JOURNAL OF OBJECT TECHNOLOGY
Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2008

Vol. 8, No. 1, January–February 2009

Static Slicing of UML Architectural Models

Jaiprakash T. Lallchandani
R. Mall
Department of Computer Science & Engineering
Indian Institute of Technology Kharagpur
Kharagpur 721302 WB INDIA
{jtl,rajib}@cse.iitkgp.ernet.in

We propose a technique for static slicing of UML models. We first transform a software architecture specified using UML into an intermediate representation which we have named Model Dependency Graph(MDG). MDG combines information available in various sequence diagrams along with the relevant information available in class diagrams into an integrated UML model. For a given slicing criterion, our slicing algorithm traverses the constructed MDG to identify the relevant model elements. Our algorithm's novelty lies in its computing a slice based on an integrated UML model as against independently processing separate UML diagrams, and determining the implicit interdependencies among the different model elements distributed across various UML diagrams. We also briefly discuss a prototype tool named SSUAM(Static Slicer for UML Architectural Models) which we have developed to implement our algorithm.

1 INTRODUCTION

The architecture of an object-oriented software system defines its high-level design structure [12]. With the increase in size and complexity of software products, the importance of architectural design models has been increasing remarkably [2,11]. A few of the important uses of an architectural design model are evaluation, understanding, and testing a design solution. An architectural design model allows an architect to reason about various system properties at a higher level of abstraction. Of late, Unified Modeling Language(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.

Analysis of UML models is a challenge since the information about a system is distributed across several model views captured through a large number of diagrams. Analysis of the impact of a change to one model on other elements therefore becomes a non-trivial problem. For example, if a method returning a value changes, then several classes may get affected either due to the variable being used in guard

Cite this article as follows: Jaiprakash T. Lallchandani, R. Mall: *Static Slicing of UML Architectural Models*, in Journal of Object Technology, vol. 8, no. 1, January–February 2009, pages 159–188,
http://www.jot.fm/issues/issues_2009_01/article2

conditions in various sequence diagrams or because of method calls.

With increase in product sizes and complexities, UML models themselves tend to become large and complex and may involve thousands of interactions across hundreds of objects. For such large architectures, it becomes exceedingly difficult to understand and analyze these models. Moreover, it becomes tedious on one hand and equally valuable on the other to determine the impact of a certain change to one model element on other model elements. For large architectures, determining the impact of a change would require taking into account the various types of dependencies that might exist among different model elements. Development of an impact analysis technique can help understanding, testing, reengineering, maintenance, and reuse of large software architectures.

In this context, researchers have proposed to use program slicing techniques to decompose large architectures into manageable portions. This can not only facilitate comprehending large architectures, but also help perform impact analysis and reliability prediction based on architectural models [7, 22, 24]. Besides, architecture model slicing can also be used to compute various types of metrics to characterize software architectures.

In the context of software architectures, a slicing technique should take into account various use cases, classes and their relationships, and objects and their interactions. UML class diagrams describe various relations among classes such as aggregation, association, composition, and generalization / specialization. On the other hand, sequence diagrams depict a sequence of actions through which a use case is realized [3]. Traditional slicing is usually performed solely based on data and control dependency relationships existing among program statements. However, to perform architecture slicing, it is necessary to first formulate an appropriate intermediate representation that can represent different types of dependence relationships that may exist among classes, sub-classes, methods, and attributes, and call sequences.

We propose an intermediate representation for software architecture by integrating various UML diagrams into a single system model. We have named this representation Model Dependency Graph(MDG). To construct an MDG, we first construct an intermediate representation for classes called Class Dependency Graph(CDG) for every UML class diagram. We also construct another intermediate representation called Sequence Dependency Graph(SDG) for every UML sequence diagram in the system. Next, we integrate the CDGs and SDGs to construct an MDG.

We have named our proposed algorithm Architectural Model Slicing through MDG Traversal(AMSMT). Our slicing algorithm is based on traversing [9, 15, 16] the edges in the MDG for any given slicing criterion. Through MDG traversal, AMSMT identifies the relevant model elements from an architecture based on the dependencies among them to compute a static architectural model slice. A novelty of AMSMT is its computation of a slice based on an integrated UML model. This is in contrast to the related work where slicing is based on individual UML diagrams, and

finding the implicit interdependencies among different model elements distributed across various model diagrams.

The rest of this paper is organized as follows. The next section reviews related work. Section 3 presents overview of some UML 2.0 concepts relevant to our discussion. Section 4 presents an integrated UML model and a few basic definitions that have been used in our algorithm. Our intermediate representation MDG is discussed in Section 5. Section 6 presents our algorithm for computing a static architectural model slice along with an example to illustrate the working of our algorithm. Section 7 compares our work with related work and Section 8 concludes the paper.

2 RELATED WORK

In this section, we briefly review the reported work on architectural and model slicing. Most of the work reported in the literature address development of techniques based on slicing the architectural description of a system given in different architecture description languages(ADLs) such as Aesop, C2, Darwin, Meta-H, Rapide, UniCon, and Wright [1]. However, the research work on slicing UML architectural models have scarcely been reported in the literature, though UML is being used as an ADL [13, 14].

Zhao [25] investigated a novel dependence analysis technique, called architecture dependence analysis, to support software architecture development. Though his work considered Acme ADL, this approach to architecture dependence analysis was shown to be independent of any specific ADL.

Zhao introduced a static architecture slicing technique [21], which operates by removing unrelated components and connectors, and ensures that the behavior of a sliced system remains unaltered. The work reported by Zhao in [23] is an extension of his earlier work reported in [21, 25] and is based on Wright ADL [1].

Kim introduced an architectural slicing technique called dynamic software architecture slicing(DSAS) in [7]. Kim's work is able to generate a smaller number of components and connectors in each slice as compared to [21]. This is especially true in situations where a large number of ports are present and their invocation can change the values of some variables, or the occurrence of certain events.

Korel *et al.* [8] present an approach to slicing state-based models, such as EFSMs (Extended Finite State Machines). They present two types of slicing - deterministic and non-deterministic slicing. Their approach also includes a slice reduction technique to reduce the size of a computed EFSM slice. Korel *et al.* [8] also report a tool that implements their slicing technique for EFSM models.

Kagdi *et al.* [4] introduce the concept of model slicing as a means to support maintenance through the understanding, querying, and analyzing large UML models. Kagdi *et al.*construct model slices from UML class models. Their slicing approach extracts parts of a class diagram in order to construct sub-models from a

given model of a system. However, class models are devoid of explicit behavioral information and depict only structural behavior.

3 RELEVANT UML 2.0 CONCEPTS

In this section, we present an overview of those aspects of UML 2.0 that are relevant to our work. We first discuss certain aspects of class diagrams and then discuss sequence diagrams.

Class Diagrams

A class diagram shows the different classes of a system, their inter-relationships, the various operations and attributes of the classes. Moreover, a system model may comprise more than one class diagram. We identify each of the class diagrams by assigning an arbitrary unique label to it. For example, Consider an example system model consisting of three class diagrams. These can be labeled as CD_1, CD_2, and CD_3 respectively.

Figure 1(a) shows an example class diagram. It shows that any object of `Composite- Class` is composed of one or more `ParentClass` instances. The association relationship is depicted using a line connecting two classes `ChildClass2` and `CompositeClass`, indicating that `instance_cc` will become an instance variable in `ChildClass2`. `Child- Class1` and `ChildClass2` are derived from base class `ParentClass` through inheritance. A note containing the description "uses pc_1" has been shown anchored to method `pc_method1()` in `ParentClass`. In the rest of the paper, notes would be used to represent the information related to the usage of data in the system being modeled.

In UML the attached notes may contain any text, and have no defined semantics. To be able to automatically process the information present in attached notes, we need to define certain rules and a suitable semantics for the UML notes. This can help to identify relevant data dependencies from the text present in the notes. We define a simple rule for writing notes. The text in the notes can either start with a word "`uses`", or "`calls`", and the words in the text followed by "`uses`" and "`calls`" are separated by a comma. We use notes to represent the following types of information: (i) A method may use one or more class attributes. (ii) A method may contain calls to one or more class methods.

We now define the semantics of the notes anchored to a method in a UML class diagram.

1. If the text string representing a note begins with the word "`uses`", then the rest of the words in the text string represent the class attributes on which the method is data dependent.

2. If the text string representing a note begins with the word "`calls`", then the rest of the words in the text string represent the methods that are called by the method anchored with the note.

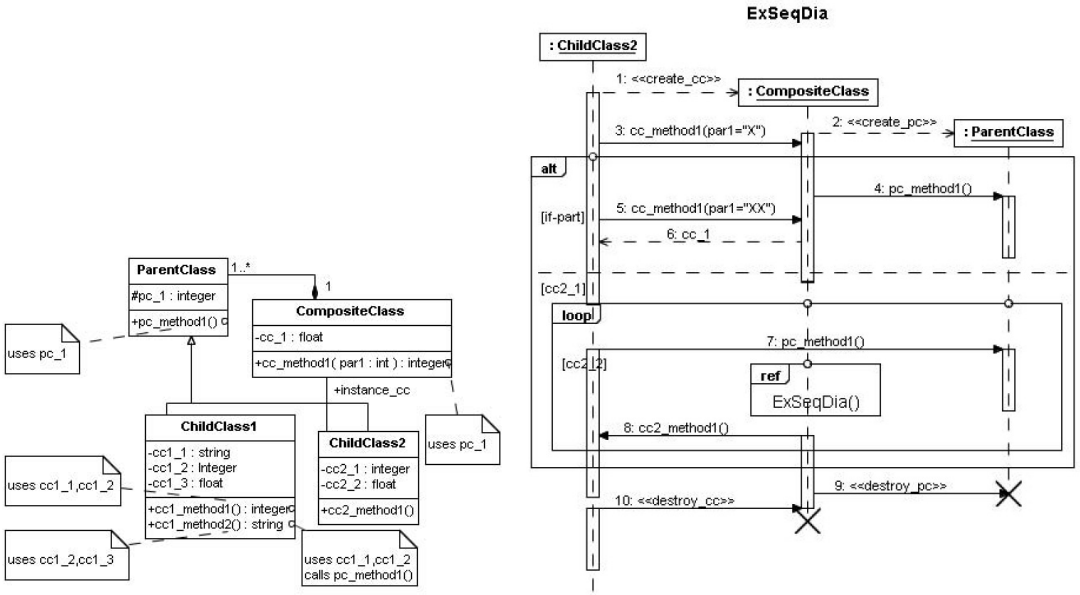


Figure 1: (a) A generic class diagram (b) A generic sequence diagram

Sequence Diagrams

An interaction represents a communication between two objects. In UML 2.0, the details of an interaction can be modeled using sequence diagrams, interaction overview diagrams, communication diagrams, timing diagrams and interaction tables.

A sequence diagram shows how objects interact with each other to achieve a behavioral goal. Unlike a communication diagram, a sequence diagram shows the sequence of messages exchanged among the objects on a lifeline. A lifeline in a sequence diagram shows time as a dimension to represent the order in which interactions occur.

An example of a UML sequence diagram is shown in Figure 1(b). The rectangles in this diagram depict objects (or lifelines). The object names have been formed by prefixing their class names with a colon and underlined to show that it applies to any object of the class. The arrows connecting the objects represent messages; and are labeled with their names, sequence numbers, and arguments. The name of a message corresponds to the name of a member function of the receiving class. The sequence diagram of Figure 1(b) depicts the use of the combined fragments `alt` and `loop`. It shows that the messages numbered `4,5` and `6` form the if-part, and messages numbered `7` and `8` along with a loop containing an interaction occurrence

labeled ExSeqDia in the lifeline of the anonymous object of CompositeClass are part of the else part.

We have shown one example sequence diagram. However, a system model may consist of many sequence diagrams. In such systems, we identify each sequence diagram by an arbitrary label assigned to it. Also, once a sequence diagram is assigned a label, it is not changed. Moreover, no two sequence diagrams can be assigned the same label. As an example, if we consider a system model consisting of four sequence diagrams, they may be labeled as $SD_1, SD_2, SD_3,$ and SD_4 respectively.

4 DEFINITION OF AN INTEGRATED UML MODEL

In this section, we present an overview of an integrated UML model which we have named Model Dependency Graph(MDG). We first discuss the key elements of MDG, and then outline the representation of a generic system using MDG. A more detailed discussion on the relationships among different elements of MDG is given in the next section.

A class diagram model of an MDG has been shown in Figure 2. It gives an overview of the elements involved in the structural design of an integrated UML model. Each instance of an MDG is composed of one or more SDG instances along with one or more associated CDG instances. In the later subsections, CDG and SDG have been discussed in more detail along with an example. An instance of a CDG or an SDG is composed of one or more node types along with one or more edges representing different dependence types as shown in the class diagram of Figure 2. We discuss how CDGs and SDGs are integrated to realize an MDG in the next section.

In the following, we define the different elements of an MDG.

- A *class access*(CA) node is defined for every class in the UML architectural model.
- A *method access*(MA) node is defined for every method of a class.
- An *attribute*(AT) node is defined for every class attribute.
- A *parameter*(PR) node is defined for every method parameter specified in a method signature.
- A *return*(RT) node is defined for every return parameter specified in a method signature.
- A *predicate class*(PC) node is defined for every combined fragment used in a sequence diagram.
- An *interaction*(IT) node is defined for every interaction occurrence used in a sequence diagram.

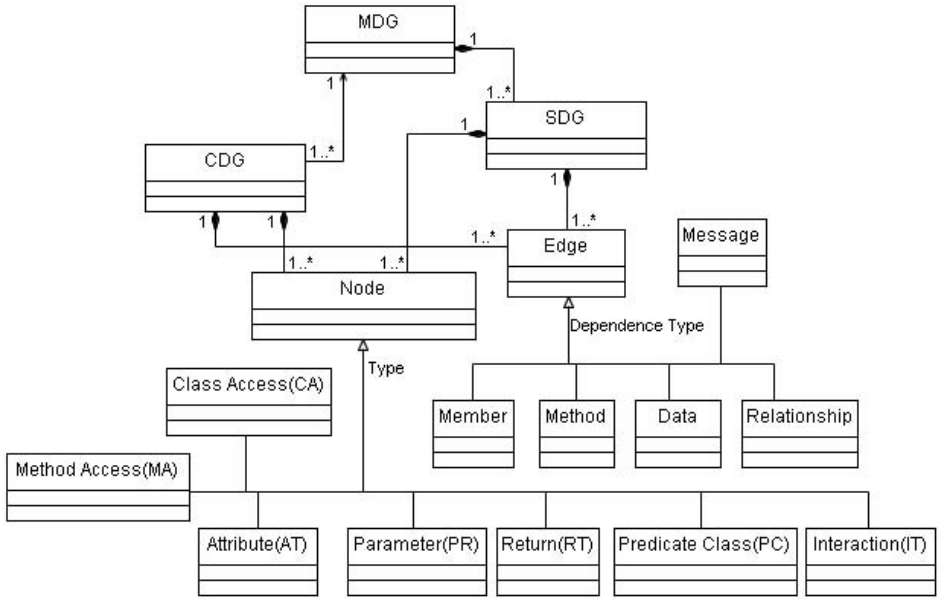


Figure 2: Class diagram for the integrated UML model MDG

A *class access*(CA) node represents an entry point to reference all the attributes and methods of a class. A *method access*(MA) node represents an entry to a class method. An *attribute*(AT) node represents an access to the corresponding class attribute. A CA node is connected to every MA node of a class, and to every AT node of the class by dependence edges. The formal parameters of a class method are represented using *parameter*(PR) nodes while the return values are represented using *return*(RT) nodes. An MA node is connected to PR and RT nodes by dependence edges. Also, an RT node may be connected to an AT node by a dependence edge.

We now discuss how a model of a generic system can be represented using the different elements of an MDG. Figure 3 presents a generic system defined in terms of classes, and its representation in the form of an MDG. The generic system consists of n classes CL(1)...CL(n), and every class has i attributes and m methods. The MDG of the system can be described as follows:

- The classes are represented by *class access*(CA) nodes $CA_{CL(1)} \ldots CA_{CL(n)}$.
- The attributes of every class CL(n) are represented by *attribute*(AT) nodes $AT_1 \ldots AT_i$.
- The methods of every class CL(n) are represented by *method access*(MA) nodes $MA_1 \ldots MA_m$.
- The k method parameters 1...k are represented by *parameter*(PR) nodes $PR_1 \ldots PR_k$.
- The return parameters are represented by *return*(RT) nodes $RT_1 \ldots RT_m$.

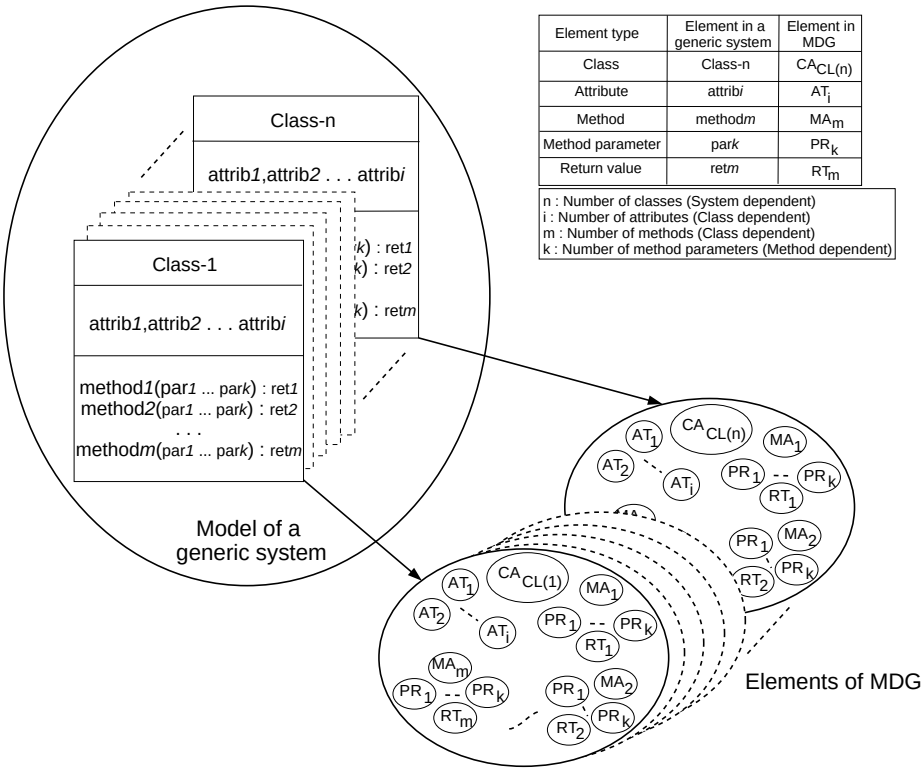


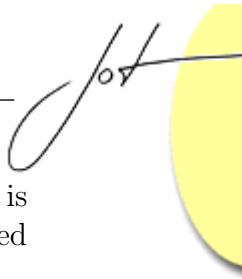
Figure 3: MDG representation of a generic system

The nodes $\texttt{AT}_1 \ldots \texttt{AT}_\texttt{i}$ and $\texttt{MA}_1 \ldots \texttt{MA}_\texttt{m}$ represent the attributes and methods of the class CL(n), and all the respective AT and MA nodes are connected to the class access(CA) node $\texttt{CA}_{\texttt{CL(n)}}$ using dependence edges. Similarly, for the nodes $\texttt{PR}_1 \ldots \texttt{PR}_\texttt{k}$ and $\texttt{RT}_1 \ldots \texttt{RT}_\texttt{m}$ representing the parameters and return values of every method m, all the related PR and RT nodes are connected to the method access node $\texttt{MA}_\texttt{m}$ through dependence edges.

For the model of the generic system in Figure 3, we consider the following view pertaining to a software architecture constructed from various UML models.

A software architecture is a collection of system entities, such as use cases, classes, objects, interactions and scenarios connected to form a system structure. It represents both the structural as well as the behavioral views of a system. To represent such UML model of a system as an integrated model MDG, we begin with a discussion on CDG representation in the next subsection. The intermediate architectural model called CDG for any software system is based on a set of classes given in the form of UML class diagrams. The CDG represents the different elements of a class diagram along with the relationships among these elements using dependence edges. The information present in the CDG is later used to transform a software architecture into a MDG representation which we discuss in Section 5.

In order to construct the CDG, the classes and all inter-dependencies among



these classes are identified from the class diagrams. We assume that a note is anchored to the methods of a class in a class diagram, capturing the attributes used by different class methods along with the related method calls.

Class Dependency Graph(CDG)

In this subsection, we define our CDG representation of UML 2.0 class diagrams. A CDG represents a set of classes and their relationships. In a CDG, classes and their attributes, methods and their call parameters, together with method return values are represented as different types of nodes. Several types of dependencies might exist among the nodes. These would be represented by using appropriate dependence edges in the CDG. Member dependence edges represent the class memberships of methods and attributes, while method dependence edges represent the dependence of the call parameters and return values(if any) on a method. Data dependence edges represent flow of data among statements of a class method. Data dependencies arise when the class methods, its parameters and return values directly or indirectly make use of the class attributes. In addition, data dependence edges also represent the effect of the calling parameters on the return value of a method. Relationship dependence edges represent how a class is related to another class, or how an instance of a class is related to the other class instances. It is to be noted that the CDG intends to represent each class along with its features such that the dependencies among them get easily identified in the process of slice computation. At the same time, CDG does not represent class relations in any different way compared to its corresponding class diagram. For example, the arrow style for representing generalization, aggregation, and composition relations using the relationship dependence edges in the CDG exactly match the UML notation used to represent these class relationships. However, the association relation is not represented in the CDG from its class diagram. An association relation represents classes which are likely to communicate with other classes due to a method invocation. The dependency arising out of such method invocations are already handled from the sequence diagram and represented using SDG to be discussed in the later subsection.

A CDG contains one class access(CA) node for every class in the class diagram. The CDG contains one attribute(AT) node for every distinct class attribute, and one method access(MA) node for every single method of a class. Depending on the number of attributes and methods for a class, the AT and MA nodes are named by appending a numeric subscript to the node symbol (AT or MA) that reflects the label assigned to the class attributes and methods. To model parameter passing to class methods, each MA node is associated with one or more parameter(PR) nodes. For every method returning a value, the return value is represented by associating a return(RT) node with the corresponding MA node. PR and RT nodes are associated with the MA nodes using the method dependence edges, while the MA and AT nodes are associated with the CA nodes using member dependence edges.

A CDG captures both the static as well as dynamic dependencies in a system.

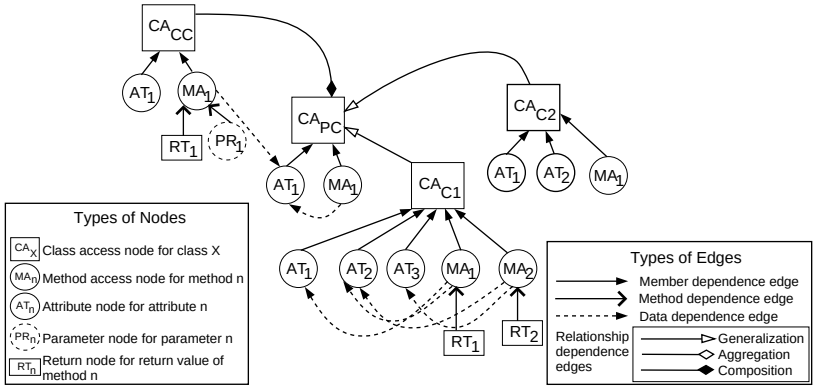


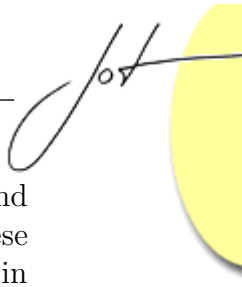
Figure 4: CDG for the example class diagram of Figure 1(a)

The static dependencies do not vary with time, viz. member dependencies, method dependencies and relationship dependencies. The dynamic dependencies change with time, and all the data dependencies fall under this category. The dependencies mentioned for the CDG are present in UML class diagrams in the form of dependence among classes, and its features. Figure 4 shows the CDG for the example class diagram of Figure 1(a).

We have already mentioned that each class diagram in the system is assigned a fixed label. The label assigned to a class diagram gets associated to the corresponding CDG during the process of its construction. If we consider three class diagrams and identify them as $\texttt{CD}_1, \texttt{CD}_2,$ and $\texttt{CD}_3$, then their corresponding CDGs will be identified as $\texttt{CDG}_1, \texttt{CDG}_2,$ and $\texttt{CDG}_3$ respectively.

An Example CDG

The class diagram of Figure 1(a) contains four classes `ParentClass`, `CompositeClass`, `ChildClass1` and `ChildClass2`. Each of these classes is represented by using class access nodes $\texttt{CA}_{\texttt{PC}}, \texttt{CA}_{\texttt{CC}}, \texttt{CA}_{\texttt{C1}}$ and $\texttt{CA}_{\texttt{C2}}$ respectively as shown in the CDG of Figure 4. To understand how other nodes are added to the CDG, let us consider `ChildClass1` of the example class diagram of Figure 1(a). It comprises of three attributes and two methods which form the features of `ChildClass1`. The attributes are represented by the attribute nodes $\texttt{AT}_1, \texttt{AT}_2,$ and $\texttt{AT}_3$, and the methods are represented by the method access nodes $\texttt{MA}_1$ and $\texttt{MA}_2$ in the CDG of Figure 4. All these attribute and method access nodes are connected to the class access node $\texttt{CA}_{\texttt{C1}}$ using member dependence edges. It can be observed that each method of `ChildClass1` has a note anchored to it. The note associated with method `cc_method1()` captures the data dependency on class attributes `cc1_1` and `cc1_2` used within the method body. Moreover, several class attributes may have been used to compute a method's return value. Each method has a return value which is represented in the CDG using the return nodes $\texttt{RT}_1$ and $\texttt{RT}_2$. The return nodes are connected to the `MA` nodes using method dependence edges. The class diagram of Figure 1(a)



shows that ParentClass has a composition relationship with CompositeClass, and also classes ChildClass1 and ChildClass2 are derived from ParentClass. These class relationships are captured using the relationship dependence edges shown in the CDG of Figure 4.

Some SDG Concepts

We now briefly present our terminology pertaining to *predicate class*(PC) and *interaction*(IT) nodes. In the context of a UML 2.0 model, if a *combined fragment* has a condition associated with it in the lifeline of an object of a class CL(n) in the sequence diagram, then the corresponding *class access*(CA) node $CA_{CL(n)}$ in the SDG of the sequence diagram is represented by a *predicate class*(PC) node.

The example sequence diagram of Figure 1(b) shows two combined fragments alt and loop. The condition associated with the alt combined fragment uses the class attribute cc2_1 while the loop combined fragment uses the class attribute cc2_2 in the lifeline of the anonymous object of ChildClass2. This adds a *predicate class*(PC) node CA_{C2} as depicted in the SDG of Figure 5.

In the context of a UML 2.0 model, if the lifeline of an object of a class CL(n) in the sequence diagram contains an *interaction occurrence*, then this reference to any sequence diagram using an *interaction occurrence* is represented as an *interaction*(IT) node in the SDG of the sequence diagram.

In the example sequence diagram of Figure 1(b), a recursive reference to the same sequence diagram is made in the lifeline of an anonymous object of CompositeClass using an *interaction occurrence* represented as a ref item in the sequence diagram. This adds an *interaction*(IT) node in the SDG of Figure 5. The sequence diagram of Figure 1(b) is named ExSeqDia which is seen from the label assigned to it on the top. The same label is associated with the ref item of the sequence diagram. As only one *interaction occurrence* is used in this sequence diagram, the SDG of Figure 5 depicts an IT node that is labeled IT_1 representing the reused sequence diagram of Figure 1(b).

For slicing the UML architectural models, we need to convert the different UML sequence diagrams into intermediate representations which we have named Sequence Dependency Graphs(SDGs). An SDG is used to represent the message exchange among different objects in a sequence diagram using dependence edges. The information present in an SDG along with a CDG is later used to transform a software architecture into an MDG representation.

Sequence Dependency Graph(SDG)

In this subsection, we define our SDG representation for UML 2.0 sequence diagrams. Each sequence diagram of a system is represented as an SDG. An SDG can depict various scenarios represented in the corresponding sequence diagram. An

SDG can be considered as a dependence graph representing a set of objects and their interactions. Moreover, in an SDG the nodes are either classes or other sequence diagrams of the system. However, when an SDG contains a node which represents another sequence diagram, and that sequence diagram itself is represented as an SDG, then an SDG becomes a hierarchical diagram in some sense. This helps to simplify the SDG, whereby a reused sequence diagram(also represented as an SDG) is represented by a single node termed as an *interaction*(IT) node in the SDG. The concept of an IT node prevents the use of a sequence diagram recursively, and restricts the number of SDG nodes to become unbounded. In fact, when an SDG contains one or more other SDGs, no additional nodes are added from other SDGs to represent the reused sequence diagrams.

Message dependence edges represent flow of messages among objects. Each message dependence edge is annotated with a label. The k^{th} message in a sequence diagram is represented by I(k) in its SDG. Different types of message dependencies may exist among the objects. These are represented by different types of dependence edges in the SDG. If a message transfer from an object of one class to an object of another class involves a method call(implicit or explicit), we represent this message dependency by a *call dependence* edge. If a message transfer is a self-call (a message from an object to one of its own methods), the corresponding message dependency is also represented by a *call dependence* edge. When an object instantiates another object, it is also represented as a *call dependence* edge, because it involves an implicit call to the constructor method of the class. On the other hand, a message transfer representing a value return on account of a method invocation by an object is a message dependency represented as a *return dependence* edge. In addition, an interaction between any two objects can involve the use of public attributes of related system classes. This may arise due to a method invocation by an object of the involved pair of classes. Moreover, the public attributes used by the class pair hold a definition-use relationship between them. This possibility arises when a class attribute with a public scope is used by an object of one class, but has its definition in another related class. The SDG represents such a message dependency as an *attribute dependence* edge. Moreover, in case of an occurrence of a *predicate class*(PC) node, all the messages in the *combined fragment* form a message dependency, and are represented as a *control dependence* edge in the SDG. However, to represent a dependence relation between an IT node and other SDG nodes, another type of message dependency is used and it is termed as an *inter-sequence* dependency. *Inter-sequence dependence* edges represent a message dependency of a sequence diagram on another sequence diagram.

An SDG contains a class access(CA) node for every class whose object is used in the sequence diagram, and additionally may contain interaction(IT) nodes to represent references to other sequence diagrams. Having identified all the dependencies that may exist in a sequence diagram, we can represent those in the corresponding SDG.

An SDG captures various static and dynamic dependencies that may exist in a

system. In the SDG, the only static dependencies are due to control dependencies. The dynamic dependencies change with time, and they include call dependencies, return dependencies, inter-sequence dependencies, and attribute dependencies. Figure 5 shows the SDG for the example sequence diagram of Figure 1(b).

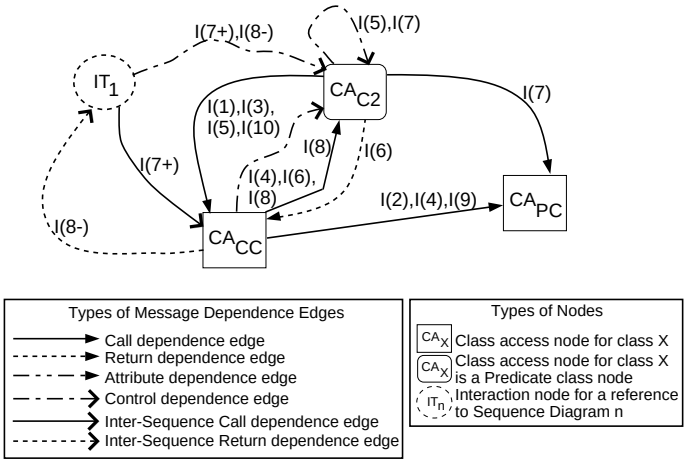


Figure 5: SDG for the example sequence diagram of Figure 1(b)

In our discussion on sequence diagrams, we mentioned that each sequence diagram of the system is identified by a unique label. In the process of constructing an SDG from its corresponding sequence diagram, the label associated with a sequence diagram is used to label the corresponding SDG. If we consider four sequence diagrams and label them as $SD_1, SD_2, SD_3,$ and SD_4, then their corresponding SDGs will be labeled as $SDG_1, SDG_2, SDG_3,$ and SDG_4 respectively.

An Example SDG

Consider the sequence diagram shown in Figure 1(b). The sequence diagram uses objects of classes `ChildClass2`,`ParentClass`, and `CompositeClass`. These classes are represented using the class access nodes CA_{C2}, CA_{PC} and CA_{CC} respectively as shown in the SDG of Figure 5. The sequence diagram of Figure 1(b) contains two combined fragments - `alt` and `loop`. Both the combined fragments are associated with the lifeline of `ChildClass2` object. All the message transfers which are part of the combined fragment are controlled by the conditions of the combined fragment. And that object containing the combined fragment as a part of its lifeline forms a *predicate class*(`PC`) node. The SDG of Figure 5 shows a predicate class node CA_{C2}. In the same sequence diagram, it recursively refers to itself using the *interaction occurrence* `ExSeqDia` in the lifeline of `CompositeClass` object, where `ExSeqDia` is the label assigned to the referred sequence diagram. This is represented in the SDG of Figure 5 as an *interaction*(`IT`) node labeled as IT_1.

In the sequence diagram of Figure 1(b), the first message transfer creates an object of type `CompositeClass`. This message transfer involves an implicit call

to the constructor method of CompositeClass. The same is depicted using a call dependence edge (CA_{C2}, CA_{CC}) in the SDG of Figure 5. This message transfer has an integer k (here k=1) associated with it in the sequence diagram of Figure 1(b). Hence, we represent this call dependence edge with the label I(1). The other message transfers are labeled I(2),I(3),I(4),I(5),I(7),I(8),I(9), and I(10), and each represents a call dependence edge in the SDG of Figure 5. A method cc_method1() associated with the message transfer numbered (k+4) returns a value using the message (k+5). This is depicted using the return dependence edge (CA_{C2}, CA_{CC}) annotated with label I(k+5)[as k=1, I(k+5)=I(6)] in the SDG. As mentioned earlier, node CA_{C2} is a *predicate class*(PC) node. The message transfers numbered (k+3),(k+4),(k+5),(k+6), and (k+7) are part of the alt combined fragment associated with object of ChildClass2, while (k+6) and (k+7) are also part of the combined fragment loop associated with the same object. This adds a control dependence edge (CA_{CC}, CA_{C2}) annotated with labels I(4),I(6),I(8), and edge (IT_1, CA_{C2}) annotated with labels I(7+),I(8-) in the SDG of Figure 5. The SDG of Figure 5 contains an *interactive*(IT) node IT_1. This is because object of CompositeClass refers to the sequence diagram ExSeqDia in its lifeline. This reference is after completion of message transfer numbered (k+6), and before the initiation of message transfer (k+7). This is depicted by an inter-sequence call dependence edge (IT_1, CA_{CC}) annotated with label I(7+), and an inter-sequence return dependence edge (CA_{CC}, IT_1) annotated with label I(8-) in the SDG of Figure 5.

5 MODEL DEPENDENCY GRAPH(MDG)

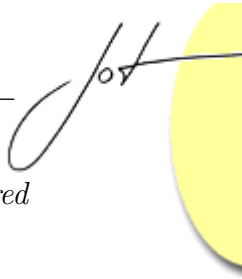
MDG represents both structural aspects modeled in various class diagrams as well as behavioral aspects modeled in sequence diagrams of an architecture. The process of constructing an MDG involves combining the nodes and edges of various SDGs along with the information present in different CDGs of a system. This process is termed as the *integration* process and is discussed in the next subsection. An MDG provides an integrated view of all system scenarios.

An MDG can contain all the different types of nodes that exist in the CDGs, and the SDGs. The CDG and SDG nodes used to construct the MDG have exactly the same representation as those in CDG and SDG.

Every message dependence represented in an SDG has a pair of object classes, attribute(s), method(s) and its parameters and return values associated with it. All such CA, AT, MA, PR and RT nodes that represent these object classes, attribute(s), method(s) and its parameters and return values are added from a CDG to the MDG. An IT node is added to the MDG, in case an IT node already exists with a SDG. Similarly, a PC node is added to the MDG when an SDG possesses a PC node.

The different nodes in an MDG are interconnected using various types of dependence edges as follows:

1. Edges of the MDG that existed either in a CDG or an SDG, and are added



without any alteration. We represent all such edges in the form of an *unaltered edge*.

2. Edges that are added to the MDG after making some alteration to the edges that existed either in a CDG or an SDG. We represent all such edges in the form of an *altered edge*.

For any edge in the MDG, when either the source, or the destination, or both are changed, then the corresponding edge connecting the changed nodes is termed as an *altered edge*. The situations under which the representation of an edge is changed is explained in the next subsection. Any edge that is not an *altered edge* automatically becomes an *unaltered edge*.

Integration of CDGs and SDGs

We now briefly discuss how CDGs and SDGs can be integrated to construct an MDG. The process of integrating CDGs and SDGs has schematically been shown in Figure 6. The integration process is carried out over many steps. The exact number of steps may vary depending on the number of SDGs present in a given UML model. In `Step-1`, an arbitrarily selected SDG $\texttt{SDG}_\texttt{i}$ along with the information present in different CDGs is used to construct a partial MDG $\texttt{MDG}_1$. Next, the process carried out in `Step-1` is repeated during `Step-2`, but on another arbitrarily selected SDG $\texttt{SDG}_\texttt{j}$. This step also updates $\texttt{MDG}_1$ constructed in previous step, resulting in a partial MDG $\texttt{MDG}_2$. The same process has to be repeated till all the SDGs have been considered. For integrating a model with `n` SDGs, the `Step-n` will result in $\texttt{MDG}_\texttt{n}$. The $\texttt{MDG}_\texttt{n}$ constructed after `n` steps is the final MDG obtained at the end of integration.

We now explain how different dependencies are added to the MDG from the CDGs and the SDGs. The member, method, and data dependencies from the CDG, and the control and inter-sequence dependencies from the SDG comprise the *unaltered edge* set. Adding the dependencies represented by various edge types in the *unaltered edge* set requires no further explanation, as they are added directly to the MDG from the CDGs or SDGs without any change. But the edges added to the MDG from the *altered edge* set requires a mapping from a pair of nodes in the CDG or SDG to an appropriate pair of nodes in the MDG. The following cases arise when an edge from either a CDG or an SDG is represented as an *altered edge* in a MDG:

1. A call dependence edge $(\texttt{CA}_{\texttt{from\_class}}, \texttt{CA}_{\texttt{to\_class}})$ between two `CA` nodes in an SDG is altered and represented between $\texttt{CA}_{\texttt{from\_class}}$ and a `MA` node of $\texttt{CA}_{\texttt{to\_class}}$ in the MDG.

2. A return dependence edge $(\texttt{CA}_{\texttt{from\_class}}, \texttt{CA}_{\texttt{to\_class}})$ between two `CA` nodes in an SDG is altered and represented between $\texttt{CA}_{\texttt{from\_class}}$ and a `RT` node of $\texttt{CA}_{\texttt{to\_class}}$ in the MDG.

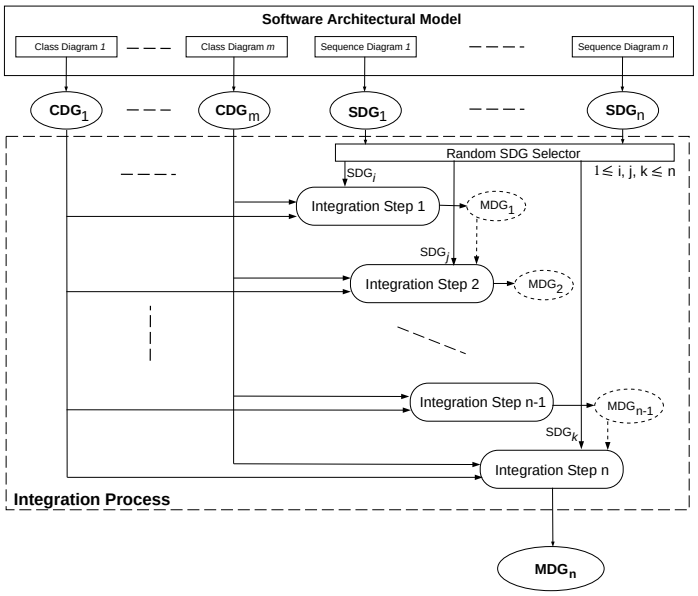


Figure 6: Integration of CDGs and SDGs into an MDG

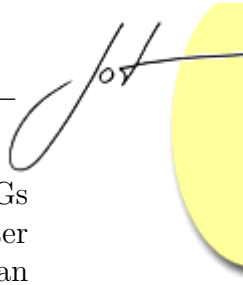
3. An attribute dependence edge $(\mathtt{CA}_{\mathtt{from\_class}}, \mathtt{CA}_{\mathtt{to\_class}})$ between two CA nodes in an SDG is altered and represented between $\mathtt{CA}_{\mathtt{from\_class}}$ and a AT node of $\mathtt{CA}_{\mathtt{to\_class}}$ in the MDG.

4. A relationship dependence edge $(\mathtt{CA}_{\mathtt{from\_class}}, \mathtt{CA}_{\mathtt{to\_class}})$ between two CA nodes in a CDG is altered based on the type of message dependency between them in an SDG, and is either represented as a call dependence edge(as in (1)), or an attribute dependence edge(as in (3)) in an MDG.

We now explain how a message dependence is represented in an MDG. To easily map between the SDG and the MDG, every message dependence edge represented by a certain label I(k) in SDG $\mathtt{SDG}_\mathtt{i}$ is represented by a label $\mathtt{I}_\mathtt{i}(\mathtt{k})$ in the MDG.

Let us consider two SDGs $\mathtt{SDG}_1$ and $\mathtt{SDG}_2$. Let the message dependence edges in $\mathtt{SDG}_1$ be represented by labels I(1)... I(i), and that in $\mathtt{SDG}_2$ be represented by labels I(1)...I(j). During integration, let us assume that $\mathtt{SDG}_2$ has been arbitrarily selected prior to $\mathtt{SDG}_1$. That is, Step-1 considers $\mathtt{SDG}_2$. The message dependence edges of $\mathtt{SDG}_2$ represented by $\mathtt{I}(1)\ldots\mathtt{I}(j)$ are updated and represented first with labels $\mathtt{I}_2(1)\ldots\mathtt{I}_2(j)$ in the MDG. Suppose $\mathtt{SDG}_1$ is selected in the next step of integration. In this case, the message dependence edges in $\mathtt{SDG}_1$ represented by $\mathtt{I}(1)\ldots\mathtt{I}(i)$ are next updated and represented with labels $\mathtt{I}_1(1)\ldots\mathtt{I}_1(i)$ in the MDG. Interestingly, irrespective of the order of selection of SDGs, integration always constructs the same MDG. In the following, we only briefly outline the reasons for getting the same MDG to conserve space. A detailed proof of this has been reported in [10].

Suppose a UML architectural model has been represented in the MDG through



a complete integration process. For every such integration, we consider that CDGs and SDGs are selected in an arbitrary order. The resulting MDG obtained after every such integration would always be identical, and all such identical MDGs can be compared based on the following points:

1. Number of nodes and edges along with their types in the MDG.

   Each of the CDGs and SDGs is a graph consisting of a set of nodes and edges. During the integration process, irrespective of the order of considering the CDGs and SDGs, the nodes and edges representing the resultant MDG would always comprise the same number and type of nodes and edges after completion of all the steps in an integration process.

2. Edges among different nodes representing various dependencies in MDG.

   In the integration process, the different dependencies in the CDGs and SDGs are represented in the MDG. The number of nodes and edges in MDG are decided based on point (1) above. The integration process would result in equivalent sets of nodes and edges. This means that the dependencies among different nodes remain of the same after successive integrations.

3. Naming of the nodes of MDG.

   The nodes in MDG have exactly the same representation as the nodes in CDG and SDG. The representation of IT node in the MDG is an exception. This is because every IT node represents an interaction occurrence, and every interaction occurrence has an associated sequence diagram having an assigned label. The label assigned to a sequence diagram gets associated with the IT node when it is represented in the MDG. Moreover, once a sequence diagram is assigned a label, the same label is subsequently assigned to its SDG. This ensures that the order in which the sequence diagrams are taken up during the integration process does not change the representation of an IT node in the MDG. Considering these facts, an IT node represented as IT_i in SDG_n is represented as IT_{ni} in the MDG to differentiate it from IT nodes associated with other SDGs.

4. Assigning labels to different dependence edges represented in MDG.

   The dependence edges represented in the MDG are based on the dependencies among SDG nodes and its related CDG nodes. In the process of representing these dependencies, every dependence edge represented by a certain label I(k) in SDG SDG_i is represented by a label $I_i(k)$ in the MDG as mentioned earlier. This ensures that, if the steps of the integration process are repeated, they do not change the labels associated with different dependence edges in the MDG.

An Example MDG

In this section, we explain the construction of the MDG for the example UML model shown in Figure 1. And, Figure 7 shows the MDG obtained after applying a complete integration. The integration process uses the CDG and the SDG shown in Figures 4 and 5 respectively to obtain the MDG.

Let the CDG of Figure 4 be denoted by CDG_1, and the SDG of Figure 5 be denoted by SDG_1.

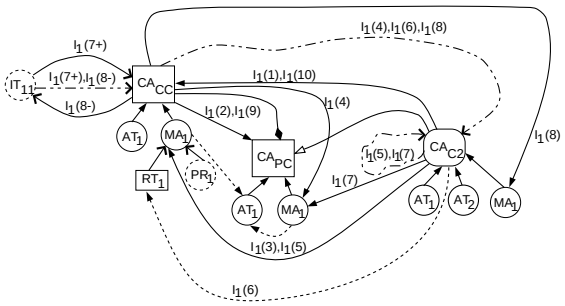


Figure 7: MDG representation for the example CDG and SDG of Figures 4 and 5

The CDG_1 in Figure 4 has four class access nodes $CA_{PC}, CA_{CC}, CA_{C1}$, and CA_{C2} but the SDG SDG_1 in Figure 5 uses only three class access nodes CA_{C2}, CA_{CC}, and CA_{PC}. One of the classes `ChildClass1` is not represented in SDG SDG_1. Therefore, the corresponding node CA_{C1} in CDG_1 is not represented in the MDG of Figure 7. SDG_1 of Figure 5 contains an interaction(`IT`) node IT_1, and has been represented as IT_{11} in the MDG of Figure 7.

SDG_1 of Figure 5 has its call dependence edges annotated with the labels `I(1)`, `I(2)`, `I(3)`, `I(4)`, `I(5)`, `I(7)`, `I(8)`, `I(9)`, and `I(10)`. The call dependence edge (CA_{C2}, CA_{CC}) has been annotated with the label `I(3)` in SDG_1. It forms an altered edge and is represented by the call dependence edge (CA_{C2}, MA_1) annotated with label $I_1(3)$ in the MDG. Here, MA_1 denotes the method access(`MA`) node for the method `cc_method1()` of `CompositeClass`. The other call dependence edges are similarly altered, and represented in the MDG.

The return dependence edge (CA_{C2}, CA_{CC}) in SDG_1 of Figure 5 has been annotated with the label `I(6)`. It is altered and represented as the return dependence edge (CA_{C2}, RT_1) and has been labeled $I_1(6)$ in the MDG. Here, RT_1 denotes the return(`RT`) node representing the return value of method `cc_method1()` of `CompositeClass`.

It can be seen in the MDG of Figure 7, that all the message dependence edges in SDG_1 have been represented in the MDG with their corresponding labels modified. We explain it by using an example. Let SDG_1 and some other SDG_i contain message dependence edges annotated with labels `I(1)`...`I(8)`. These message dependence edges can easily be distinguished after they are represented in the MDG. For SDG_1 those edges are represented by the edges $I_1(1) \ldots I_1(8)$ respectively in the MDG,

and so on. For any SDG_i, they are represented by $I_i(1) \ldots I_i(8)$ in the MDG.

6 SLICING USING MDG

A static slice can be computed by identifying the different architectural elements and the dependencies among them for an UML model. These selectively identified architectural elements can comprise classes and their objects, different attributes, and the method calls. We collectively term these identified architectural elements as a slice of an architecture. These architectural elements are identified based on a *slicing criterion*. In the following, we define a slicing criterion, and its corresponding computed static slice for an architectural model.

Slicing Criterion - Given the MDG G_M of an architecture having a CA node $CA_{CL(n)}$ and an edge with label $I_i(k)$ from that CA node in G_M, a *slicing criterion* is of the form $[CA_{CL(n)}, I_i(k)]$. The slicing criterion represented by $[CA_{CL(n)}, I_i(k)]$ is said to involve an object of class $CL(n)$, and a message transfer represented by $I_i(k)$ using a message dependence edge in the MDG.

Architectural Model Slice - An *architectural model slice* is defined as the part of an architecture comprising a set of class objects along with their related attributes and methods which participate, either directly or indirectly, in a message transfer represented by $I_i(k)$ based on a slicing criterion $[CA_{CL(n)}, I_i(k)]$. The static architectural model slice is represented by $StaticArchModelSlice(CA_{CL(n)}, I_i(k))$, where $[CA_{CL(n)}, I_i(k)]$ is the slicing criterion.

For architectural model slicing(discussed in next subsection), our algorithm traverses the edges of the MDG to compute a static slice for a given slicing criterion. During the process of MDG traversal based on the slicing criterion, the slicer computes and stores the slice in $StaticArchModelSlice$. At the end of traversal, $StaticArchModelSlice$ contains the computed slice.

Computation of A Static Slice - Let $StaticArchModelSlice(CA_{CL(n_1)}, I_i(k))$ be an static slice for an architectural model with respect to a slicing criterion $[CA_{CL(n_1)}, I_i(k)]$. Let $\{(CA_{CL(n_1)}, CA_{CL(n_2)}),\ldots,(CA_{CL(n_1)},CA_{CL(n_k)})\}$ be the set of all dependence edges that can be traversed from $CA_{CL(n_1)}$ in the MDG G_M for a message transfer represented using $I_i(k)$. Then, the computation of a static slice can be represented as,

$$\begin{aligned} StaticArchModelSlice(CA_{CL(n_1)}, I_i(k)) = \{&(CA_{CL(n_1)}, CA_{CL(n_2)}) \\ &\cup \ldots \cup (CA_{CL(n_1)}, CA_{CL(n_k)})\} \\ \cup\ & StaticArchModelSlice(CA_{CL(n_2)}, I_i(k)) \\ \cup \ldots \cup\ & StaticArchModelSlice(CA_{CL(n_k)}, I_i(k)) \end{aligned}$$

Architectural Model Slicing through MDG Traversal(AMSMT)

We have named our static slicing algorithm Architectural Model Slicing through MDG Traversal(AMSMT). AMSMT takes a UML architectural model and a slicing criterion as its input and produces the computed static slice. The operation of our slicing algorithm can be divided into three main phases: (i) Graph construction (ii) Integration of graphs into MDG (iii) Traversal of MDG to compute a static slice.

In the first phase, the CDGs and SDGs are constructed from a static analysis of the UML class and sequence diagrams respectively. The second phase of the algorithm constructs the MDG by integrating the constructed CDGs and SDGs. The MDG constructed during the second phase is traversed for the given slicing criterion in the third phase of AMSMT. The traversal of MDG helps to identify different architectural elements forming the slice. This slice is stored in `StaticArchModelSlice`, and can be fetched anytime. This helps AMSMT to save storage space as the same static slice information is not stored at any other nodes in the MDG.

AMSMT Algorithm

This subsection presents our AMSMT algorithm in pseudo-code form. It assumes that the class and sequence diagrams are given in XML format.

```
Algorithm AMSMT
Requires : SetCD = {CD_1 ... CD_m}
           {* Set of class diagrams, each given in XML *}
           SetSD = {SD_1 ... SD_n}
           {* Set of sequence diagrams, each given in XML *}
Initialization : Graph SetCDG = NULL{*Set of graphs, each representing a
CDG*}
                 Graph SetSDG = NULL{*Set of graphs, each representing a
SDG*}
                 Graph MDG = NULL
Input : [CA_CL(n), I_i(k)]  {* Slicing Criterion *}
Output : StaticArchModelSlice(CA_CL(n), I_i(k))
Phase 1: Static graph construction
        for every i < m and CD_i ∈ SetCD do
        {* Call a procedure for CDG construction *}
            CDG_i = ConstructCDG(CD_i);
            SetCDG = SetCDG ∪ CDG_i
        end for
        for every j < n and SD_j ∈ SetSD do
        {* Call a procedure for SDG construction *}
            SDG_j = ConstructSDG(SD_j);
            SetSDG = SetSDG ∪ SDG_j
```

```
        end for
Phase 2 : Integration of graphs into MDG
        {* Call a procedure for MDG construction *}
            MDG = ConstructMDG(SetCDG, SetSDG);
Phase 3 : Traversal of MDG to compute a static slice
        for every dependent node traversed from CA_CL(n) corresponding to I_i(k) do
            TraverseMDG(MDG, I_i(k));
            StaticArchModelSlice(CA_CL(n), I_i(k)) = TrackStaticSlice(MDG, I_i(k));
        end for
        DisplaySlice(StaticArchModelSlice(CA_CL(n), I_i(k)));
End AMSMT
```

After the slicing criterion is given as an input, AMSMT computes the static slice on a fly by executing Phase 3 of the algorithm.

SetCDG and SetSDG are first initialized to NULL indicating that initially no CDGs or SDGs exist. The MDG is also initialized to NULL indicating that initially no nodes or edges exist in the MDG. The loop of Phase 3 traverses the MDG for any given slicing criterion. This loop calls two other procedures viz., TraverseMDG() and TrackStaticSlice(). Next, Phase 3 of the algorithm ends with a call to the procedure DisplaySlice(). A detailed description of the various procedures called during execution of AMSMT is available in [10], including the pseudo-code representation. The different procedures called in Phase 2 and 3 of AMSMT perform the following tasks:

- ConstructMDG() - This procedure implements the steps of the integration process discussed in the previous section. It takes two parameters viz., a SetCDG, and SetSDG computed in Phase 1, and constructs the MDG.

- TraverseMDG() - This procedure identifies the dependence edges based on the slicing criterion and traverses the nodes of the MDG. It takes two parameters from the given slicing criterion viz., a MDG, and $I_i(k)$ representing a dependence edge in the MDG. This process of traversing the nodes in the MDG identifies all the relevant model elements to be included in the static slice.

- TrackStaticSlice() - This procedure tracks the process of static slice computation during MDG traversal. It stores the computed slice using the data structure StaticArchModelSlice. It takes two parameters similar to the TraverseMDG() procedure.

- DisplaySlice() - This procedure takes only one parameter representing the computed static slice viz., StaticArchModelSlice($CA_{CL(class)}, I_i(k)$) and displays it.

Complexity Analysis of AMSMT

In this subsection, we analyze the space and time complexities of AMSMT. To conserve space, the detailed proofs for the space and time complexities of AMSMT are not reported here, and can be found in [10]. However, we briefly discuss a few important aspects relevant to the complexity of AMSMT.

The space complexity of our proposed AMSMT algorithm is $O(T^2)$, where T is the number of model elements represented as nodes in an MDG. There is no additional space requirement at run-time, as no new nodes are added to the MDG during the process of slice computation. However, during phase 3 AMSMT maintains a data structure `StaticArchModelSlice` to track the process of slice computation. But the space needed for `StaticArchModelSlice` is negligible for nontrivial architectures in comparison to $O(T^2)$ space needed to construct CDGs, SDGs and the MDG in phase 1 and 2 of AMSMT. Therefore, the total space requirement of AMSMT is limited by $O(T^2)$.

In AMSMT, the phase 1 constructs the CDGs and SDGs, and phase 2 constructs the MDG. The time complexity of each of these phases is $O(T)$, where T is the number of model elements. After the MDG is constructed, phase 3 traverses the MDG in $O(T)$ time and computes the static slice. Combining the time required to execute each of the three phases of AMSMT, the time requirement for AMSMT sums up to $O(3 * T)$. Hence, the time complexity of AMSMT is of the order of $O(T)$.

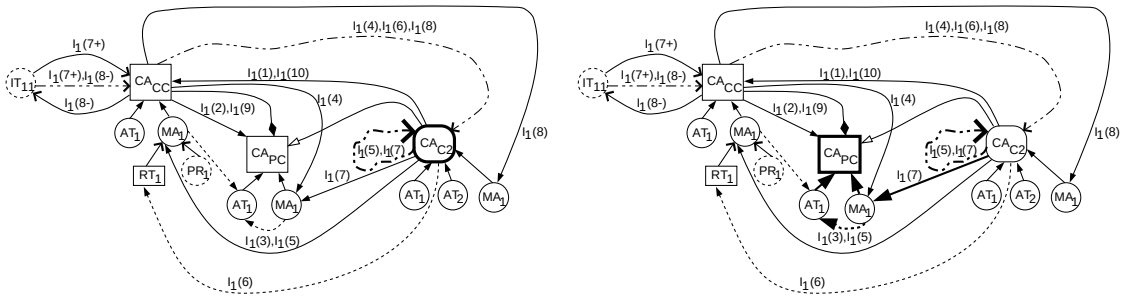
We illustrate the working of AMSMT using different examples in the next subsection followed by an overview of an implementation of the AMSMT algorithm.

Illustration of Working of AMSMT

We explain the working of our AMSMT algorithm by using the example MDG of Figure 7 obtained at the end of Phase 2 of the algorithm. We assume that the CDGs and SDGs of Figures 4 and 5 have already been constructed by applying Phase 1 of the algorithm.

Let us consider computation of the static architectural model slice for the slicing criterion $[CA_{C2}, I_1(7)]$, $[CA_{PC}, I_1(7)]$ and $[CA_{PC}, I_1(5) \ldots I_1(7)]$. We show the MDG obtained during Phase 3 of AMSMT in Figures 8(a), 8(b), and 9 respectively displaying the classes, methods, attributes, and interactions contributing to the corresponding slice. The MDGs of Figures 8 and 9 are identical except for the case that each of them have different dependence edges highlighted for the particular slicing criterion.

Figure 8(a) shows MDG traversal using the class access(`CA`) node of `ChildClass2` highlighted for the slicing criterion $[CA_{C2}, I_1(7)]$. It depicts the MDG maintained by the architectural model slicer for the class access node CA_{C2} denoting `ChildClass2`. It only shows the control dependence edge (CA_{C2}, CA_{C2}) highlighted(made thick) to indicate that it is the only traversed edge. Therefore, the static slice for the slicing criterion $[CA_{C2}, I_1(7)]$ includes a single edge (CA_{C2}, CA_{C2}). This implies that, only the



(a) (b)

Figure 8: (a) MDG showing static architectural model slice computed for the slicing criterion $[\mathtt{CA_{C2}}, \mathtt{I_1(7)}]$ during the execution of phase 3 of AMSMT (b) MDG showing static architectural model slice computed for the slicing criterion $[\mathtt{CA_{PC}}, \mathtt{I_1(7)}]$ during the execution of phase 3 of AMSMT

object of class `ChildClass2` contributes to the slice.

Figure 8(b) shows MDG traversal using the class access(`CA`) node of `ParentClass` highlighted for the slicing criterion $[\mathtt{CA_{PC}}, \mathtt{I_1(7)}]$. It depicts the MDG maintained by the architectural model slicer for the class access node $\mathtt{CA_{PC}}$ denoting `ParentClass`. It shows control dependence $(\mathtt{CA_{C2}}, \mathtt{CA_{C2}})$, call dependence $(\mathtt{CA_{C2}}, \mathtt{MA_1})$, data dependence $(\mathtt{MA_1}, \mathtt{AT_1})$, and member dependence edges $(\mathtt{AT_1}, \mathtt{CA_{PC}})$, and $(\mathtt{MA_1}, \mathtt{CA_{PC}})$ highlighted indicating the traversed edges. Therefore, the static slice for the slicing criterion $[\mathtt{CA_{PC}}, \mathtt{I_1(7)}]$ includes all those nodes connected by these edges. This implies that an anonymous object of classes `ChildClass2` and `ParentClass`, method `pc_method1()` and an attribute `pc_1` of class `ParentClass` contribute to the slice. This example shows how slicing uncovers the data dependencies on invocation of a method.

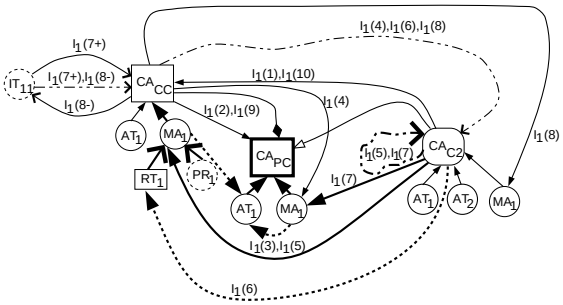


Figure 9: MDG showing the static architectural model slice computed for the slicing criterion $[\mathtt{CA_{PC}}, \mathtt{I_1(5)} \ldots \mathtt{I_1(7)}]$ during the execution of phase 3 of AMSMT

Figure 9 shows the MDG traversal using the class access(`CA`) node of `ParentClass` highlighted for the slicing criterion $[\mathtt{CA_{PC}}, \mathtt{I_1(5)} \ldots \mathtt{I_1(7)}]$. It shows control dependence $(\mathtt{CA_{C2}}, \mathtt{CA_{C2}})$, call dependencies $(\mathtt{CA_{C2}}, \mathtt{PC(MA_1)})$ and $(\mathtt{CA_{C2}}, \mathtt{CC(MA_1)})$, data dependencies $(\mathtt{PC(MA_1)}, \mathtt{PC(AT_1)})$ and $(\mathtt{CC(MA_1)}, \mathtt{PC(AT_1)})$, member dependencies $(\mathtt{AT_1}, \mathtt{CA_{PC}}), (\mathtt{MA_1}, \mathtt{CA_{PC}})$,

and (MA_1, CA_{CC}), method dependencies (RT_1, MA_1) and (PR_1, MA_1), and the return dependence edge (CA_{C2}, RT_1) highlighted to indicate the traversed edges. Therefore, the static slice for the slicing criterion [$CA_{PC}, I_1(5) \ldots I_1(7)$] includes those nodes connected by all these edges. This implies that the anonymous object of classes `ChildClass2`, `ParentClass` and `CompositeClass`, method `pc_method1()` and an attribute `pc_1` of class `ParentClass`, and a method `cc_m- ethod1()` of the class `CompositeClass` contribute to the slice when we consider the slice computation for an interaction comprising of messages represented by `I(5),I(6)` and `I(7)`. This example shows how the slicing algorithm takes into account various dependencies for an interaction having more than one message interchange.

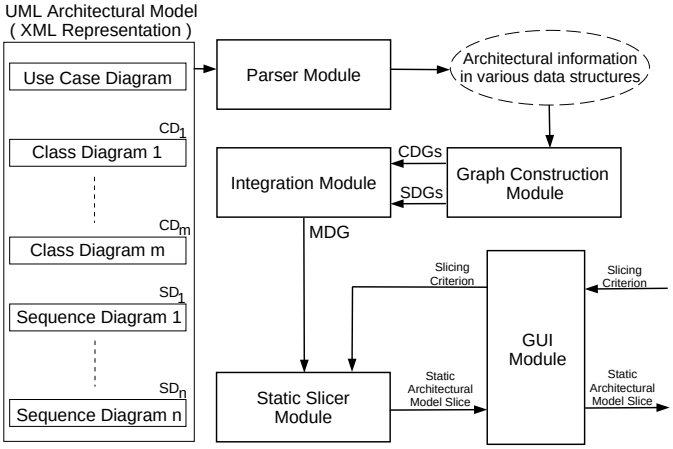


Figure 10: Schematic design of the prototype tool SSUAM

Experimental Studies

In this section, we present an implementation of our AMSMT algorithm. We have implemented a prototype tool to compute static architectural model slices using our AMSMT algorithm and have named it SSUAM (Static Slicer for UML Architectural Models). Our tool can be integrated with many UML model development tools such as MagicDraw UML [17,18] which can export UML models in XML (Extensible Markup Language) format. This makes SSUAM independent of any specific CASE tool. We have implemented our tool using Java and used the Document Object Model(DOM) API of Java for parsing XML files. DOM provides a standard programming interface that is used in a wide variety of modeling environments and applications. Moreover, XML is increasingly being used for representing different kinds of model related information that may be stored and used in diverse systems. Additionally, XML presents the data associated with these UML models as documents, and the DOM may be used to manage this data.

The schematic design of SSUAM has been shown in Figure 10. The different components of this design are explained in the following.

Table 1: Average runtime requirements of AMSMT algorithm

Sr. No.	Architecture size (# classes)	Number of objects	Normal exec. time (in ms)	Average runtime (in ms)	Over head (in ms)
1	3	7	40	55	15
2	6	15	52	74	22
3	13	33	85	103	18
4	18	54	112	137	25
5	22	70	145	166	21
6	34	89	178	208	30
7	45	123	220	264	44

SSUAM takes as input a UML architectural model comprising the use-case diagram, class and sequence diagrams in XML format. This is parsed by the *Parser Module*, which extracts information regarding different classes, their attributes and methods from the class diagrams. This module also gathers information regarding objects of different classes participating in interactions along with the messages exchanged among them from the sequence diagrams. The *Parser Module* then initializes all the data structures needed to construct the CDGs and SDGs. The *Graph Construction Module* constructs a CDG for every class diagram, and an SDG for every sequence diagram using the information present in the data structures initialized by the *Parser Module*. One CDG per class diagram, and one SDG per sequence diagram are constructed, and added to `SetCDG` and `SetSDG` respectively. This has been represented in the schematic of Figure 10 by `CDGs(or SetCDG)` and `SDGs(or SetSDG)`. The *Parser Module* and the *Graph Construction Module* together implement the *Phase 1* of AMSMT.

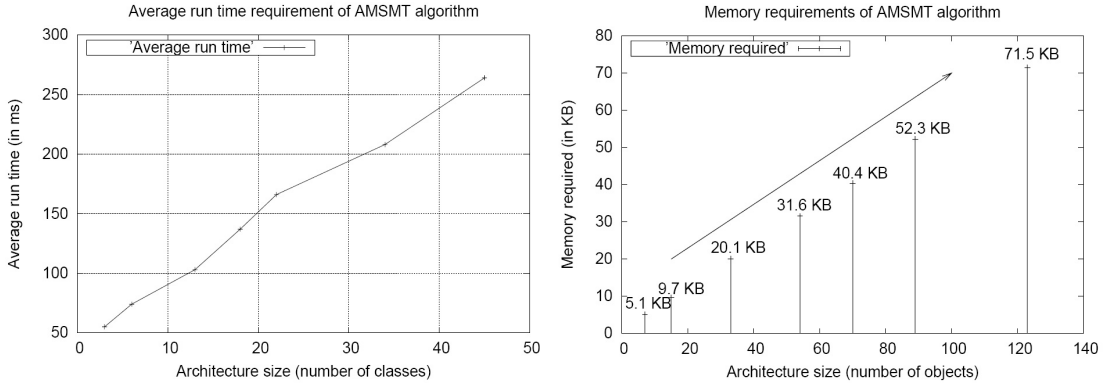
The *Integration Module* constructs an MDG by using the sets `SetCDG` and `SetSDG`. The sets `SetCDG` and `SetSDG` consist of CDGs and SDGs respectively. This module implements the `ConstructMDG()` procedure that is executed during *Phase 2* of the AMSMT. Next, based on the specified slicing criterion the *Static Slicer Module* of SSUAM traverses the MDG for computation of the static slice.

SSUAM supports a graphical user interface(GUI) for all user interactions such as specification of slicing criteria, display of computed slice, etc. The *Static Slicer Module* traverses the MDG through the dependence edges based on the specified slicing criterion and computes a static architectural model slice. During MDG traversal, AMSMT stores the computed slice in the `StaticArchModelSlice` data structure. The *GUI Module* presents the computed slice through the highlighted dependence edges on the MDG. Together, the *Static Slicer Module* and the *GUI Module* implement the *Phase 3* of AMSMT.

We have conducted several experiments using SSUAM on a number of architectures described using UML and for different slicing criteria. Our prototype implementation makes the assumption that the information about various attributes used

by a class method are available in notes attached to class diagrams. This information is used to determine the data dependencies.

A summary of results from our experimental studies has been presented in Tables 1 and 2. Table 1 summarizes the average runtime requirements and overhead costs of the AMSMT algorithm. From the experimental results, it can be observed that the average runtime requirement increases sub-linearly with architecture sizes. Figure 11(a) graphically presents this result. The increase in runtime requirement with class size is possibly due to the increased size of the constructed DOM tree resulting from parsing the XML representations of the class and sequence diagrams. This increases the average runtime to execute Phase 1, and subsequently Phase 2 of the AMSMT. Moreover, any large sized architecture would finally require traversal of a large MDG during Phase 3 and contribute to increased average runtime of AMSMT.



(a) (b)

Figure 11: (a) Increase in average run time of AMSMT with increase in architecture size (b) Increase in memory requirement of AMSMT with increase in architecture size

Table 2 summarizes the memory requirements of the AMSMT algorithm. The plot of Figure 11(b) shows that the average memory requirement of AMSMT increases almost linearly with architecture size. This can be attributed to the fact that the architecture size described in terms of number of classes determines the runtime memory requirement to maintain the CDGs. In addition, the number of class objects and their interactions determine the runtime memory requirement to maintain the data structures for SDGs and the MDG. Also, the DOM tree maintained in order to construct the CDGs and SDGs incurs memory of the order of its nodes. And, the number of nodes in a DOM tree depend on the size of an architecture. Typically, the size of each node of a DOM tree is 24 bytes. However, Table 2 does not show the memory(in terms of executable code size) needed for execution of SSUAM, which for our implementation is 119 KB and remains almost the same for all cases. Moreover, our technique represents every class using a unique class access(CA) node in the MDG. This is irrespective of the number of objects of a class existing across various sequence diagrams. This obviates the necessity to create additional nodes in case of repeated instantiation of any architecture classes, and their

Table 2: Memory requirements of AMSMT algorithm

Sr. No.	Architecture size (# classes)	Number of objects	Mem. (G^*) (in KB)	Mem. (S^+) (in KB)	Total Memory (in KB)
1	3	7	4.5	0.6	5.1
2	6	15	8.5	1.2	9.7
3	13	33	17.5	2.6	20.1
4	18	54	28.0	3.6	31.6
5	22	70	36.0	4.4	40.4
6	34	89	45.5	6.8	52.3
7	45	123	62.5	9.0	71.5

G^* indicates memory needed to maintain all graphs in various data structures,
S^+ indicates memory needed to compute and store slice in various data structures

objects. This ensures that the data structures used to maintain an MDG remain bounded by the number of classes. Taking all these factors into consideration, we are confident that SSUAM can be used to slice large architectures.

7 COMPARISON WITH RELATED WORK

Architecture slicing of ADL architectural models has been investigated by Stafford *et al.* [19, 20], Zhao [21, 23, 26] and Kim [7]. Korel [8] has reported a work on slicing of state-based models. Those are not directly comparable to our work since most of those use architecture descriptions using some ADLs, or use some form of FSMs to consider architectures whereas we consider architectures represented in UML notation. The architectures used in these techniques do not separately distinguish between the structural and behavioral aspects of a system. This does not allow the computed architectural slices to completely uncover the dependencies existing among different model elements. Kagdi's [4] work focuses on model slicing using UML class diagrams. This work also is not directly comparable to our work since it does not consider any behavioral information from the UML models. In the absence of any directly comparable work, we compare our method with the existing architectural and model slicing methods reported for other ADLs and EFSM architectural models. Our algorithm for static slicing of UML architectural models incorporates several novelties as compared to other work reported in the literature. One important novelty is in the computation of a slice based on an integrated model constructed from several UML diagrams. The computed slice is based on the dependencies among different model elements that are distributed across various UML diagrams. Slicing based on an integrated model can correlate the information present in different model elements and help understand how changing one of them will have an impact on the rest of the model architecture.

The graph representations used in [21, 23, 25, 26] are based on information flow

while those in [5, 6, 7] are essentially event-based. These representations do not distinguish among the various control, data, communication or event dependencies that arise among components and connectors. There is, therefore always a possibility of computation of an inaccurate slice. Our graph representation is substantially different from all the other existing techniques and takes care of various model dependencies with a major focus on identifying data dependencies extractable from various UML architectural models.

The worst case time requirement and space complexity for the architecture slicing algorithms reported by Zhao [21,23,25,26] is quadratic in the number of components, connectors and the attachments. The computation of architecture slice in [5, 6, 7] is based on filtering of events based on the slicing criterion. The DSAS algorithm of Kim *et al.* [5, 6, 7] needs $O(N^2)$ space in the worst case and $O(N)$ time to extract architecture slices, where N is the total number of event occurrences. Note that N may be unbounded for large systems containing event cycles. Our AMSMT algorithm has the space complexity of $O(T^2)$ and a time complexity of $O(T)$, where T is the number of model elements represented as nodes in an MDG. AMSMT has no additional space overhead at run-time as no new nodes are added to the MDG during the process of slice computation.

Korel *et al.* [8] compute static model slices of EFSM models and then apply a slice reduction step after the computation of slice. Kagdi *et al.* [4] also focus on model slicing by considering the structural information from UML class diagrams only, whereas our slicing technique is based on an integrated intermediate model constructed from various UML diagrams and considers both the structural and the behavioral information.

8 CONCLUSION

We have presented a slicing technique for UML architectural models. Slicing UML architectural models is a difficult problem since the model information is distributed across several diagrams with implicit dependencies among them. We first construct an intermediate representation called MDG from various architectural model elements. MDG integrates the structural and behavioral aspects of an architectural design into a single representation. Our AMSMT algorithm uses the MDG representation to compute static slices. Such static slices can be used for studying the impact of design changes, reliability prediction, understanding large architectures, etc. We have implemented a prototype architectural slicing tool called SSUAM based on our AMSMT algorithm. We are now trying to enhance our intermediate model by integrating the state and activity models into MDG to compute more accurate slices.

REFERENCES

[1] Robert John Allen. *A formal approach to software architecture*. PhD thesis, Carnegie Mellon, School of Computer Science, January 1997. Chair-David Garlan.

[2] Paul Clements, Rick Kazman, and Mark Klein. *Evaluating Software Architectures: Methods and Case Studies*. SEI Series in Software Eng. Addison Wesley Professional, October 2002.

[3] Jose Daniel Garca, Jesus Carretero, Jose Mara Perez, Felix Garcia, and Rosa Filgueira. Specifying use case behavior with interaction models. *Journal of Object Technology*, 4(9):143–159, November-December 2005. http://www.jot.fm/issues/issue_2005_11/article5/.

[4] Huzefa Kagdi, Jonathan I. Maletic, and Andrew Sutton. Context-free slicing of uml class models. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 635–638, Washington, DC, USA, 2005. IEEE Computer Society.

[5] Taeho Kim, Yeong-Tae Song, Lawrence Chung, and Dung T. Huynh. Dynamic software architecture slicing. In *COMPSAC '99: 23rd International Computer Software and Applications Conference*, pages 61–66, Washington, DC, USA, 1999. IEEE Computer Society.

[6] Taeho Kim, Yeong-Tae Song, Lawrence Chung, and Dung T. Huynh. Software architecture analysis using dynamic slicing. In *Proceedings of AoM-IAoM 17th International Conference on Computer Science*, August 1999.

[7] Taeho Kim, Yeong-Tae Song, Lawrence Chung, and Dung T. Huynh. Software architecture analysis: A dynamic slicing approach. *Journal of Computer and Information Science*, 1(2):91–103, 2000.

[8] Bogdan Korel, Inderdeep Singh, Luay Tahat, and Boris Vaysburg. Slicing of state-based models. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, pages 34–43, Washington, DC, USA, 2003. IEEE Computer Society.

[9] Jaiprakash T. Lallchandani and R. Mall. Computation of dynamic slices for object-oriented concurrent programs. In *12th Asia-Pacific Software Engineering Conference (APSEC 2005)*, pages 341–350, Taipei, Taiwan, December 2005. IEEE Computer Society.

[10] Jaiprakash T. Lallchandani and R. Mall. Static slicing of UML models. Technical Report IIT-CS07-SE-13, Indian Institute of Technology(IIT), Kharagpur, West Bengal, India, February 2007.

[11] John McGregor. Complexity, its in the mind of the beholder. *Journal of Object Technology*, 5(1):31–37, January-February 2006. http://www.jot.fm/issues/issue_2006_01/column3.

[12] John D. McGregor. Software architecture. *Journal of Object Technology*, 3(5):65–77, May-June 2004. http://www.jot.fm/issues/issue_2004_05/column7/.

[13] Nenad Medvidovic, David S. Rosenblum, David F. Redmiles, and Jason E. Robbins. Modeling software architectures in the unified modeling language. *ACM Transactions on Software Engineering and Methodology*, 11(1):2–57, January 2002.

[14] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, January 2000. Reprinted in Rational Developer Network: Seminal Papers on Software Architecture. Rational Software Corporation (July 2001).

[15] G. B. Mund and R. Mall. An efficient interprocedural dynamic slicing method. *Journal of Systems and Software*, 79(6):791–806, 2006.

[16] G. B. Mund, R. Mall, and S. Sarkar. An efficient dynamic program slicing technique. *Information & Software Technology*, 44(2):123–132, 2002.

[17] Dave Neuendorf. Review of magicdraw uml 11.5 professional edition. *Journal of Object Technology*, 5(7):115–118, September-October 2006. http://www.jot.fm/issues/issue_2006_09/review8.

[18] N.M.Inc. Magicdraw uml v11.6. http://www.magicdraw.com.

[19] J. Stafford, D. Richardson, and A. Wolf. Aladdin: A tool for architecture-level dependence analysis of software systems. Technical Report CU-CS-858-98, University of Colorado, Dept. of Computer Science, April 1998.

[20] J. Stafford, A. Wolf, and M. Caporuscio. The application of dependence analysis to software architecture descriptions. In *Lecture Notes in Computer Science*, volume 2804, pages 52–62, 2003.

[21] Jianjun Zhao. Slicing software architectures. Technical Report 97-SE-137, Information Processing Society of Japan (IPSJ), November 1997.

[22] Jianjun Zhao. A slicing-based approach to extracting reusable software architectures. In *CSMR*, pages 215–223, October 2000.

[23] Jianjun Zhao. Applying slicing technique to software architectures. *CoRR*, cs.SE/0105008, 2001.

[24] Jianjun Zhao. On assessing the complexity of software architectures. *CoRR*, cs.SE/0105010, 2001.

[25] Jianjun Zhao. Using dependence analysis to support software architecture understanding. *CoRR*, cs.SE/0105009, 2001.

[26] Jianjun Zhao, Hongji Yang, Liming Xiang, and Baowen Xu. *Architectural Slicing to Support System Evolution*. Idea Group Publishing, Hershey, PA, USA, 2005.

ABOUT THE AUTHORS

Jaiprakash T. Lallchandani is a PhD student and a senior research fellow at the Department of Computer Science and Engineering at Indian Institute of Technology(IIT), Kharagpur, India. He can be reached at jtl@cse.iitkgp.ernet.in.

R. Mall is currently a professor in the department of Computer Science and Engineering, Indian Institute of Technology(IIT), Kharagpur, India. He obtained his Ph.D, M.E.,and B.E. degrees from Indian Institute of Science(IISc), Bangalore, India. He has published over 100 refereed research papers and has authored two books. He is a member of the domain experts board of the International Journal of Patterns(IJOP). He was the general chair of IEEE Indicon 2004 and program chair for CIT 2005. He was also a program committee member for a large number of international conferences. His current research interests include analysis and testing of object-oriented programs. He can be reached at rajib@cse.iitkgp.ernet.in.