widely being used for representing and constructing the architectural models of software systems. It provides a wide range of visual artifacts to model different aspects of a system. Thus the use of UML in aspect-oriented modeling can be considered as the key for conveying accurately the software specifications that will be used to implement the software. The use of the UML in the aspect-oriented architectural and software design assures traceability, provides a generic structure for problem solving, furnishes abstractions to manage complexity, reduces time-to-market for business problem solutions, decreases development costs, and manages the risk of committing mistakes.
To represent aspect-oriented based UML sequence diagram (\( AS_d \)), we need to find the correct interaction of aspects with the base system. We know that aspects in AOP are similar to classes in OOP. Aspect is the part of the code describing how pointcuts and advices should be combined together. Join points are well defined points in the execution of the code. Join Point is a fundamental concept of AOP identifying an execution point in a system. The categories of join points available in AspectJ (a popular AOP language) are method call and execution, constructor call and execution, read/write access to a field, exception handler execution, and class initialization execution. Pointcuts are used to select relevant join points. A pointcut may select a call to a method and capture the method's context. In other words, we may say that pointcuts specify the weaving rules and join points represent the situations satisfying those rules. During the aspect-oriented software design phase, we need to find how to represent the relationship between the pointcut and aspect. This relationship can be found by studying the behaviour of the join points with their respective aspect through the appropriate pointcuts. We have represented all these three constructs (join point, aspect and pointcut) in a single model diagram, which will be easier to analyze and understand the relationship between the classes and aspects, while computing the dynamic slices.

In our work, we have considered the online railway reservation system as the motivating example. Fig. 7.1 shows the aspect-oriented (woven) sequence diagram (\( AS_d \)) for **issueticket** use case with the security concern AccessControl. We have considered three base/system classes (**Passenger**, **ServiceMonitor** and **Database**), one “aspect” (**AccessControl**) and one “around advice” (**login**). The first alt fragment involving the objects **Accesscontrol** and **login** (in Fig. 7.1) represents the aspect-orientation part of the issue ticket use case. The intervention of aspect plays a vital role as it assures, the system’s security by allowing the authenticate users for the enquiry process. So, during the design phase, a careful study has to be made for capturing the interactions of classes and aspects. In Fig. 7.1, we have considered details() method of Passenger (base class) as join points which are captured by pointcut **resrvmonitor** through a “call” pointcut designator. Pointcut designator can be defined as a formula that specifies the set of join points to which a piece of advice is applicable. A “pointcut designator” identifies all types of join points and matches the appropriate join points at runtime. The authenticate users are allowed to access the system. All the actions for **issueticket** are depicted in the sequence diagrams as a series of events.

The around advice on **resrvmonitor** (pointcut) in AccessControl aspect
stops the current process and takes over the control. It results in two scenarios: either stopping the current process or resuming the stopped process by giving the control back. We have represented the constraints “pointcut” in the sequence diagram by a dotted line going from the matching point at the base class Passenger to the aspect AccessControl. This dotted line can also be considered as synchronous message call because it represents an operation call. First the message is sent and then the execution is suspended, until a response is acknowledged. It’s functionality will be the same in the diagram as of normal synchronous message calls. We have made the arrows dotted in order to avoid the confusion with this call with standard sequence diagram’s synchronous message calls. In Fig. 7.1, after validating the authenticated users, the control moves back to the suspended base class Passenger, which resumes the process by allowing authorised users for making enquiry about the details (availability, fare and berth) of trains.
### 7.2 Definitions and Terminologies

In this section, we present some basic definitions, and terminologies associated with our proposed intermediate program representation and dynamic slicing algorithm. We have named our proposed intermediate representation Aspect Model Dependency Graph (AMDG) to represent aspect-oriented based UML 2.0 sequence diagrams \((AS_d)\). AMDG represents the interaction and behavioural aspects that are modelled among the classes and aspects of a system. The process of constructing an AMDG involves drawing the nodes and edges, where nodes represent messages or notes and edges represent either data or control dependences associated with the nodes. An AMDG provides an integrated view of the entire system.

**Definition 7.2.1. Aspect Model Dependency Graph (AMDG):** We define Aspect Model Dependency Graph (AMDG) as a directed graph \(G = (N, E)\), where \(N\) is a set of nodes and \(E\) is a set of edges. AMDG shows the dependency of a given node on the other nodes. We have used AMDG as the intermediate program representation in our work. In this context, a node represents either a message or a note in the sequence diagram \((AS_d)\) and an edge represents either a control or data dependency associated with the nodes. The AMDG of the sequence diagram \((AS_d)\) given in Fig. 7.1 is shown in Fig. 7.2 [94].

![Aspect Model Dependency Graph (AMDG) of Figure. 7.1](image-url)

Figure 7.2: Aspect Model Dependency Graph (AMDG) of Figure. 7.1
Definition 7.2.2. recentDef(var): For each variable var in the message (mes) of the sequence diagram, recentDef(var) represents the node corresponding to the most recent definition of var with respect to some point s in an execution.

Definition 7.2.3. Def(var): Let var be a variable used in a sequence diagram. A node v of the AMDG is said to be a def(var) node if v represents a definition(assignment) statement that defines the variable var.

Definition 7.2.4. Data Dependence: A node n of AMDG is said to be data dependent on a node m of AMDG, if there exists a variable var in the corresponding sequence diagram such that all of the following hold:
1. the node m defines var,
2. the node n uses var, and
3. there exists a directed path from m to n along which there is no intervening definition of var.

Definition 7.2.5. dyslice(m): For each node m of the AMDG (i.e. message m of the sequence diagram), dyslice(m) represents the dynamic slice with respect to the most recent execution of node m.

7.3 Architectural Aspect-Oriented Dynamic Slicing Algorithm

In this section, we have proposed an algorithm for computing the dynamic slices of aspect-oriented models. We have named our proposed algorithm Architectural Aspect-Oriented Dynamic Slicing (AAODS) Algorithm. AAODS takes an aspect-oriented based UML sequence diagram and a slicing criterion as its input and produces the dynamic slice as output. The operation of our dynamic slicing algorithm is divided into three main phases: (i) construction of Aspect Model Dependency Graph (AMDG), (ii) Managing the AMDG during run-time and (iii) computation of dynamic slice.

In the first step, the AMDG is constructed from a static analysis of the aspect-oriented based UML 2.0 sequence diagram. Then, the AMDG is traversed. The traversal of AMDG helps to identify different architectural elements forming the slice. The dynamic slice of an aspect-oriented based UML sequence diagram is computed from its corresponding AMDG. An AMDG is created statically only once. For each message in the sequence diagram, there will be a corresponding node in the AMDG. After creating the AMDG statically, our dynamic slicing algorithm marks the corresponding nodes when the
associated dependencies arise and unmarks when the associated dependencies cease during runtime.

Let the total nodes of AMDG be \( x_1, x_2, \ldots, x_k \). During the execution process of AMDG, let \( \text{dyslice}(m) \) denote the dynamic slice with respect to the node \( m \) for the most recent execution. Let \((m, x_1), (m, x_2), \ldots, (m, x_k)\) be all the marked (control or data) dependence edges of \( m \) in the updated AMDG. Then,

\[
\text{dyslice}(m) = [x_1, x_2, x_3, \ldots, x_k] \cup \text{dyslice}(x_1) \cup \text{dyslice}(x_2) \cup \text{dyslice}(x_3) \cup \ldots \cup \text{dyslice}(x_k).
\]

(7.1)

### 7.3.1 Algorithm: Architectural Aspect-Oriented Dynamic Slicing (AAODS) Algorithm

This subsection presents our AAODS algorithm in pseudo-code form. It assumes that the sequence diagrams are given in XML format as the input.

**Algorithm: AAODS**

**Stage 1: Construction of AMDG statically**

1. AMDG Construction
   - (a) **Node Construction**
     1. For each message \( m \) represented by arrows in the aspect-oriented based UML 2.0 sequence diagram \((AS_d)\), do the following:
        A. create a node \( n \) for each message of the sequence diagram.
        B. initialize the node \( n \) with its type, list of messages and variables (if any) used or defined, and its scope.
   
     (b) **Add control dependence edges**
     for each test (predicate) node \( m \), do the followings:
     for each node \( x \) within the scope of the node \( m \), do the following:
     Add a control dependence edge \((m, x)\) and mark it.

   (c) **Add data dependence edges**
   for each node \( x \), do the followings:
   for each message \((mes)\) of the sequence diagram \((AS_d)\), used at node \( x \), do the following:
   for each reaching definition \( m \) of \((mes)\), do the following,
   Add a data dependence edge \((m, x)\) and unmark it.

**Stage 2: Managing AMDG during run-time**
1. Initialization. Do the following, before traversing the intermediate dependence graph (AMDG).

   (a) Unmark all the control and data dependence edges of AMDG.
   (b) Mark each control dependence edge \((m,x)\), for which \(x\) is not a loop control node.
   (c) Set \(\text{dyslice}(n) = \phi\) for every node \(n\) representing each message \(mes\), of the AMDG.
   (d) Set \(\text{recentDef}(mes) = \phi\) for every message \(mes\) of the sequence diagram \((AS_d)\).

2. Run-time updations. Traverse the aspect-oriented based sequence diagram \((AS_d)\), with the given set of input values and do the following after each message \(mes\) for the corresponding node \(m\) of the sequence diagram is processed:

   (a) For each message \(mes\) used at node \(m\), do the followings:
      i. Unmark all the incoming marked data dependence edges associated with the message \(mes\) corresponding to the previous execution of message \(mes\), with respect to node \(m\).
      ii. Mark the data dependence edge \((n,x)\), where \(x = (\text{recentDef}(m))\).
   (b) Update dynamic slice for different dependencies.
      i. Let \((x_1, m), (x_2, m), \ldots, (x_j, m)\) be the set of marked incoming data dependence edges to node \(m\) in the AMDG. Then, update the dynamic slice set as:

         \[
         \text{dyslice}(m) = [x_1, x_2, x_3, \ldots, x_j] \cup \text{dyslice}(x_1) \cup \text{dyslice}(x_2) \cup \text{dyslice}(x_3) \cup \ldots \cup \text{dyslice}(x_j).
         \]

         where, \(x_1, x_2, x_3, \ldots, x_j\) are the initial vertices of the corresponding marked incoming edges of node \(m\).
      ii. Let \((c, m)\) be the marked control dependence edge. Then, update the dynamic slice set as:

         \[
         \text{dyslice}(m) = \text{dyslice}(m) \cup [c] \cup \text{dyslice}(c).
         \]
   (c) If \(m\) is a \(\text{Def}(mes)\) message, then update \(\text{recentDef}(mes) = m\).
   (d) If \(m\) is a loop control node, then,
      i. If this execution of \(m\) corresponds to the entry to the loop, then mark each control dependence edge \((x, m)\).
ii. If this execution of \( m \) corresponds to the *exit* of the loop, then unmark each incoming control dependence edge \((x, m)\).

**Stage 3: Computation of Dynamic Slice**

1. for every message \( mes \), used at node \( m \), do the followings:
   Let \((r, m)\) be a *marked data dependence edge* corresponding to the *most recent* definition of message \( mes \) and \((c, m)\) be the *marked control dependence edge*. Then,
   
   \[
   dyslice(m) = [x, c] \cup dyslice(x) \cup dyslice(c)
   \]

2. Exit when encounter the terminate message.

### 7.3.2 Working of the AAODS Algorithm

We illustrate the working of our AAODS algorithm with the help of an example. Consider the aspect-oriented based UML 2.0 sequence diagram for the issueticket use case of Online railway reservation system shown in Fig. 7.1 and its AMDG shown in Fig. 7.2. The updated AMDG after applying stage 2 of our AAODS algorithm is shown in Fig. 7.3. We are interested to compute the dynamic slice at message number 21 of Fig. 7.1. So, let us assume the slicing criterion as \(\langle 21, \text{ticket} \rangle\), where 21 is the message number in the sequence diagram and \( \text{ticket} \) is the variable associated with message number 21 given in Fig. 7.1. Now consider the input values \(\text{validate} = \text{“yes”}, \text{avail} = \text{“yes”}, \text{fare}(f) = \text{“}1000\text{“} \) and \( f = \text{“yes”}. \) Now, we explain how our algorithm computes the dynamic slice. To this input value, our AAODS algorithm will execute the messages 1, 2, 5, 6, 7, 8, 9, 10, 13, 14, 15, 16, 19, 20, 21 in order. So, our AAODS algorithm marks the edges \((1,2), (2,5), (5,6), (6,7), (7,8), (8,9), (9,10), (10,13), (13,14), (14,15), (15,16), (16,19)\), in the AMDG given in Fig. 7.2. During the Initialization step, our algorithm first unmarks all the edges of AMDG, *marks* only the control dependence edges \((m,x)\) for which \( x \) is not a loop control node and sets the \(dyslice(n)\) and \(\text{recentDef}(mes)\) as \(\phi\), for every node \( n \) representing each message \( mes \), of the AMDG. In our defined AMDG, message number 1 is control dependent on message number 2, where 2 is not a loop control node, so our algorithm marks the *control dependence edge* \((1,2)\). Similarly, our algorithm also marks the message number \((7,8)\) and \((9,10)\), which are *control dependence edges* and not encountered in a loop. Similarly, the algorithm also marks the data dependence edges \((2,5), (8,9), (10,13), (15,16), (16,19)\). All the marked edges in Fig. 7.3 are shown in bold lines.
Now we shall find a backward dynamic slice computed with respect to slicing criterion \( ⟨21, \text{ticket}⟩ \). According to the AAODS algorithm, the dynamic slice at node 21, (representing a message number in the aspect-oriented based UML 2.0 sequence diagram \( (AS_d) \)) is given by the expression:

\[
dyslice(21) = [19, 20] \cup dyslice(19) \cup dyslice(20)
\]

By evaluating the expression in a recursive manner, we can get the final dynamic slice for message number 21. During run-time, the dynamic slice for each node, is computed immediately after the execution of the message. Although message numbers 3, 4, 11, 12, 17, 18 can be reached from message number 21, they cannot be included in the dynamic slice. Our algorithm successfully eliminates message numbers 3, 4, 11, 12, 17, 18 from the final resulting dynamic slice. The final dynamic slice includes the nodes 1, 2, 5, 6, 7, 8, 9, 10, 13, 14, 15, 16, 19, 20, 21. The shaded vertices shown in Fig. 7.3 denote the messages included in the dynamic slice with respect to the slicing criterion \( ⟨21, \text{ticket}⟩ \). Thus, our algorithm computes more precise and correct dynamic slices.

![Figure 7.3: The updated AMDG of the aspect-oriented based UML 2.0 sequence diagram given in Fig. 7.1 w.r.t. slicing criterion \( ⟨21, \text{ticket}⟩ \)](image)

### 7.3.3 Complexity Analysis

In the following, we analyse the space and time complexities of our AAODS algorithm.
7.3 Architectural Aspect-Oriented Dynamic Slicing Algorithm

Space Complexity

Let \((AS_d)\) be an aspect-oriented UML 2.0 based sequence diagram having \(a\) messages. The AMDG constructed in the Stage 1 are directed graphs on \(n\) nodes. A graph on \(n\) nodes with optionally marked edges requires \(O(n^2)\) space. So, the space requirement for AMDG of \((AS_d)\) is \(O(n^2)\). We need the following additional run-time space for managing the intermediate program representation (AMDG).

1. To store the dyslice(\(m\)) for every message of the sequence diagram \((AS_d)\), at most \(O(n)\) space is required, as the maximum size of the slice is equal to the number of messages of the sequence diagram \((AS_d)\). So, for \(n\) messages, the space requirement for dyslice(\(m\)) is \(O(n^2)\).

2. To store \((\text{recentDef}(\text{var}))\) for every variable of message \((\text{mes})\) of sequence diagram \((AS_d)\), at most \(O(n)\) space is required.

So, the space complexity of the AAODS algorithm is \(O(n^2)\), where \(n\) is the number of messages of the aspect-oriented based UML 2.0 sequence diagram.

Time Complexity

Let \((AS_d)\) be an aspect-oriented based UML 2.0 sequence diagram having \(n\) number of messages. To determine the time complexity, we need to consider two factors. The first one the execution time requirement for the run-time maintenance of AMDG. The second one is the time required to calculate the dyslice(\(m\)).

The time needed to store the required information at each node is \(O(n)\), where \(n\) is the number of messages in the sequence diagram \((AS_d)\). The time required for traversing the complete AMDG is \(O(n^2)\), where \(n\) is the number of messages in the sequence diagram \((AS_d)\). Hence, the worst case time complexity of our AAODS algorithm for computing the dynamic slice is \(O(n^2S)\), where \(S\) is the length (in time) while traversing the AMDG and calculating the dynamic slice by updating the dyslice set for different existing dependencies.

7.3.4 Correctness of AAODS Algorithm

In this subsection, we sketch the proof of correctness of our AAODS algorithm.

**Theorem 7.3.1.** AAODS algorithm always find a correct dynamic slice with respect to a given slicing condition.
Proof. We can prove this through mathematical induction. Let $A S_d$ be a sequence diagram for which we want to compute the dynamic slice using AAODS algorithm. According to the definition, for any set of input values, the computed dynamic slice with respect to the first executed message is certainly correct. Using this argument, we establish that the dynamic slice with respect to the second executed message is also correct. During the execution, we assume that the AAODS algorithm has produced correct dynamic slice prior to the present execution of a node $s$ of the AMDG. Let $\text{var}$ be a variable used at $s$, and $\text{dyslice}(s, \text{var})$ be the dynamic slice with respect to the slicing criterion $\langle s, \text{var} \rangle$ for the present execution of the node $s$. Let the node $d = (\text{recentDef}(\text{var}))$ is the reaching definition of the variable $\text{var}$ for the present execution of the node $s$. The node $d$ is executed prior to the current execution of the node $s$ and a dynamic slicing criterion, which contains all those nodes that affect the current value of variable $\text{var}$ used or defined at $s$, our AAODS algorithm has marked all the incoming edges to $d$ only from those nodes on which $d$ is dependent during execution. Further the steps 2(a), 2(c) and 2(d) of Stage 2 of our algorithm ensure that the node is data or control dependent on a node $v$ iff the edges $(s, v)$ is marked in the updated AMDG. Let $x_1, x_2, \ldots, x_k$ be all the nodes on which $s$ is data or control dependent with respect to its present execution. Then,

$$\text{dyslice}(m) = [x_1, x_2, \ldots, x_k] \cup \text{dyslice}(x_1) \cup \text{dyslice}(x_2) \cup \ldots \cup \text{dyslice}(x_k)$$

Since, $\text{dyslice}(x_1) \ldots \text{dyslice}(x_k)$ are all correct dynamic slices, the dynamic slice $\text{dyslice}(s)$ computed by Stage 2 of the algorithm must also be correct. Further, Stage 3 of the algorithm guarantees that the algorithm stops when it encounters a termination message during execution. This establishes the correctness of the algorithm.

$$\square$$

7.4 DSAAM: Our Dynamic Slicing Tool

In this section, we present a brief description of our tool called Dynamic Slicer for Architectural Aspect-oriented Models (DSAAM) which we have developed to implement our dynamic slicing algorithm for aspect-oriented models. The objective of this tool is to compute the dynamic slice of a aspect-oriented model with respect to any given slicing criterion. The current implementation handles only a limited subset of aspect-oriented software models. In the following, we briefly discuss the design, implementation, and working of our slicing tool.
7.4 DSAAM: Our Dynamic Slicing Tool

7.4.1 Overview of the Design of DSAAM

Fig. 7.4 represents a schematic diagram, which describes the working of the slicing tool DSAAM. The arrows in the figure show the data-flow among the different blocks of the tool. The blocks shown in rectangular boxes represent executable components and the blocks shown in ellipses represent passive components of the slicing tool.

![Figure 7.4: Architectural representation of the working of DSCAP](image)

DSAAM takes UML architectural model in XML format as an input. The overall control for the Slicer (DSAAM) is done through a coordinator with the help of a graphical user interface (GUI). The coordinator takes user input from the GUI, interacts with other relevant components to extract the desired results and returns the output back to the GUI. The Parser Module is used to extract information regarding the interaction between objects of different classes. We have used the Document Object Model (DOM) API of Java and AspectJ for parsing XML files. XML represents the data associated with UML models as documents then enabling DOM to be used to manage this data. The information gathered using the Parser Module is then used to initialize all the data structures needed to construct the AMDG. The Graph Construction Module constructs the AMDG by Injecting Module. The constructed AMDG is used by the Slicer as an input. The Slicer block also takes the slicing criterion as input from the Graphical User Interface (GUI) block and outputs the computed dynamic slice of the given aspect-oriented model. The GUI
block encapsulates the complexities involved in the functioning of all the other blocks by supporting user-friendly interface to the user.

### 7.4.2 Implementation

We have implemented our AAODS algorithm for aspect-oriented model represented in UML sequence diagram. Our dynamic slicing tool is coded in Java and uses XML parser. The different types of messages involved in the sequence diagram shown in Fig. 7.1 is listed in Table 7.1. The XML parser uses these message information to generates the requisite data structures and related information to the graph construction module to generate AMDG as intermediate representation statically once. Also while constructing AMDG, we store the data in data structures called class AMDG. This class contains the the details of the message interactions happens in the sequence diagrams of object class and aspect class. The AMDG is constructed using by using the sequence diagram for object class and the sequence diagram for aspect class. For storing the AMDG we have used a two dimensional integer array:

```java
int amdg[][];
```

If there is an edge from node $x$ to $y$ then $amdg[x][y] = 1$, else 0. The code used for this is as given in Table 7.3.

<table>
<thead>
<tr>
<th>Code</th>
<th>Message Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Statement Messages</td>
</tr>
<tr>
<td>1</td>
<td>Request Message</td>
</tr>
<tr>
<td>2</td>
<td>Accept Message</td>
</tr>
<tr>
<td>3</td>
<td>Terminate Message</td>
</tr>
<tr>
<td>4</td>
<td>Acknowledgment</td>
</tr>
</tbody>
</table>

This class contains the the details of the message interactions happens in the sequence diagrams of object class and aspect class.

<table>
<thead>
<tr>
<th>Code</th>
<th>Edge Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Edge does not exist</td>
</tr>
<tr>
<td>1</td>
<td>Control Dependence Edges</td>
</tr>
<tr>
<td>2</td>
<td>Data Dependence Edges</td>
</tr>
</tbody>
</table>

After constructing the AMDG statically, we invoke the `dyslice()` method, which marks and unmarks the edges of AMDG appropriately and updates the dynamic slice. The dynamic slice of each statement is stored in a two dimensional integer array named: int `archslicex``[]``[]`. The first dimension of `archslicex` denotes the node number to which the dynamic slice belongs to.
For example, if the slicing criterion is $\langle x, a \rangle$ where $x$ is the corresponding node for computing dynamic slice, then the one dimension array $\text{archslic}[x][]$ denotes dynamic slice with respect to node $x$. If the node $y$ is included in the dynamic slice of node $x$, then $\text{archslic}[x][y] = 1$, else 0. Finally when the dynamic slice of a particular statement is requested, our slicer DSAAM provides the dynamic slice for the given slicing criterion.

### 7.4.3 Working with DSAAM

First, we have tested the working of our tool DSAAM on the sample aspect-oriented sequence diagram given in Fig. 7.1. Fig 7.5 shows the result after computing the dynamic slice for the slicing criterion $\langle 21, \text{ticket} \rangle$ for the sequence diagram shown in Fig. 7.1.

![Dynamic Slice computed for the sequence diagram shown in Fig. 7.1 by using DSAAM Tool with respect to the Slicing Criterion $\langle 21, \text{ticket} \rangle$](image)

Then, we have tasted DSAAM on some other sequence diagram models such as cancel ticket, check seat availability etc. for railway reservation system. However as already mentioned, currently DSAAM can handle only one sequence diagram related to a specific software model.

We analyzed the run-time requirements of our AAODS algorithm for several sequence diagrams and for several scenarios. Table 7.3 summarizes the average run-time requirements of AAODS algorithm. In the absence of any existing algorithm for dynamic slicing of architectural aspect-oriented models, we have presented only the results obtained from our experiments. The performance results of our implementation agree with the theoretical analysis.
### Table 7.3: Average run-time of AAODS algorithm

<table>
<thead>
<tr>
<th>Sl. No.</th>
<th>Name of Use-case</th>
<th>Name of the system</th>
<th># Objects</th>
<th># Fragments</th>
<th>No. of Messages</th>
<th>Avg. Run-Time (in Sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>issueticket</td>
<td>Online Railway Reservation System</td>
<td>5</td>
<td>3</td>
<td>21</td>
<td>0.11</td>
</tr>
<tr>
<td>2</td>
<td>cancelticket</td>
<td>Online Railway Reservation System</td>
<td>8</td>
<td>2</td>
<td>26</td>
<td>0.13</td>
</tr>
<tr>
<td>3</td>
<td>withdraw</td>
<td>ATM</td>
<td>10</td>
<td>3</td>
<td>35</td>
<td>0.15</td>
</tr>
<tr>
<td>4</td>
<td>issuebook</td>
<td>Library Management System</td>
<td>13</td>
<td>3</td>
<td>39</td>
<td>0.17</td>
</tr>
<tr>
<td>5</td>
<td>patient-discharge</td>
<td>Hospital Management System</td>
<td>15</td>
<td>3</td>
<td>42</td>
<td>0.18</td>
</tr>
</tbody>
</table>

#### 7.5 Comparison with Related Work

In the absence of any directly comparable work, we compare our proposed algorithm with the existing dynamic slicing algorithms of object-oriented and aspect-oriented software. All dynamic slicing algorithms for object-oriented programs reported [65, 77, 80, 81, 83, 101, 109, 115] were based on raw code for computation of the slice. These reported work [65, 77, 80, 81, 83, 101, 109, 115] have not considered slicing of object-oriented design models.

A number of algorithms computing static and dynamic slicing of aspect-oriented programs had been reported in literature [23, 84, 97, 116, 117]. But, they have not considered slicing of UML diagrams. But, our proposed AAODS algorithm computes the dynamic slice of aspect-oriented software at architectural level.

However, Lallchandani et al. [64] proposed an algorithm for computing the dynamic slicing for UML architectural model. In their approach they considered a generic class and sequence diagrams for object-oriented software. Then, they have constructed an intermediate representation termed as Model Dependency Graph (MDG) by combining Class Dependency Graph (CDG) and Sequence Dependency Graph (SDG) where the CDG and SDG are constructed from the generic class and sequence diagrams respectively. The slices are computed by updating the MDG. But, they have not considered any aspect-oriented constructs in their proposed slicing algorithms.
7.6 Summary

In this chapter, we have proposed a novel technique for computing dynamic slices of Architectural Aspect-Oriented models by considering sequence diagrams using UML 2.0 representations. In this work we have introduced Aspect Model Dependency Graph (AMDG) as the intermediate representation. We have named our dynamic slicing algorithm as Architectural Aspect-Oriented Dynamic Slicing (AAODS) algorithm. Our algorithm marks and unmarks the control dependence and data dependence edges of AMDG appropriately as when the dependencies arise and cease during runtime. We have constructed a tool named Dynamic Slicer for Architectural Aspect-oriented Models (DSAAM) for the implementation of our proposed algorithm. Our AAODS algorithm achieves fast response time on distributed network. Advantage of our algorithm is that it can support maintenance through understanding, querying and analysis. It captures and represents all the necessary constructs between the classes and aspects correctly.