Infosys Labs Briefings
VOL 11 NO 2
2013

Architectural Aspect-Oriented Dynamic Slicing

By Abhishek Ray, Siba Mishra and Durga Prasad Mohapatra PhD

Slicing of AOP at Design phase reduces computation time and increases testability and usability

A novel approach for dynamic slicing of aspect-oriented software based on UML 2.0 sequence diagrams is discussed in this paper. To represent the classes, aspects, pointcuts and advices in a single intermediate graph is quite difficult and complex in nature. Firstly, we construct an UML sequence diagram representing all relevant information and interaction between the classes, aspects, pointcuts, join points and advices. Then, an intermediate representation termed as Aspect Model Dependency Graph (AMDG) is constructed from the UML sequence diagram. The concept of program slicing in UML models is introduced as a mean to support software maintenance through understanding, querying, and analysis. 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 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 between the classes and aspects correctly.

Program slicing [3, 5, 11, 12, 21, 25, 28, 34, 36, 35, 37] was introduced by Mark Weiser based on the observation that programmers have some abstractions about the program during debugging. Program slicing [3, 5, 11, 12, 21, 25, 28, 34, 36, 35, 37] can be defined as a program analysis, decomposition and practical disintegration software reverse engineering technique that extracts certain parts from program statements, relevant to particular computation or some point of interest, known as "slicing criterion". The impact, applications and power of program slicing comes from the ability to assist the researchers and software developers in lots of tedious and error prone tasks such as program debugging, testing, integration, software safety, understanding, re-engineering, decompilation and software maintenance [3, 5, 11, 12, 21, 25, 28, 34, 36, 35, 37]. Slicing does this by extracting an algorithm whose computation may be scattered throughout a program from intervening irrelevant statements specified within the criterion.

Weiser's [35, 36, 37] originally introduced slicing. Static Slicing uses static analysis to derive slices. The source code of the program is analyzed and the slices are computed for all possible input values. No assumptions may be made about the input values, however, predicates may evaluate either to true or false which leads to relatively large slices. Korel and Laski [16] introduced the concept of dynamic slicing which is used to identify the parts of a program that contribute to the computation of the selected function for a given program execution (program input). Dynamic slicing [16] may help to narrow down the parts of program that contributes to the computation function of interest for a particular program input. Dynamic slices [16] are frequently much smaller than static slices and thus used to understand program execution. In a later work, Korel [15] had shown that slicing could be used as a reduction technique on specifications like state based models.

Over a decade ago, Aspect-Oriented Programming (AOP), [8, 9, 13, 18, 22] an emerging programming paradigm have been proposed by Gregor Kiczale's team from Xerox Palo Alto Research Centre (PARC). AOP 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 [1], AspectJ [7, 14, 19, 23] and Spring AOP [2]. 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 [4, 6, 10].

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 [20, 30]. With this, the slices are derived from 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 also makes automatic code generation possible for AOP systems with higher levels of separation of concerns at the generated code [4, 6, 10].

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 needs 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 computation of dynamic slices from aspect-oriented based UML models. In this paper, first we construct a dependence based intermediate representation to represent aspect-oriented based UML sequence diagrams. Then, we propose a slicing algorithm to compute the dynamic slices using the above constructed intermediate representation.

BASIC CONCEPTS

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

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 communicate with the desired structure and behaviour of the system, visualizing and controlling the system's architecture, exposing opportunities for simplification, reuse and to manage risk.

UML sequence diagrams are used to show the flow of functionality through a use case. They emphasize the time ordering of messages and shows chronological sequence of the 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. Next, 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 diagrams is shown in Fig. 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 an interaction 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 denotes the communication between object instances using messages. In the context of sequence diagram, a message can be defined a request to the receiver object to perform an operation. *Synchronous call messages* are shown with filled arrow head. It represents an operation call, i.e. first the message is send then the execution is suspended, until a response is acknowledged. *Asynchronous call message* is shown with an open arrow head. In *asynchronous call*, first a message is send and the next message is send without receiving a response of the first send 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 (and can be marked by a return message).

UML 2.0 allows an element called note, for adding additional information to the sequence diagram. Generally notes are represented in a dog-eared shaped rectangle that is linked to the dashed lines of the lifelines. A note may contain pre and post conditions and constraints. In UML 2.0 more complex interactions can be created with "combined fragment". *A combined fragment* consists of one or more interaction operands. An interaction operator kind 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 number of traces will be in a more compact and concise manner. *A combined fragment* with an operator alt (for alternative) is shown in Fig. 1.

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 UML in 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 mistakes.

To represent aspect-oriented based UML sequence diagram (AS_d), we need to find the correct interaction of aspect with the base system. As we know, aspects in AOP are similar to classes in OOP. Aspect is the part of 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 point cuts specify the weaving rules and join point represent the situations satisfying those rules. During the aspect-oriented software design phase, we need to find how to represent the relationship of the pointcut and aspect. This relationship can be found by studying the behaviour of join points with their respective aspect through appropriate pointcuts. We have represented all these three constructs in a single diagram, which will make easier to analyze and understand the relationship between classes and aspects, while computing the dynamic slices.

In our work, we have considered Online railway reservation system as the running

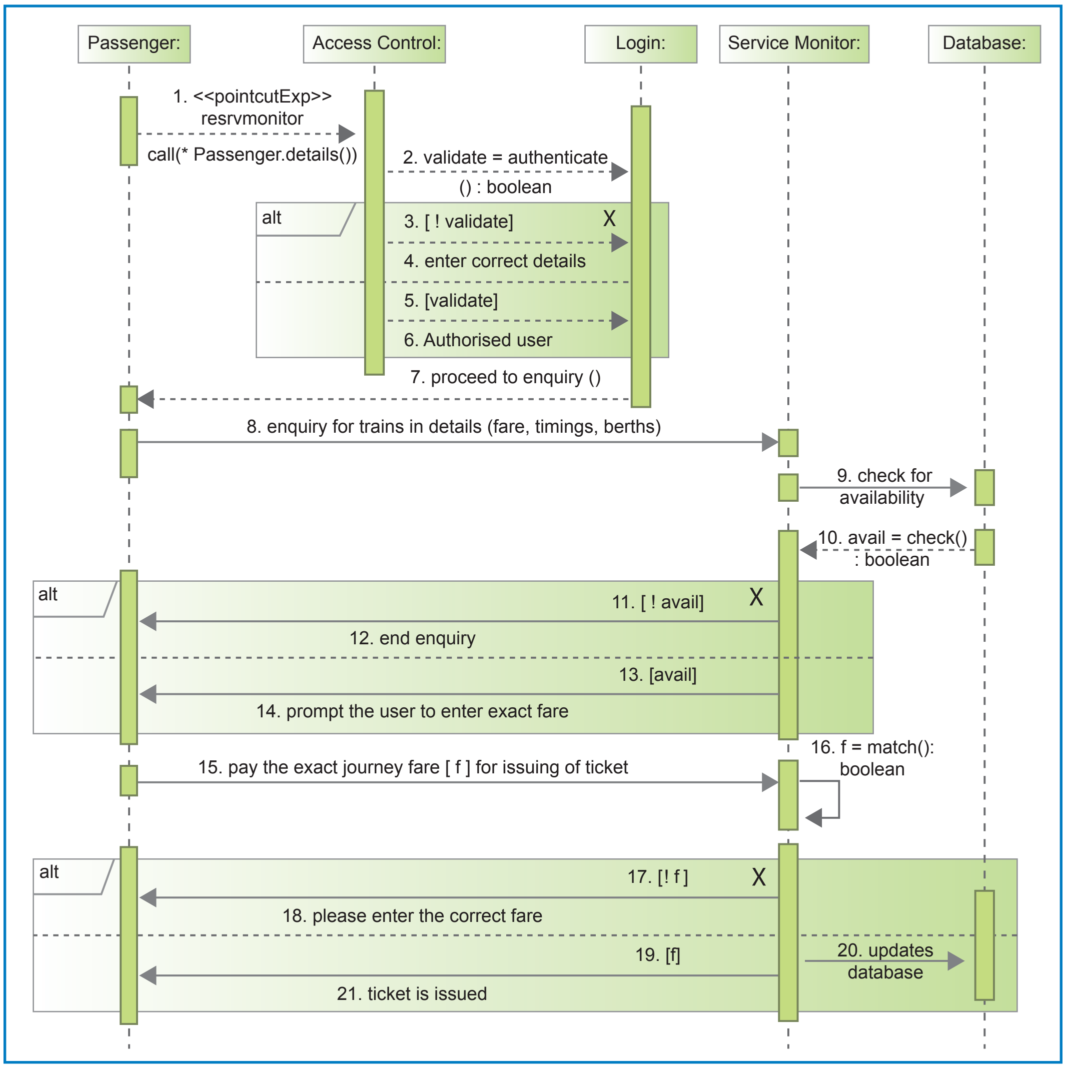
Passenger:
Access Control:
Login:
Service Monitor:
Database:
1. <<pointcutExp>> resrvmonitor
call(* Passenger.details())
2. validate = authenticate () : boolean
alt
3. [! validate]
X
4. enter correct details
5. [validate]
6. Authorised user
7. proceed to enquiry ()
8. enquiry for trains in details (fare, timings, berths)
9. check for availability
10. avail = check() : boolean
alt
11. [! avail]
X
12. end enquiry
13. [avail]
14. prompt the user to enter exact fare
15. pay the exact journey fare [f] for issuing of ticket
16. f = match(): boolean
alt
17. [! f]
X
18. please enter the correct fare
19. [f]
20. updates database
21. ticket is issued

Figure 1: *An Aspect-oriented based UML Sequence Diagram of online railway reservation system for issueticket use case* ***Source:*** *Academic Research*

example. Fig. 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 intervention of aspect plays a vital role as it assures, the system 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 our diagram, we have considered details() method of Passenger (base class) as join points which is 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 Fig. 1 sequence diagram 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 in Fig. 1 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. Here, first the message is sent and then the execution is suspended, until a response is acknowledged. It's functionality in the diagram will be the same as that of normal synchronous message calls. We have made the arrows as dotted in order to avoid the confusion of this call with the standard synchronous message calls in sequence diagrams. In Fig. 1, after validating the authenticate users, control moves back to the suspended base class Passenger, which resumes the process by allowing authorized users for enquiry about the details (availability, fare and berth) of trains.

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 to represent aspect-oriented based UML 2.0 sequence diagrams (AS_d) *Aspect Model Dependency Graph* (AMDG). AMDG represents the *interaction and behavioural* aspects that are modeled between the classes and aspects of a system. The process of constructing an AMDG involves combination of the nodes and edges, where nodes represents a message or a note and edges represents either data or control dependences associated with the nodes. In the absence of raw code, it is difficult to capture the correct control and data dependences in the architectural model. So, we have considered messages and notes as "nodes" in our proposed AMDG. An AMDG provides an *integrated* view of the entire system.

Definition 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, the node represents either a message or a note in the sequence diagram *(AS_d)* and edges represent either control or data dependences associated with the nodes. The AMDG of the sequence diagram *(AS_d)* given in Fig. 1 is shown in Fig. 2.

Definition 2. *Correct Dynamic Slice:* A correct dynamic slice contains all the statements that affect the slicing criterion for a given input value.

Definition 3. *(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.

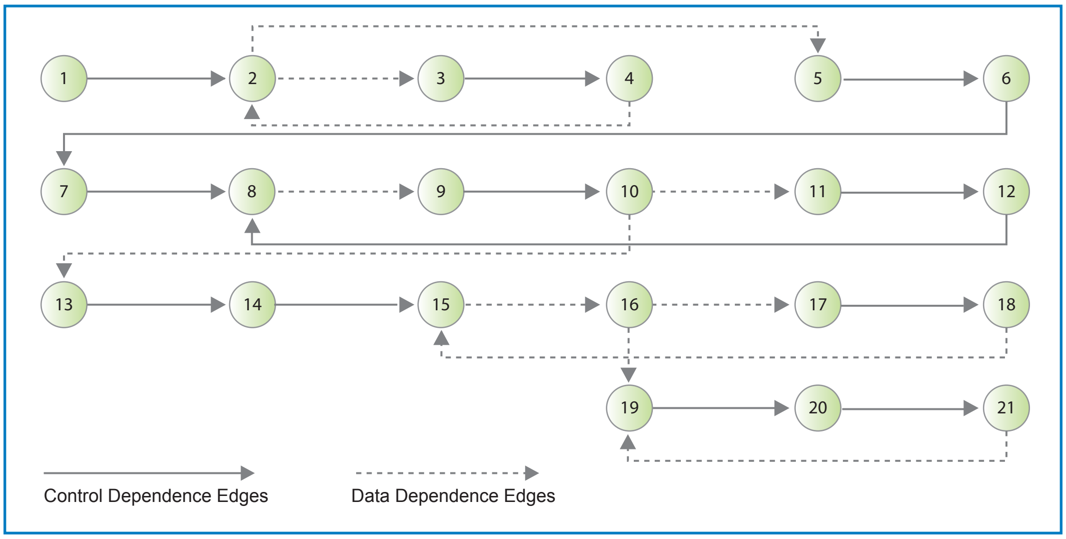

Figure 2: *Aspect Model Dependency Graph (AMDG) of Fig. 1* ***Source:*** *Academic Research*

Definition 4. *(Def(var)):* Let var be a variable used in the 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 5. *Data Dependence:* A node *n* is said to be data dependent on a node *m*, if there exists a variable *var* in the 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 6. *dyslice(m):* For each node m of the AMDG (message m of the sequence diagram), dyslice(m) represents the dynamic slice with respect to the most recent execution of node *m*.

ARCHITECTURAL ASPECT-ORIENTED DYNAMIC SLICING

We have named our dynamic slicing algorithm as Architectural Aspect-Oriented Dynamic Slicing (AAODS) Algorithm. AAODS takes a aspect-oriented based UML sequence diagram and a slicing criterion as its input and produces the computed dynamic slice as output. The operation of our dynamic slicing algorithm can be divided into three main phases: (i) constructing AMDG statically, (ii) Managing the AMDG during run-time and (iii) computing the dynamic slice.

In the first step, the AMDGs are constructed from a static analysis of the aspect-oriented based UML 2.0 sequence diagrams. 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 dependencies arise and unmarks them when the dependencies cease during runtime. Then, we compute the dynamic slice recursively, using Eq. 1. Let x_1, x_2,…, x_k be the nodes of the AMDG. During the execution process of AMDG, let dyslice(*m*) denote the dynamic slice with respect to the node *m* for the most recent execution. Let (m, x_1), (m, x_2), … ,(m, x_k) be all the marked (control or data) dependence edges of m in the updated AMDG.

$$dyslice(m) = [x_1, x_2, ..., x_k] \cup dyslice(x_1) \cup dyslice(x_2) \cup ... \cup dyslice(x_k) \quad (1)$$

Architectural Aspect-Oriented Dynamic Slicing (AAODS) Algorithm

This subsection presents our AAODS algorithm in pseudo-code form. First, we construct the UML 2.0 sequence diagram and then represent the sequence diagrams in XML format.

Algorithm: AAODS

Input: {Slicing Criterion}

Output: Dynamic Slice w.r.t. Slicing Criterion

Stage 1: Construction of AMDG statically

```
1. AMDG Construction
(a) Node Construction
    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 for each message
           of the sequence diagram.

        B. Initialize the node 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
    following,
        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 following,
        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 the AMDG during run-time

```
1. Initialization. Do the following, before
traversing the intermediate dependence graph
(AMDG).

    (a) Set dyslice(n) = Ø for every node
        representing each message mes, of
        the AMDG.

    (b) Set recentDef(mes) = Ø for every
        message mes of the sequence diagram
        (ASd).

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 following:

i. Unmark all the incoming marked data dependence edges associated with the message corresponding to the previous execution of message *mes*, with respect to node *m*.

ii. Mark the data dependence edge (*n,r*), where r = (*recentDef(m)*).

(b) Update dynamic slice for different dependencies.

i. Handling data dependency. Let (r_1, m), (r_2, m), … , (r_j, m) be the set of marked incoming data dependence edges to node m in the AMDG. Then update the dynamic slice set as:

$$dyslice(m) = [r_1, r_2, ..., r_k] \cup dyslice(r_1) \cup dyslice(r_2) \cup ... \cup dyslice(r_j)$$

where, *r1,r2,...,rj* are the initial vertices of the corresponding marked incoming edges of node *m*.

ii.Handling control dependency. Let (*c,m*) be the marked control dependence edge. Then update the dynamic slice set as:

$$dyslice(m) = dyslice(m) \cup [c] \cup dyslice(c)$$

(c) If *m* is a Def(*mes*) message, then update 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: Slice Look-Up:

1. If a slice command is given then carry the following:

For every message *mes*, used at node *m*, do the following:

i. 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

$$dyslice(m) = [r, c] \cup dyslice(r) \cup dyslice(c)$$

ii. Look up *dyslice(m)* for the content of the slice for message *m*.

2. If it encounters the terminate message, then exit else go to *Stage 2*.

Working of the AAODS Algorithm

We illustrate the working of the algorithm with the help of an example. Consider the aspect-oriented based UML 2.0 sequence diagram of online railway reservation system for *issueticket* use case shown in Fig. 1 and the dependence based intermediate program representation (AMDG) shown in Fig. 2. The updated AMDG after applying stage 2 of our AAODS algorithm is shown in Fig. 3. We are interested to compute the dynamic slice at message number 21 of Fig. 1. So, let us assume the slicing criterion as ⟨21, ticket⟩, where 21 is the message number of the sequence diagram and ticket is the variable

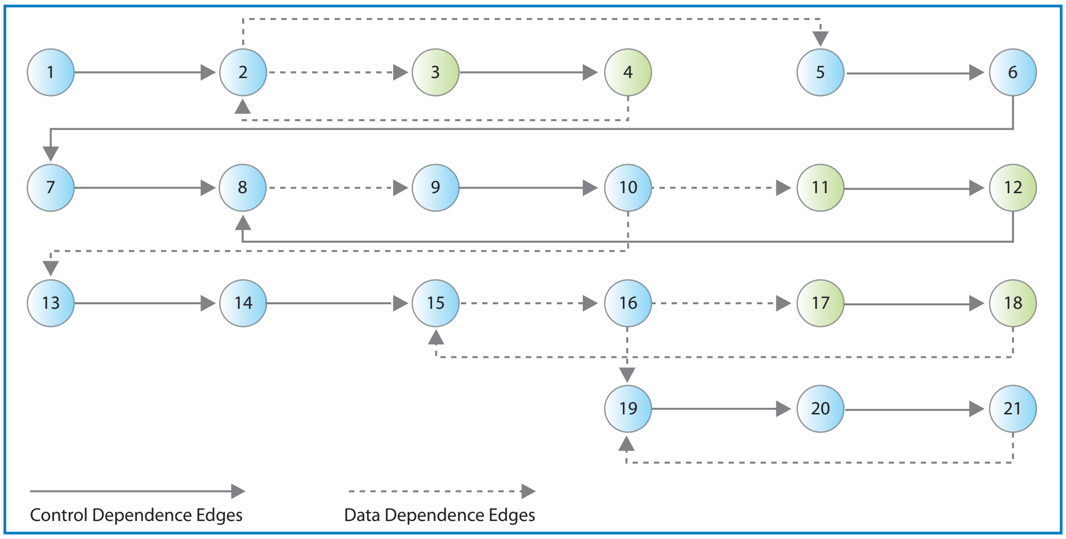

Figure 3: *Aspect Model Dependency Graph (AMDG) of Fig. 1* **Source:** *Academic Research*

associated with message number 21 given in Fig. 1. Now consider the input values validate = "yes", avail = "yes", fare(f) = "1000" and f = "yes". We explain how our algorithm computes the 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), (19, 20), (20, 21). During the Initialization step, our algorithm first unmarks all the edges of AMDG, marks only the control dependencies edges (*m, x*) for which x is not a loop control node and sets the dyslice(*n*) and recentDef(*mes*) as Ø, 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. 3 are shown in bold lines. Now we shall find a backward dynamic slice computed with respect to slicing criterion ⟨21, ticket⟩. According to the AAODS algorithm, the dynamic slice at node 21, (a message 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 number 3, 4, 11, 12, 17, 18 can be reached from message number 21, it 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. 3 denote the messages included in the dynamic slice with respect to the slicing criterion ⟨21, ticket⟩. Thus, our algorithm computes more precise and correct dynamic slices.

Correctness of AAODS Algorithm

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

Theorem 1. AAODS algorithm always finds a correct dynamic slice with respect to a given slicing condition.

Proof: We can prove this through mathematical induction. Let AS_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 *var* be a variable used at *s*, and dyslice (*s*,*var*) be the dynamic slice with respect to the slicing criterion ⟨*s*,*var*⟩ for the present execution of the node *s*. Let the node *d* = (recentDef(*var*)) is the reaching definition of the variable *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 *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 our algorithm ensures 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.

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

Since, $dyslice(x_1)$ ... $dyslice(x_k)$ are all correct dynamic slices, the dynamic slice *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.

Complexity Analysis

In the following, we analyze the space and time complexities of our AAODS 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 nodes. A graph on a nodes with optionally marked edges requires $0(a^2)$ space. So, the space requirement for AMDG of (AS_d) is $0(a^2)$. 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 $0(a)$ 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 a messages, the space requirement for *dyslice*(*m*) is $0(a^2)$.

2. To store (recentDef(*var*)) for every variable of message (*mes*) of sequence diagram (AS_d), at most $0(a)$ space is required.

So, the space complexity of the AAODS algorithm is $0(a^2)$, where a 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 a number of messages. To determine the time complexity, we need to consider two factors. The first one is 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 $0(a)$, where a is the number of messages in the sequence diagram (AS_d). The time required for traversing the complete AMDG is $0(a^2)$, where 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 $0(a^2s)$, 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.

COMPARISION 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 [21, 24, 25, 27, 28, 32, 38, 39] were based on raw code for computation of the slice. These reported works [21, 24, 25, 27, 28, 32, 38, 39] were 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 [26, 29, 31, 33, 40, 41].

Zhao [40] was first to propose a static slicing algorithm for aspect-oriented program. He had proposed dependence based intermediate representation of aspect-oriented software called Aspect-oriented System Dependence Graph (ASDG). This graph is a combination of three parts: a System Dependence Graph (SDG) [21] for non-aspect code, a group of dependence graphs for aspect code called as *Advice Dependence Graph* (ADG), *Introduction Dependence Graph* (IDG) and *Method Dependence Graph* (MDG) and some additional dependence arcs used to connect the system dependence graph to the dependence graphs for aspect code. He had constructed ASDG by first constructing the SDG [21] for the non-aspect code and ADG for aspect code, and then inserted the weaving vertices to the SDG. Then he had used a coordination arc to connect each weaving vertex to the advice start vertex of its corresponding ADG. Next, he had added a call arc between a call vertex and the start vertex of the ADG, IDG, or MDG of the called advice, introduction, or method. Next, Actual and formal parameter vertices are connected by parameter arc. He also added summary arcs between the actual-in and actual-out vertices at call sites for advices, introductions, or methods in a previously analyzed aspect. Since the author considers ASDG as an extension of Larsen-Harrold SDG [21] he used two-pass slicing algorithm proposed in [17] to compute the static slice of an aspect-oriented program based on the ASDG.

Zhao and Rinard [41] extended the *dependence-based representation technique* for AOP [40] to construct a SDG for aspect-oriented program. Zhao [40] had not provided any information to handle pointcut. In this paper the authors had tried to handle the pointcuts by constructing *Module Dependence Graph* (MDG) for each piece of advice, introduction and method in aspects and classes. It then uses existing techniques for object-oriented programs to connect these MDGs at call sites to form a partial SDG. Finally, all MDGs of advice and the partial SDG are weaved together for that method whose behaviour may be affected by the advice; hence the final SDG is constructed.

Braak [33] extended the ASDG proposed by Zhao [40, 41] to include inter-type declarations in the graph. Each inter-type declaration was presented in form of a field or a method as a successor of the particular class. Then, Braak [33] used the two-phase slicing algorithm of Horwitz et al. [12] to find static slice of AspectJ program.

Sahu and Mohapatra [31] were the first to propose an algorithm for dynamic slicing of aspect-oriented programs named as Node-Marking Dynamic Slicing (NMDS) algorithm. They had used an intermediate representation of aspect-oriented program called as Extended Aspect-oriented System Dependence Graph. The EASDG of an aspect-oriented program consists of a System Dependence Graph (SDG) of non-aspect code and an Aspect Dependence Graph (ADG) for aspect code and some additional dependence edges. The ADG is constructed by combining the *advice dependence graph, introduction dependence graph, pointcut dependence graph and method dependence graph.* A special vertex called as aspect entry vertex used in ADG representing the entry point into the aspect. All the members of aspect are connected through *aspect membership edges* to the aspect entry vertex. The complete EASDG is constructed by connecting the SDG and ADG by identifying the weaving vertices using a special kind of edge called as weaving edges. This EASDG is constructed statically only once before the execution of the aspect-oriented program. During the execution of program this algorithm marks and unmarks the executed nodes to calculate the dynamic slice of aspect-oriented programs.

Mohapatra et al. [26] proposed a dynamic slicing algorithm for aspect-oriented program called *Trace Based Dynamic Slice* (TBDS) algorithm. This algorithm uses a dependence-based representation called *Dynamic Aspect-oriented Dependence Graph* (DADG) as the intermediate representation of program. The DADG is an *arc-classified digraph* where all the vertices of the graph correspond to the statements and predicates of the program and all the edges (arcs) between the vertices represents dependence relationship between the statements. A DADG consists of *control dependence arc, data dependence arc* and *weaving arc* as the dependence relationship arcs. The construction of DADG of an aspect-oriented program is based on the analysis of control and data flow of the program at run time. Then they had used breadth-first or depth-first order graph traversal over DADG to compute the dynamic slice with respect to the statement of interest *(as defined in slicing criterion)* as the starting point of traversal.

Ray et al. [29] proposed a dynamic slicing algorithm for aspect-oriented programs by marking and unmarking the edges. Firstly, they had constructed an intermediate representation called Aspect System Dependence Graph (AOSG). AOSG was constructed by combining System Dependence Graph (SDG) of non-aspect code and Aspect Dependence Graph (ADG) of the aspect code with the help of aspect-membership arcs. Then, they compute the dynamic slice by updating AOSG.

All the above mentioned works were concentrated on computing the slice by considering the raw code of aspect-oriented programs. They have not considered slicing of aspect-oriented architectural models. But, our algorithm computes the dynamic slice of aspect-oriented software at architectural level.

Lallchandani et al. [20] 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 intermediate representation as well as in the proposed slicing algorithms. In our work, we have considered the AOP constructs such as pointcuts, advices, joinpoints, etc. both in the intermediate representation and the slicing algorithm.

CONCLUSION

In this paper, we have proposed a novel dynamic slicing algorithm for aspect-oriented based UML 2.0 sequence diagrams. We have considered the modeling of aspect-oriented programs using sequence diagrams. However, our work can be easily extended to handle other UML 2.0 diagrams and models. We have used the Aspect Model Dependency Graph (AMDG) as the intermediate program representation. Our proposed Architectural Aspect-Oriented Dynamic Slicing (AAODS) algorithm is based on marking and unmarking the edges of the AMDG as and when the dependencies arise and cease at run-time. The computed slices can be used for studying the impact of design changes, reliability prediction, understanding large architectures, regression testing, etc. The advantage of our approach is that when a request for a slice is made, it is already available and it can be readily obtained through a mere table looks up. We are now extending the intermediate model to support both class and sequence diagrams of aspect-oriented UML models.

REFERENCES

1. JBoss Aspect Oriented Programming. http://labs.jboss.com/portal/jbossaop.
2. Spring framework. http://www.springframework.org/docs/reference/aop.html.
3. Agrawal, H. and Horgan, J. (1990). Dynamic program slicing. SIGPLAN Not., 25(6), pp.246-256.
4. Aldawud, O., Elrad, T. and Bader, A. (2001), A UML profile for AOP. In OOPSLA, workshop on Aspect Oriented Programming.
5. Binkely D, and K. B. Gallagher (1996). Program Slicing. Technical report, Academic Press,San Diego,CA.
6. Bustos, A. and Eterovic, Y. (2007) Modeling aspects with UML'S class, sequence and state diagrams in an industrial setting. In IASTED International Conference on Software Engineering and Applications, pages 403-410. ACTA Press.
7. Colyer, A. Clement, A., Harley, G. and Webster, M. (2004), Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools. Addison Wesley Professional.

8. Georgieva, K. (2009), Testing Methods and Approaches in Aspect-Oriented Programs. Master's thesis, Department of Computer Science, University of Magdeburg, Germany.
9. Gradecki, J. and Lesiecki, N. (2003), Mastering AspectJ. Wiley Publishing, Indianapolis, Indiana.
10. Gupta, P., Garg, S. and Khalon, K. (2011), Designing aspects using various UML diagrams in resource-pool management. International Journal Of Advanced Engineering Sciences And Technologies (IJAEST), 7(2), pp. 228-233.
11. Horwitz, S., Prins, J. and Reps. T. (1989), Integrating noninterfering versions of programs. ACM Transactions on Programming Languages and Systems, 11(3).
12. Horwitz, S., Reps, T. and Binkley. D. (1990), Interprocedural Slicing Using Dependence Graphs. ACM Transaction on Programming Languages and Systems, 12(1), pp. 26-61.
13. Kiczales, G. and Mezini, M. (2005), Aspect-Oriented Programming and Modular Reasoning. In Proceedings of ICSE.
14. Kiselev, I. (2003), Aspect-Oriented Programming with AspectJ. Sams Publishing.
15. Korel, B. and Ferguson, R. (1992). Dynamic slicing of distributed programs. Applied Mathematics and Computer Science 2.
16. Korel B. and Laski, J. (1988), Dynamic Program Slicing. Information Processing Letters, 29(9), pp. 155-163.
17. Korel B. and Laski, J. (1990), Dynamic slicing of computer programs. Journal of Systems and Software, 13(3), pp. 187-195.
18. Krupa. A. (2010), Analyze Aspect-Oriented Software Approach and Its Application. Master's thesis, Department Of Information Systems And Programs, Athabasca, Alberta.
19. Laddad, R (2003), AspectJ in Action. Manning Publications Co.
20. Lallchandani, J. and Mall, R. (2011). A dynamic slicing technique for UML architectural models. IEEE Transactions on Software Engineering, 37(6).
21. Larsen, L. and Harrold, M. (1996), Slicing object-oriented software. In Proceedings of 18th International Conference on Software Engineering, pages 495-505, 1996.
22. Mendhekar, A., Kiczales, G. and Lamping, R. (1997), A Case-Study for Aspect-Oriented Programming. Technical report, Xerox Palo Alto Research Centre.
23. Russell Miles. AspectJ Cookbook. O'Reilly, 1005 Gravenstein Highway North, Sebastopol, December 2004.
24. D. P. Mohapatra, R. Kumar, and R. Mall. Computing dynamic slices of concurrent object-oriented programs. Information & Software Technology, 47(12):805-817, 2005.
25. D. P. Mohapatra, R. Kumar, and R. Mall. An Overview of Slicing Techniques for Object Oriented Programs. Informatica, 30(2):253-277, 2006.
26. D. P. Mohapatra, M. Sahu, R. Kumar, and R. Mall. Dynamic Slicing of Aspect-Oriented Programs. Informatica, 32(3):261-274, 2008.
27. Mohapatra, D. (2005), Dynamic Slicing of Object-Oriented Programs. PhD thesis, Department of Computer Science and Engineering, Indian Institute of Technology, Kharagpur.

28. Mohapatra, D., Mall, R., and Kumar. R. (2004), An edge marking technique for dynamic slicing of object-oriented programs. In COMPSAC, pp.60-65.
29. Ray, A., Mishra, S. and Mohapatra, D (2012), A Novel Approach for Computing Dynamic Slices of Aspect-Oriented Programs. International Journal of Computer Information Systems, 4(9), pp. 6-12.
30. Rumbaugh, J., Jacobson, I. and Booch, G (1998). The Unified Modeling Language Reference Manual. ADDISON-WESLEY.
31. Sahu, M. and Mohapatra, D. (2007). A Node-Marking Technique for Dynamic Slicing of Aspect-Oriented Programs. In Proceedings of 10th International Conference On Information Technology, pp.155-160.
32. Steindl, C. (1999). Program Slicing for Object-Oriented Programming Languages. PhD thesis, Johannes Kepler University Linz.
33. ter Braak, T. (2006). Extending Program Slicing in Aspect-Oriented Programming with Inter-Type Declarations. In Proceedings of 5th Twente Student Conference on IT.
34. Tip, F. (1995). A Survey of Program Slicing Techniques. Journal of Programming Languages, 3:121-189.
35. Weiser, M. (1979), Program slices: formal, psychological, and practical investigations of an automatic program abstraction method. PhD thesis, University of Michigan, Ann Arbor, MI.
36. Weiser, M. (1981), Program slicing. In Proceedings of the 5th International Conference on Software Engineering, ICSE, pages 439-449, Piscataway, NJ, USA. IEEE Press.
37. Mark Weiser. Programmers Use Slices with Debugging. Communications of the ACM, pages 446-452, July 1982.
38. B. Xu and Z. Chen. Dynamic Slicing Object-Oriented Programs for Debugging. In Proceedings of SCAM 2002, pages 115-122, 2002.
39. J. Zhao. Dynamic Slicing of Object-Oriented Programs. Technical report, Information Processing Society of Japan, 1998.
40. J. Zhao. Slicing Aspect-Oriented Software. In Proceedings of 10th International Workshop On Program Slicing, pages 251-260, June 2002.
41. J. Zhao and M. Rinard. System Dependence Graph Construction for Aspect-Oriented Programs. Technical report, Laboratory Of Computer Science, Massachusetts institute of Technology, USA,March 2003.

Author's Profiles

ABHISHEK RAY is an Associate Professor with School Of Computer Engineering, KIIT University, Bhubaneswar. He can be contacted at ar_mmclub@yahoo.com.

SIBA MISHRA received his M.TECH from KIIT University, Bhubaneswar, Odisha, India. His research interests include Software Engineering, Automata Theory, and Discrete Mathematics. He can be contacted at sibamishra@yahoo.co.in.

DURGA PRASAD MOHAPATRA PhD is Associate Professor in the Department of CSE at the National Institute of Technology, Rourkela. He can be contacted at durga@nitrkl.ac.in.

For information on obtaining additional copies, reprinting or translating articles, and all other correspondence, please contact:

Email: InfosyslabsBriefings@infosys.com

© Infosys Limited, 2013

Infosys acknowledges the proprietary rights of the trademarks and product names of the other companies mentioned in this issue of Infosys Labs Briefings. The information provided in this document is intended for the sole use of the recipient and for educational purposes only. Infosys makes no express or implied warranties relating to the information contained in this document or to any derived results obtained by the recipient from the use of the information in the document. Infosys further does not guarantee the sequence, timeliness, accuracy or completeness of the information and will not be liable in any way to the recipient for any delays, inaccuracies, errors in, or omissions of, any of the information or in the transmission thereof, or for any damages arising there from. Opinions and forecasts constitute our judgment at the time of release and are subject to change without notice. This document does not contain information provided to us in confidence by our clients.

Infosys®

POWERED BY INTELLECT
DRIVEN BY VALUES