Practical Approach for Model Based Slicing

Thesis submitted in partial fulfillment of the requirements for the award of degree of

Master of Engineering
in
Software Engineering

Submitted By
Rupinder Singh
(Roll No. 801131022)

Under the supervision of:
Mr. Vinay Arora
Assistant Professor

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT
THAPAR UNIVERSITY
PATIALA – 147004

June 2013

Certificate

I hereby certify that the work which is being presented in the thesis entitled, **"Practical Approach for Model Based Slicing"**, in partial fulfillment of the requirements for the award of degree of Master of Engineering in *Software Engineering* submitted in Computer Science and Engineering Department of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of *Mr. Vinay Arora* and refers other researcher's work which are duly listed in the reference section.

The matter presented in the thesis has not been submitted for award of any other degree of this or any other University.

(Rupinder Singh)

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

(Mr. Vinay Arora)
Assistant Professor
CSED, Thapar University
Patiala

Countersigned by

(Dr. Maninder Singh)
Head of Department
CSED, Thapar University
Patiala

(Dr. S. K. Mohapatra)
Dean (Academic Affairs)
Thapar University
Patiala

Acknowledgement

First of all, I am thankful to God for all the blessings and showing me the right direction. Due to the mercy of God, it has been made possible for me to reach so far.

I wish to express my deep gratitude to Mr. Vinay Arora, Assistant Professor, Computer Science and Engineering Department, Thapar University, Patiala for providing his support throughout the span of my thesis. This work would not have been possible without his encouragement and valuable guidance.

I am also thankful to Dr. Maninder Singh, Head, Computer Science and Engineering Department for his kind help and cooperation. I express my gratitude to all the staff member of Computer Science and Engineering Department for providing me all the facilities required for the completion of my thesis work.

I express my heartfelt thanks to my parents, my sisters and my well wishers for their co-operation, which they were always ready to extend. At the end, I want to express my appreciation to every person who contributed with either inspirational or actual work to this thesis.

Rupinder Singh

Rupinder Singh

Abstract

Software testing is an activity that aims at evaluating an attribute or capability of system and determining that whether it meets required expectations. The most intellectually challenging part of testing is the design of test cases. Test case generation from design specifications has the added advantage of allowing test cases to be available early in the software development cycle. Now days, UML has been widely used for object oriented modeling and design. This is due to the fact that UML metamodel extends support to describe structural and behavioral aspects of an architecture.

However, it is still difficult to understand this behavior, because the size of automatically generated model diagrams tends to be huge. To overcome this problem of software visualization model based slicing technique came into existence. Model based slicing is a decomposition technique to extract and identify relevant model parts or related elements across diverse model views. We have proposed a novel methodology to extract the sub-model from a big model diagrams on the basis of slicing criteria. The proposed methodology use the concept of model based slicing to slice the sequence diagram to extract the desired chunk. In the presented approach UML, conversion of UML into XML, Java DOM API for parsing and slicing has been used.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

1.1 Software Testing

1.1.1 The Notion of Software Testing

Software testing is an evaluation process to determine the presence of errors in code snippet. Software testing cannot completely test software because exhaustive testing is not possible due to time and resource constraints. Testing is fundamentally a comparison activity in which the results are monitored for specific inputs. Software is subjected to different probing inputs and its behavior is evaluated against expected outcomes. Testing is the dynamic analysis of the product [1], means that the testing activity probes software for faults and failures while it is actually executed. It is apart from static code analysis, in which analysis is performed without actually executing the program. According to *ANSI/IEEE* 1059 standard [2, 3], Testing can be defined as "A process of analyzing a software item to detect the differences between existing and required conditions (that is defects/errors/bugs) and to evaluate the features of the software item".

Software testing incorporates verification and validation (V&V) technique. Verification and validation uses reviews, analysis and techniques to determine whether a software system and its intermediate products fulfill the expected fundamental capabilities and quality attributes [3].

There are some pre-established principles about testing software. Firstly, testing is a process that confirms the existence of quality, not establishing quality. Quality is the overall responsibility of the project team members and is established through right combinations of methods and tools. Quotes Brian Marick [4], a software testing consultant, said that the first mistake that people make is thinking that the testing team is responsible for assuring quality. Secondly, the prime objective of testing is to discover faults that are preventing the software in meeting customer requirements. Moreover, testing requires planning and designing of test cases and the testing effort should focus on

areas that are most error prone. The testing process progresses from component level to system level in an incremental way.

1.1.2 Test Levels

During the lifecycle of software development, testing is performed at several stages as the software evolved component by component. The accomplishment of reaching a stage in the development of software calls for testing the developed capabilities. Test driven development takes a different approach in which firstly tests are developed and then functionality has been designed around those tests. The testing at defined stages is termed as test levels and these levels progresses from individual units to combine into larger components. Simple projects may consist of only one or two levels of testing while complex projects may have more levels [5]. Figure 1.1 given below depicts the traditional V- model with added testing levels.

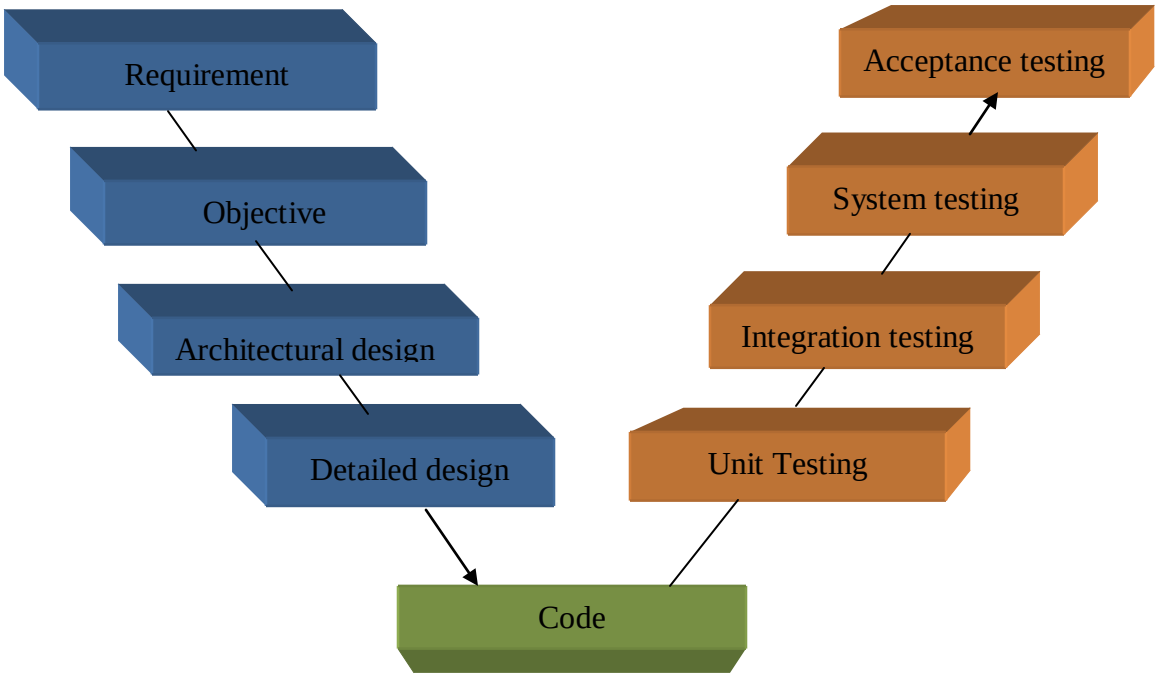

Figure 1.1: V-model of testing [5]

The identifiable levels of testing in the V-model are unit testing, integration testing, system testing and acceptance testing. According to *IEEE standard 1008-1987*, unit testing activities constitutes of planning the general approach, resources and determining

features to be tested, refining the general plan, designing the set of tests, implementing the refined plan and designing, executing the test procedure and checking for termination and evaluating the test effort and unit [6].

As the individual modules are integrated together, there are chances for finding bugs related to the interfaces between modules. It is because integration of modules might not provide the desired functionality, as data can be lost across interfaces, imprecision in calculations may be magnified, and interfacing faults might not be detected by unit testing [7]. These faults can be identified by integration testing. The different approaches used for integration testing includes incremental integration (top-down and bottom-up integration) and big-bang. Figure 1.2 and 1.3 below shows the top-down and bottom-up integration strategies respectively [8].

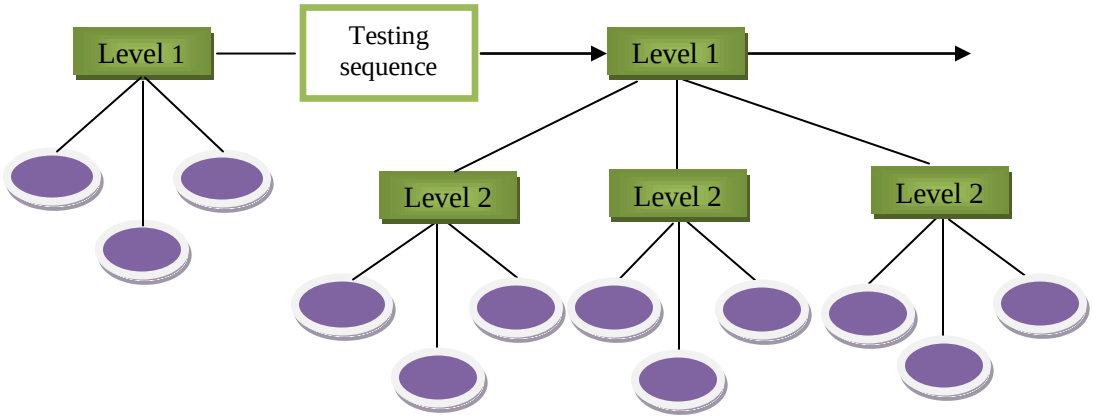

Figure 1.2: Top-down integration strategies [8]

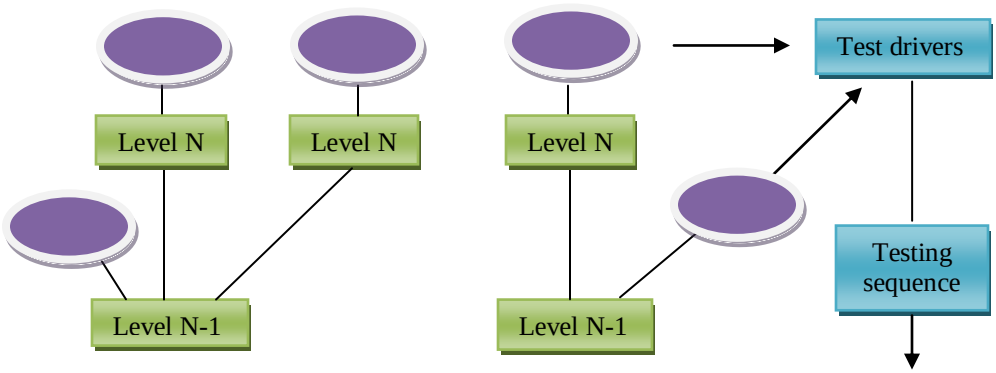

Figure 1.3: Bottom-up integration strategies [8]

The objective of system testing is to determine that the software meets all of its requirements as mentioned in the Software Requirements Specifications (SRS) document. The focus of system testing is at the requirements level. As part of system and integration testing, regression testing is performed to determine if the software still meets the requirements if changes are made to the software [7]. Acceptance testing is normally done by the end customers or while customers are partially involved. Normally, it involves selecting tests from the system testing phase or the ones defined by the users [7].

1.1.3. Types of Testing

1.1.3.1. Black Box

Black box testing is the testing of a piece of software without regard to its underlying implementation. Specifically, it dictates that test cases for a piece of software are to be generated based solely on an examination of the specification (external description) for that piece of software. The goal of black box testing is to demonstrate that the software being tested does not adhere to its external specification. Figure 1.4 represents the overview of black box testing. The objective is to search for interface errors, function or process errors, performance shortcomings, start-up/shutdown errors, and errors in local (module) databases by selecting appropriate inputs and external conditions and monitoring outputs.

Black Box Testing Techniques includes: Equivalence Partitioning, Boundary Value Analysis and Comparison Testing.

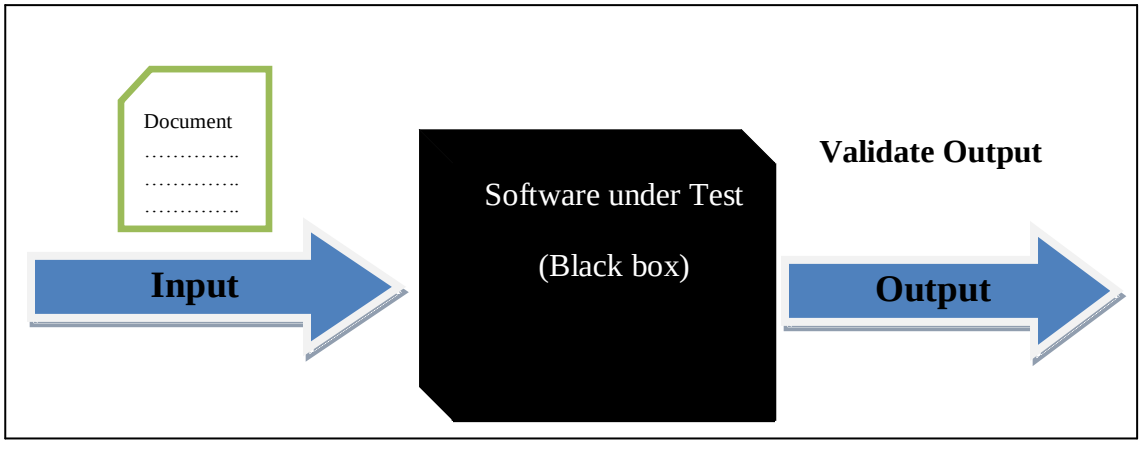

Figure 1.4: Black box testing [4, 9]

1.1.3.2. White Box Testing

White box testing is also known as *glass box testing* .It is a test case design method that uses the control structure of the procedural design to derive test cases [9]. Using white box testing methods, the software engineer can derive test cases that

1. guarantee that all *independent paths* within a module have been exercised at least once.
2. exercise all logical decisions on their true and false sides.
3. execute all loops at their boundaries and within their operational bounds.
4. exercise internal data structures to assure their validity.

Thus white box testing is the testing of the underlying implementation of a piece of software (e.g., source code) without regard to the specification (external description). The goal of white box testing of source code is to identify infinite loops, paths through the code which should be allowed, but which cannot be executed and dead (unreachable) code etc. Figure 1.5, shows the structural overview of white box testing techniques. White box testing incorporates control flow and data flow testing techniques.

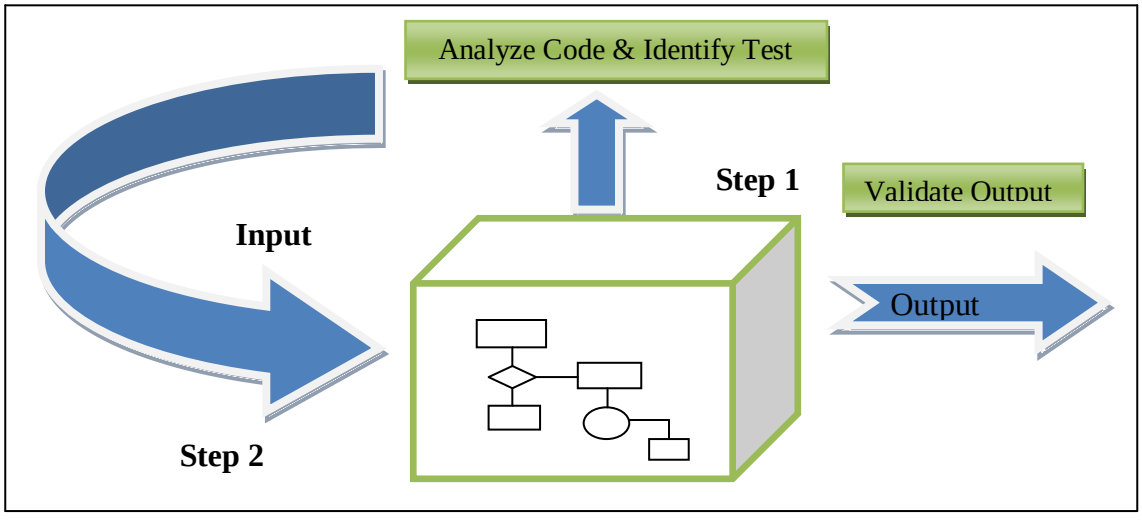

Figure 1.5: White box testing [4, 9]

Control flow testing: Testing on the basis of the flow of control of a program. This can be categorized under following types:

- *Statement:* each statement executed at least once
- *Branch:* each branch traversed (and every entry point taken) at least once
- *Condition:* each condition has to be scanned for it true and false value at least once.
- *Branch/Condition:* both Branch and Condition coverage achieved.
- *Multiple Conditions:* multiple conditions coverage technique states that test cases must be written such that all possible combinations of conditions in each decision are taken at least once.
- *Loop:* loop coverage technique states that test cases must be written to test the loop counters.

Data flow testing: Testing on the basis of the flow of data in the application. This can be partitioned under following types [1]:

- *All Definition-Uses:* It requires that every definition of every variable to every use of that definition be exercised under the test.
- *All Uses:* this test set include at least one path segment from every definition of every variable to every use of that definition
- *All p-uses/some c-uses:* In All p-uses/some c-uses, test set include at least one path segment from every definition of every variable to every predicate use of that definition; if there are definitions of variables that are not covered by the above prescription then additional computational use test cases are required to cover every definition.
- *All c-uses/ some p-uses:* In All c-uses/some p-uses, test set include at least one path segment from every definition of every variable to every computational use of that definition; if there are definitions of variables that are not covered by the above direction, and then add predicate use test cases are required to cover every definition.
- *All definitions:* In this, test set includes every definition of every variable be covered by at least one use of that variable, (be that use the computational use or predicate use).
- *All p-uses:* In All p-uses, test set include at least one path segment from every definition of every variable to every predicate use of that definition; if there are

definitions of variables that are not covered by the above direction then leave them.

- *All c-uses:* In All c-uses, test set include at least one path segment from every definition of every variable to every computational use of that definition; if there are definitions of variables that are not covered by the above prescription then, leave them.

1.1.4. Automation of Software Testing

Software testing is very labor-intensive and expensive; it accounts for approximately 50% of the cost of a software system development. If the testing process can be automated, the cost of developing software will be reduced significantly. The most critical component of the testing is the generation of test cases. Figure 1.6 shows the Steps for the automation of software testing to validate and verify the requirement and generate test cases.

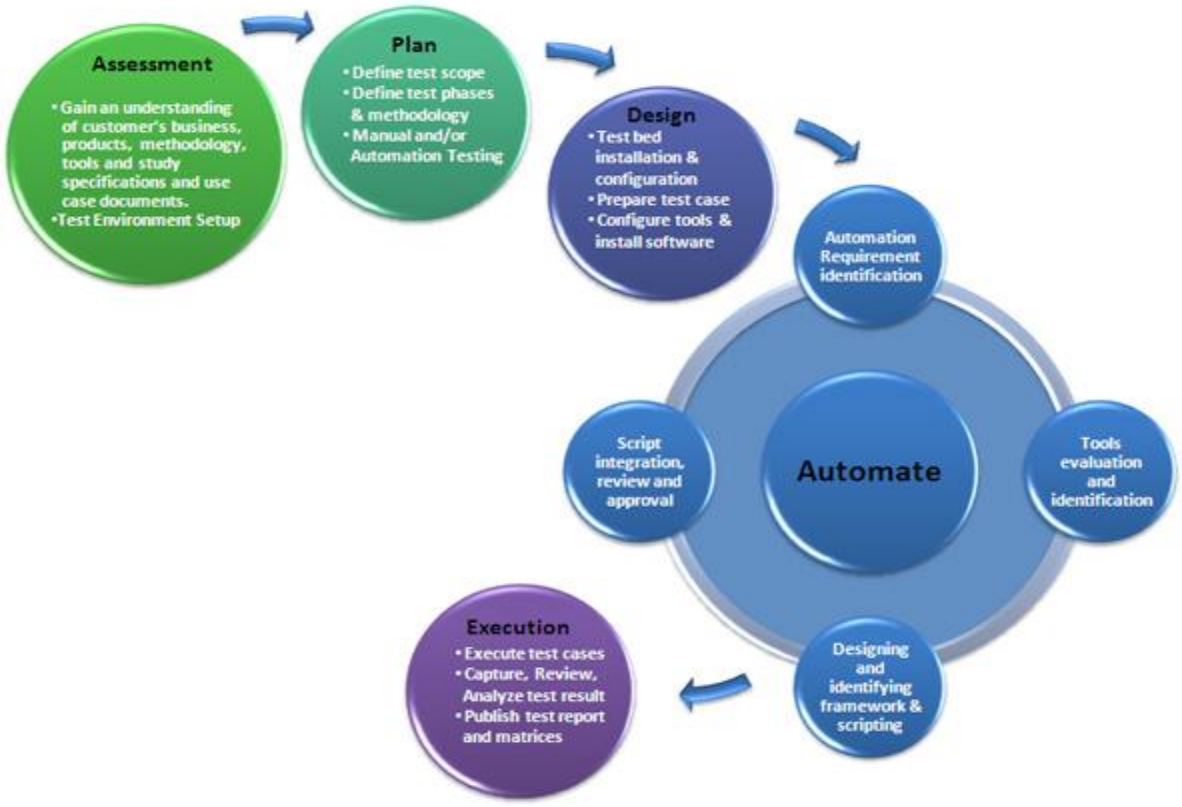

Figure 1.6: Steps for automation of software testing [10]

1.2. Slicing

Slicing may be referring as process or strategy to break body of information into smaller parts to examine it from different viewpoints that will yield more information so that researcher can understand it better. The term is also used to mean the presentation of information in a variety of different and useful ways. In the area of computer science research and software development slicing can be used in two broad fields. First is code based slicing and second one Model based slicing.

1.2.1. Code based slicing

In computer programming, code based slicing is the computation of the set of programs statements that may affect the values of variables at some point of interest, referred to as a slicing criterion. Basically program slicing [11] [12] [13] [14] [15] is a technique for focusing on certain aspects of a program's behavior and removing all other parts of code that are not concerned with this behavior. Alternatively, we can state that slicing is a technique to identify program sections relevant to a slicing criterion.

Weiser [13], [15] was the first to introduce the concept of a static slice. A static program slice s, of a program p, is constructed with respect to a slicing criterion (v, i), where v is a set of variable identifiers and 'i' is a program point. Statements in p that cannot affect the values of V when the next statement to be executed is at point i may be removed from p to form s. In contrast, Korel and Laski [16] defined a dynamic program slice that is computed with respect to a specific input in V and is only valid for that input. A dynamic slicing criterion was defined to consist of a set of variables, a line number in the program code and a sequence of input values.

Firstly the main application of program slicing is debugging [12][13][10] where if a program computes an erroneous value for some variable x at some program point, the bug is likely to be found in the slice with respect to x at that point. Later on slicing have been used in other applications also like parallelization, program differencing and integration, software maintenance, testing [17, 18, 19. 20], reverse engineering, and compiler tuning.

```
1    read (n) ;                  read (n) ;
2    i : = 1 ;                   i : = 1 ;
3    sum : = 0 ;
4    product : = 1 ;             product : = 1 ;
5    while i < = n do            while i < = n do
     Begin                       Begin
6    sum : = sum + i ;
7    product : = product * i ;   product : = product *
                                 i ;
8    i : = i +1                  i : = i +1
     end ;                       end ;
9    write (sum) ;
10   write (product)             write (product)
```

Figure (a) Figure (b)

Figure 1.7**: (a)** An example program. **(b)** A slice of the program w.r.t. criterion (10, product) [11].

Figure 1.7 (a) shows an example program which asks for a number n, and computes the sum and the product of the first n positive numbers. Figure 1.7 (b) shows a slice of this program with respect to criterion (10, product). As can be seen in the Figure, all computations involving variable sum have been `sliced away.

1.2.2. Model Based Slicing

Model Based slicing is a decomposition technique to extract and identify relevant model parts (or fragments) or related elements across diverse model views. In this context, a slicing technique for software architectures needs to take into account various classifiers, their relations and objects and their interactions. This is due to the fact that a UML metamodel extends support to describe structural and behavioral aspects of an architecture. For instance, the structural models (e.g. class diagrams) are used to describe various relations among classes, such as aggregation, association, composition, and generalization/specialization.

On the other hand, the behavioral models (e.g., communication and sequence diagrams) are used to depict a sequence of actions in an interaction, through which a use case is

realized. Traditional slicing is usually performed solely based on data and control dependency relationships existing among program statements. However, in architectural slicing of UML design models, several other types of model relations, such as class-class, class-operation, operation-operation, class-object, object-object, etc need to be taken into account [21].

In early stage of development of Model based slicing techniques , slicing has been applied to state machines [22] where similar benefits as those listed above for program slicing are claimed. State machine slicing is an example of applying slicing to a model of a system rather than to the system implementation. However, system models such as those represented in terms of the UML-family of languages are much more complex than state machines (and contain state machine sub-languages). Several approaches and attempts have been made at slicing UML class diagrams. The approach in [24] describe context-free slicing of UML class models where the issue of context is defined to be object location, which is a dynamic property of the scenario therefore context free slicing is a static slice of a structural model. As noted in [24] the criteria used for slicing a model is more complex than that used in program or state machine slicing since there are more types of elements and relationships; they note that OCL (object constraint language) should be used to express the slicing criteria. A similar approach is used to modularize the UML meta-model into collections of components that are relevant to the different UML diagram types in [23] although the predicate used to determine the slicing criteria is fixed in terms of traversing the meta-model elements starting with a collection of supplied classes. Class models are sliced together with OCL invariants in [25] thereby reducing the state-space explosion that would otherwise occur when using a model-checker (in this case Alloy) to verify a class-model.

UML statecharts can be sliced as described in [26] [19] [28] although these approaches do not generalize the results to include other parts of the UML language family. Both static and dynamic aspects of UML can be combined and sliced as described in [17] [18] where class and sequence diagrams are merged into a single representation (a model dependency graph MDG) that can be subsequently sliced to show partial dynamic and structural information resulting from criteria containing both structural and dynamic

constraints. Slicing UML sequence diagrams in order to generate test cases is described in [21] [27].

UML sequence diagrams (or scenarios) are essentially underspecified executions of a program. They can be used as slicing criteria for programs as described in [20] where the scenario significantly limits the scope of the slice.

1.3. Unified Modeling Language (UML)

UML is a standardized, general-purpose modeling language in the field of software engineering. It includes a set of graphic notation techniques to create visual models of object-oriented software-intensive systems. UML combines techniques from data modeling (entity relationship diagrams), business modeling (work flows), object modeling, and component modeling. It can be used with all processes, throughout the software development life cycle, and across different implementation technologies [29].

UML offers a standard way to visualize a system's architectural blueprints that include activities, actors, business processes, database schemas, components, programming language statements, Reusable software components [30].

UML has synthesized the notations of the Booch method, the Object-modeling technique (OMT) and Object-oriented software engineering (OOSE) by fusing them into a single, common and widely usable modeling language [29]. UML aims to be a standard modeling language which can model concurrent and distributed systems.UML models can be automatically transformed to other representations (e.g. Java) by means of QVT-like (Query/View/Transformation) transformation languages.

UML is not a development method by itself; however it was designed to be compatible with the leading object-oriented software development methods of its time (for example OMT, Booch method). Since UML has evolved, some of these methods have been recast to take advantage of the new notations (for example OMT), and new methods have been created based on UML, such as IBM Rational Unified Process (RUP). Others include abstraction method and dynamic systems development method.

1.3.1. Modeling

It is important to distinguish between the UML model and the set of diagrams of a system. A diagram is a partial graphic representation of a system's model. The model also contains documentation that drives the model elements and diagrams (such as written use cases).

UML diagrams represent two different views of a system model [30]:

- Static view: emphasizes the static structure of the system using objects, attributes, operations and relationships. The structural view includes class diagrams and composite structure diagrams.
- Dynamic view: emphasizes the dynamic behavior of the system by showing collaborations among objects and changes to the internal states of objects. This view includes sequence diagrams, activity diagrams and state machine diagrams.

UML models can be exchanged among UML tools by using the XMI interchange format.

1.3.2. Diagrams overview

UML 2.2 [31] has 14 types of diagrams divided into two categories. Seven diagram types represent *structural* information, and the other seven represent general types of *behavior*, including four that represent different aspects of *interactions*. These diagrams can be categorized hierarchically as shown in the following class diagram Figure 1.8:

UML does not restrict UML element types to a certain diagram type. In general, every UML element may appear on almost all types of diagrams; this flexibility has been partially restricted in UML 2.0. UML profiles may define additional diagram types or extend existing diagrams with additional notations. In keeping with the tradition of engineering drawings, a comment or note explaining usage, constraint, or intent is allowed in a UML diagram.

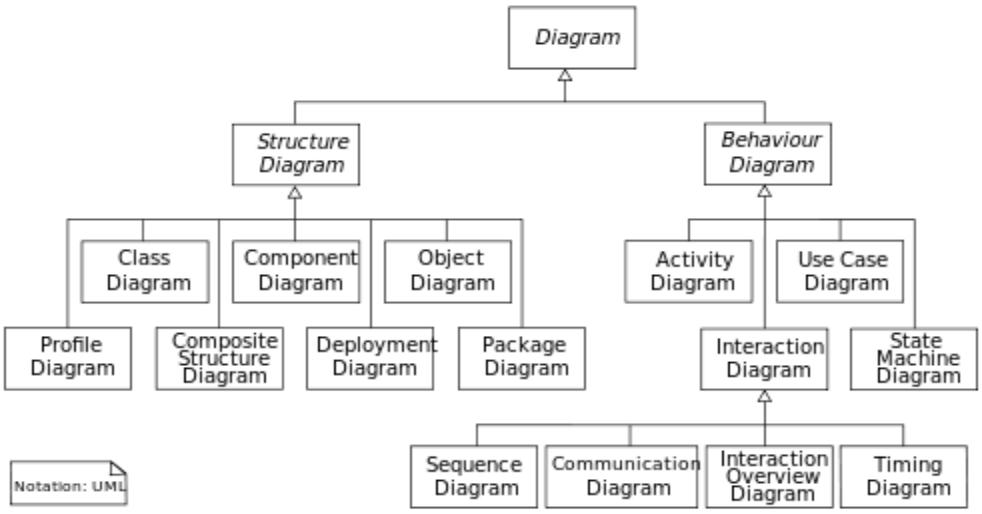

Figure 1.8: Overview of UML diagram Classification [29] [30] [31]

1.3.2.1. Structure diagrams

Structure diagrams emphasize the things that must be present in the system being modeled. Since structure diagrams represent the structure, they are used extensively in documenting the software architecture of software systems [30].

- Class diagram: describes the structure of a system by showing the system's classes, their attributes, and the relationships among the classes.
- Component diagram: shows how a software system is split up into components and shows the dependencies among these components.
- Composite structure diagram: tells the internal structure of a class and the collaborations that this structure makes possible.
- Deployment diagram: describes the hardware used in system implementations and the execution environments and artifacts deployed on the hardware.
- Object diagram: shows a complete or partial view of the structure of an example modeled system at a specific time.
- Package diagram: defines how the system is split up into logical groupings by showing the dependencies among these groupings.

- Profile diagram: operates at the metamodel level to show stereotypes as classes with the <<stereotype>> stereotype, and profiles as packages with the <<profile>> stereotype. The extension relation (solid line with closed, filled arrowhead) indicates what metamodel element a given stereotype is extending.

1.3.2.2. Behavior diagrams

Behavior diagrams emphasize what must happen in the system being modeled. Since behavior diagrams illustrate the behavior of a system, they are used extensively to describe the functionality of software systems [30].

- Activity diagram: illustrates the business and operational step-by-step workflows of components in a system. An activity diagram shows the overall flow of control.
- UML state machine diagram: describes the states and state transitions of the system.
- Use Case Diagram: shows the functionality provided by a system in terms of actors, their goals represented as use cases, and any dependencies among those use cases.

1.3.2.3. Interaction diagrams:

Interaction diagrams, a subset of behavior diagrams, emphasize the flow of control and data among the things in the system being modeled [30]:

- Communication diagram: shows the interactions between objects or parts in terms of sequenced messages. They represent a combination of information taken from Class, Sequence, and Use Case Diagrams describing both the static structure and dynamic behavior of a system.
- Interaction overview diagram: provides an overview in which the nodes represent communication diagrams.
- Sequence diagram: shows how objects communicate with each other in terms of a sequence of messages. Also indicates the lifespan of objects relative to those messages.

- Timing diagrams: a specific type of interaction diagram where the focus is on timing constraints.

1.3.3. UML Modeling Tools

The most well-known UML modeling tool is IBM Rational Rose. Other tools are Visual Paradigm for UML, Dia, ArgoUML, BOUML, Enterprise Architect, MagicDraw UML, Modelio, PowerDesigner, Rational Rhapsody, Rational software architect, StarUML and Umbrello. Some of the popular development environments also offer UML modeling tools, e.g. Eclipse, NetBeans, and Visual Studio.

1.4 Sequence Diagram

A sequence diagram is a kind of interaction diagram that shows how processes operate with one another and in what order. It is a construct of a Message Sequence Chart [29]. A sequence diagram shows object interactions arranged in time sequence. It depicts the objects and classes involved in the scenario and the sequence of messages exchanged between the objects needed to carry out the functionality of the scenario. Sequence diagrams are typically associated with use case realizations in the Logical View of the system under development. Sequence diagrams are sometimes called event diagrams, event scenarios, and timing diagrams [13, 32].

UML Sequence diagrams capture time dependent (temporal) sequences of interactions between objects. They show the chronological sequence of the messages, their names and responses and their possible arguments. A sequence diagram has two dimensions: the vertical dimension represents time, and the horizontal dimension represents different instances.

Normally time proceeds from top to bottom [21]. Message sequence descriptions are provided in sequence diagrams to bring forth meanings of the messages passed between objects. Sequence diagrams describe interactions among software components, and thus are considered to be a good source for cluster level testing. In UML, a message is a request for a service from one UML actor to another, these is typically implemented as

method calls. We assume that each sequence diagram represents a complete trace of messages during the execution of a user-level operation.

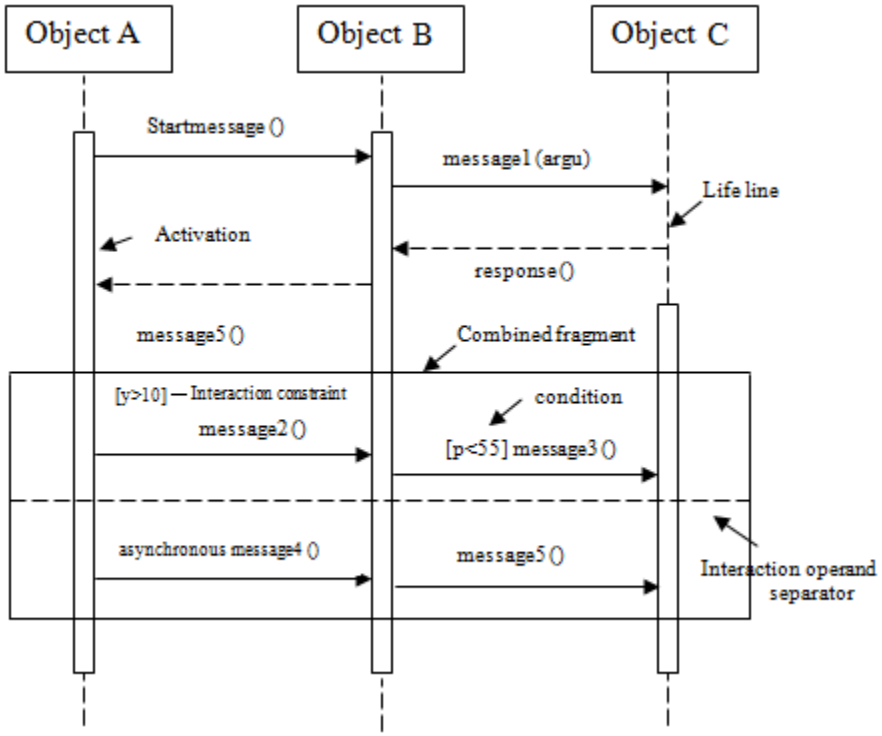

Figure 1.9: Sequence Diagram [21]

An example of a UML sequence diagram is shown in Figure. 1.9. The vertical dashed line in the diagram is called a lifeline. A lifeline represents the existence of the corresponding object instance at a particular time. Arrows between the lifelines denote communication between object instances using messages. A message can be a request to the receiver object to perform an operation (of the receiver). A synchronous message is shown with a filled arrowhead at the end of a solid line. An asynchronous message is depicted with an open arrowhead at the end of a solid line. Return messages are usually implied. We can explicitly show return messages using an open stick arrowhead with a dashed line as shown in Figure. 1.9. An object symbol shown with a rectangle is drawn at the head of the lifeline. Activation (focus of control) shows the period during which an instance is performing a procedure. The procedure being performed may be labeled in text next to the activation symbol or in the margin.

UML 2.0 also allows an element called note, for adding additional information to the sequence diagram. Notes are shown with dog-eared rectangle symbols linked to object lifeline through a dashed line as shown in Figure.9. Notes are convenient to include pseudo code, constraints, pre-conditions, post-conditions, text annotations etc. in sequence diagram. However, in our approach we restrict the notes to contain only executable statements. Messages in the sequence diagram are chronologically ordered. So we have numbered them based on their timestamps. Further, we have numbered the notes in an arbitrary manner.

In UML 2.0, a set of interactions can be framed together and can be reused at other locations. Different interaction fragments can be combined to form a combined fragment. A combined interaction fragment defines an expression of interaction fragments. A combined interaction fragment is defined by an interaction operator and corresponding interaction operands. Through the use of combined fragments, the user will be able to describe a number of traces in a compact and concise manner.

1.5 Extensible Markup Language (XML)

Extensible Markup Language (XML) is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable. The design goals of XML emphasize simplicity, generality, and usability over the internet. It is a textual data format with strong support via Unicode for the languages of the world. Although the design of XML focuses on documents, it is widely used for the representation of arbitrary data structures, for example in web services [33].

XML was created to structure, store, and transport information. XML is now most common tool for data transmissions between all sorts of applications. XML language has no predefined tags so it is easy to define user-define tags which are very important part while converting the UML diagrams into xml.

In the real world, computer systems and databases contain data in incompatible formats. XML data is stored in plain text format. This provides a software- and hardware-independent way of storing data. This makes it much easier to create data that can be

shared by different applications. One of the most time-consuming challenges for developers is to exchange data between incompatible systems over the Internet. Exchanging data as XML greatly reduces this complexity, since the data can be read by different incompatible applications.

The design goals of XML include, "It shall be easy to write programs which process XML documents." A variety of APIs for accessing XML have been developed and used, and some have been standardized [33].

1. Stream-oriented APIs accessible from a programming language, for example SAX
2. Tree-traversal APIs accessible from a programming language, for example DOM.
3. XML data binding, that provides an automated translation between an XML document and programming-language objects.
4. Declarative transformation languages such as XSLT and X Query.

1.6 Java API For Parsing

Document Parser class parses the XML file corresponding to a UML sequence diagram. We used the Document Object Model (DOM) API that comes with the standard edition of the Java platform, for parsing XML files. The package org.w3c.dom.*, provides the interfaces for the DOM. The DOM parser begins by creating a hierarchical object model of the input XML document. This object model is then made available to the application for it to access the information it contains in a random access fashion. This allows an application to process only the data of interest and ignore the rest of the document [34].

The DOM plug ability classes allow a programmer to parse an XML document and obtain an org.w3c.dom.Document object from a Document Builder implementation which wraps an underlying DOM implementation. In order to obtain a Document Builder instance, an application programmer first obtains an instance of a Document-Builder-Factory. The Document-Builder-Factory instance is obtained via the static new Instance method of the Document-Builder-Factory class. This method uses the Javax.xml.parsers.DocumentBuilderFactory system property to determine the DocumentBuilderFactory implementation class to load, instantiate and return. If the

javax.xml.parsers. DocumentBuilderFactory system property is not defined, and then a platform default DocumentBuilderFactory instance will be returned [34].

If in the parser DocumentBuilderFactory implementation class described by the javax xml parsers. Document Builder Factory property cannot be loaded or instantiated at runtime, a Factory Configuration Error is thrown. This error message must contain a descriptive explanation of the problem and how the user can resolve it.

The instance of DocumentBuilderFactory can optionally be configured by the application programmer to provide parsers that are namespace aware or validating, or both. These settings are made using the setNamespaceAware and setValidating methods of the factory. The application programmer can then obtain a DocumentBuilder implementation instance from the factory. If the factory cannot provide a parser configured as set by the application programmer, then a Parser Configuration Exception is thrown [34]. The DOM parser is called a DocumentBuilder, as it builds an in-memory Document representation. The DocumentBuilder is created by the javax.xml.parsers.DocumentBuilderFactory. The Document Builder creates an org.w3c.dom.Document instance, which is a tree structure containing nodes in the XML Document. Each tree node in the structure implements the org.w3c.dom.Node interface. There are many different types of tree nodes, representing the type of data found in an XML document. The most important node types are:

1. Element nodes that may have attributes
2. Text nodes representing the text found between the start and end tags of a document element.

By taking the simple example as consideration as shown in Figure.1.10. It can be explain how to parse XML content using DOM parser instance DocumentBuilderFactory. In the following example after creating the instance, parser parse the xml file “project.xml” and fetch the node list by getElementsByTagName. If the node list is not empty the parser parse the tree again correspond to that node and fetch its attribute and properties also.

```
public class Read
{
public static void main (String argv [])
{
int type_id;
   try {
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
DocumentBuilder db = dbf.newDocumentBuilder();
Document doc = db.parse (new File("project.xml"));//.openStream());

doc.getDocumentElement().normalize();
System.setOut(new PrintStream(new FileOutputStream("Test.txt")));
NodeList nodeList1 = doc.getElementsByTagName("InteractionLifeLine");
System.out.println("over all class tag:"+nodeList1.getLength());

if (nodeList1 != null && nodeList1.getLength() > 0)
{
     for (int k = 0; k < nodeList1.getLength(); k++)
               {
    Element ell = (org.w3c.dom.Element) nodeList1.item(k);
    System.out.println("Class-Name="+ell.getAttribute("Name")+" "+"Class-Id="+ell.getAttribute("Id"));
    System.out.println("\n");
               }
}}}}
```

Figure 1.10: DOM parser

Chapter 2

Literature Survey

2.1. Slicing UML models

Slicing UML Models is a process of decomposition to extract and identify relevant model parts or related elements across model that corresponds to user defined slicing criterion. In Model based slicing several types of model relations, and dependency such as class-class, class-operation, operation-operation, class-object, object-object, guard condition in sequence diagram , conditional predicate, control flow , data flow etc., need to be taken into account. In this work, sequence diagram has been taken into account and various approaches present till date for slicing UML diagram have been listed.

2.2. Slicing Techniques and Methodologies of UML Models

2.2.1 Using Dependency Graph

Dependency Graph is an intermediate representation step while slicing UML Models that can describe the various types of dependencies. Zhao [36] introduced the concept of architectural slicing which operates on architectural description of software system. According to the proposed architectural description there will be three types of dependencies. First is component-connector dependency where information flows from port (interface) of a component to role of a connector. Second type is connector component dependency in which information flow is from role of connector to port of component. Third type of dependency is additional dependency which can be used to represent a relation between two ports or roles within a component or connector. To compute the architectural slice a two phase algorithm has been proposed that works on SADG (software architectural dependency graph). First phase of algorithm is to compute slice 'Sg' over the SADG and second phase is to compute an architectural slice in which each element of ‘Sg’ is mapped to equivalent source code of description. As an extension of his previous work Zhao [37] introduced Architectural Information Flow Graph with

three types of information flow arcs: Component-connector, Connector-component, internal flow arcs to apply the slicing technique on software architecture precisely.

Wang et.al [38] presented a method for slicing hierarchical automata. The importance of Wang's algorithm is its ability to remove the hierarchies and concurrent states, which are irrelevant to the properties of the hierarchical automata. The given approach was based on representing the UML state charts by hierarchical automata for modeling dynamic aspects of software. The proposed method reduces the state space during model checking of UML state chart. The output slice proposed by technique is Extended Hierarchical automata (in which a set of dependence relations is specified after analyzing characteristics such as hierarchy, concurrency and synchronization.) instead of UML State chart models.

Wu and Yi et al. [39] developed an approach that comprises different class relationships to define dependence relations corresponding to the relations among classes. Based upon these set of dependency relations they construct a dependence graph of UML class diagram. Their proposed model can be used in two important applications: slicing the architecture and measurement of coupling between components. As their graph representation has been derived from class diagrams alone, usefulness is limited to understanding only the static aspects of a modeled system.

Kagdi et al. [24] proposed the concept of model slicing to support maintenance of software through the understanding, querying and analyzing large UML models. Kagdi developed model slices from UML class models. His approach was to extracts parts of a class diagram in order to construct sub models from a given model of a system. However, class models are lacking of explicit behavioral information and represent only structural behavior. For the purpose of model slicing they define a model 'M' as directed multi graph for finite set of elements, their set of relationships, and a function that maps elements to elements via a relationship.

Langehove [40] presents an algorithm for reducing the number of interference dependencies in state charts by using the concept of slicing with concurrent states. The proposed approach considers data dependencies from the definition and use of variables

that are common to parallel executing statements. The core idea of approach is to define happens-before relation between states and transitions of orthogonal regions to improving the degree of refinement in measurement of interference dependencies. He achieves this by exploiting the internal broadcasting mechanism and maintaining the state chart's execution semantics.

Bae et.al [23][41] proposed UML metamodel slicer to manage the complexity of UML metamodels which addresses to all UML diagram by modularizing metamodels into small metamodels. The proposed approach extracts diagram-specific metamodels from the UML metamodel because the diagram-specific metamodels consist of a considerable small number of elements and relationships. The UML Metamodel Slicer generates a metamodel, 'MMdt' by a given set of key elements 'KEdt'. The elements in 'KEdt' are used for identifying the model elements as slicing criteria which are relevant to the diagram type 'dt'. Figure 2.1 shows the relationships among various models and model elements in the data architecture for generic system.

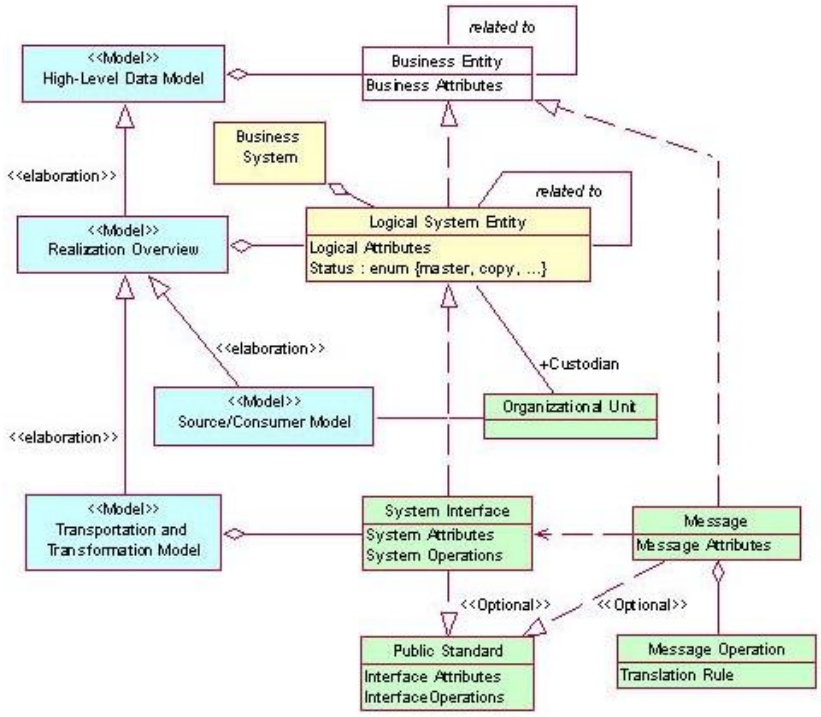

Figure 2.1 Metamodel for generic system [23, 41]

Sen et al. [42] presented an approach for pruning metamodels. The proposed pruner takes input slicing criteria, i.e. classes, operations, etc of the metamodel to slice the architecture. The pruner resulted into an output slice that satisfies all the structural constraints imposed by the input metamodel.

Lallchandani et al. [17] propose a technique for constructing dynamic slices of UML models using the integrated state-based information. In order to achieve this they proposed an algorithm Architectural Model Slicing through MDG Traversal (AMSMT) that first represents a model dependency graph to collect all the information about every dependency at different states of variables as shown in Figure 2.2. Researchers proposed an algorithm that generate the dynamic slices correspond to any slicing criteria by traversing the model dependency graph which holds all the dependency of variable. Such slices can be used for studying the impact of design changes, reliability prediction, understanding large architectures because it holds the object's state information. By using the same algorithm (AMSMT) researchers had implemented a prototype architectural slicing tool called SSUAM [43] to generate static slices for UML Architecture models. Later on, in another approach [44] they proposed a DSUAM algorithm which uses the MDG representation to compute dynamic slices. There slicing algorithm is based on traversing the edges in the MDG for any given slicing criterion. During MDG traversal, DSUAM identifies the relevant model parts from architecture. Algorithm has two phases. In first phase first phase, an MDG is constructed by extracting model information through static analysis of the UML structural and behavioral models. And in second phase MDG is traversed according to the given slicing criteria and produce desired chunk relative to condition.

Samuel and Mall [45] presented a scheme to generate slice and test cases with the help of edge marking dynamic slicing algorithm for activity diagrams. They used the flow dependency graph (FDG) which shows the dependencies among activities that arise during run time. The proposed technique uses the edge marking concept to mark the stable and unstable edges in FDG so that they can slice the graph according to slicing criteria from FDG correspond to conditional predicate present at each activity edge and can generate test cases according to them.

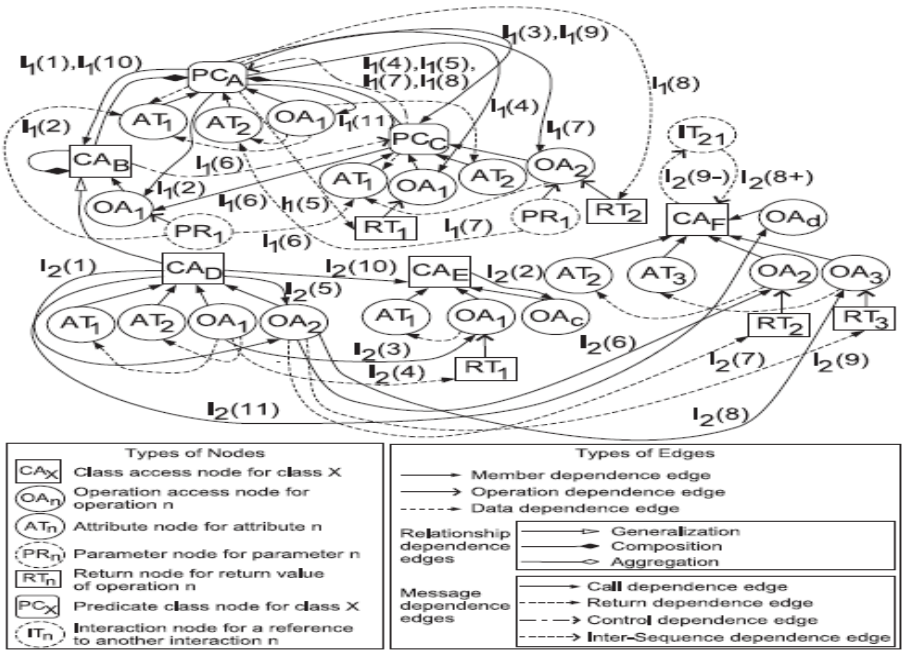

Figure 2.2 MDG of generic system [17]

Samuel et al. [21] presented a methodology to generate dynamic slices and test cases with the help of UML sequence diagram. In which Message dependency graph (MDG) is constructed which represent every message as node. To identify the conditional predicates associated with messages in a sequence diagram, slicer can create dynamic slice according to the criteria.

As an extension of their previous work to generate automated test cases according to the functionality of the system at a designing part of the SDLC, they proposed an approach [27] to use slicing technique on the UML Sequence diagram. Sequence Diagram can capture time dependent sequence of interaction between different objects and components. By analyzing these relations a proper functionality of the system can be visualized and captured to generate test cases for better verification. The way to generate test data in their proposed approach was to select conditional predicate from sequence diagram and make as a slicing criteria in the slicer while keeping all other variable constant while traversing the every node of sequence diagram until the solution is found. Figure 2.3 shows the MDG derived from sequence diagram. Where doted arrow denotes unstable edges and stable arrow denotes stable edges.

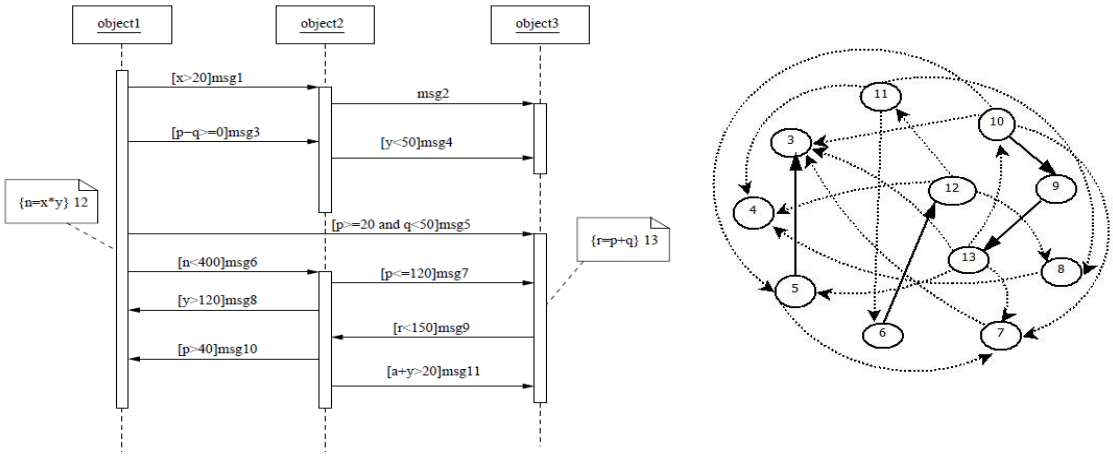

Figure 2.3 Example of Sequence Diagram and their corresponding MDG [21][27]

Noda et.al [46] proposed a sequence diagram slicing method to visualize the object oriented program's behavior. In order to achieve this, a tool has been proposed that named as 'Reticella' which is implemented as eclipse plug-in. The proposed tool take java program as input and after analyzing, fetch the static information and draw B-model tree. The slicer extracts a slice according to user define slicer criteria from graph and Drawer convert the data sequence slice into sequence diagram with the help of Quick sequence diagram editor.

Swain et.al [47] proposed an approach to generate test cases from UML interaction diagram by using the condition slicing. In their approach they identify the message guard condition from interaction diagram and use the condition slicing to generate test cases. In the proposed approach, they first build a message dependency graph from UML Interaction diagram and then apply the conditional slicing on a predicate node of the graph by considering guard condition of message flow as a slicing criterion to compute slices and to generate test cases.

J. Kim et.al [48] proposed an approach to address the hierarchy and orthogonality problems while tracing the data dependency in slicing of UML State machine diagram. They first, constructed a control flow graph (CFG) to track every transaction and parallel flow, and then they created a hierarchy graph that represent a parent-child relationship among regions, states and behaviors of states. By utilizing CFG and hierarchy graph researchers have generated dependency graphs that represent the related functionality.

Yatapanage et al. [49] focused their work on Model Checking as fully automated technique to reduce the size of model with the help of slicing. They used Behavior Tree dependency graph (BTDG) to capture all functional requirements and dependency between components and attributes. After creating the BTDG they use the slicing criterion consists of all state-realization nodes that update the state of one of the components or attributes mentioned in the temporal logic property to reduce the parallelism and size of models in order to improve verification time in model checking techniques.

2.1.2 Using Control and Data Flow

Control and Data Flow are the important aspect of system modeling or UML models that describe the nature of every component, their behavior, and working with other components and sequential pattern of interaction. Many researchers dedicated their work to slice the models or architecture of the system into desirable small chunks. Lano [26] defined that slicing can be carried out for UML state machines, using data and control flow analysis to remove elements of the machine that do not contribute to the value of a set of features in a selected state of the machine. The proposed technique of slicing by refactoring enables the models to be simplified and factored on the basis of features. Author also represents the pre and post condition relationship of the state during path predicate coverage. The notion of data and control dependency between states is simpler than the concepts of transition post-domination used by Korel. The proposed technique is non-termination insensitive, since the data-dependency calculation does not take account of cases where an infinite loop of states may arise, preventing the target state from being reached.

Korel et.al [50] dedicated their work on slicing the state based models, such as EFSMs (Extended Finite State Machines). As a result two types of slicing came to existence—deterministic and nondeterministic slicing. Their approach also includes a slice reduction technique to reduce the size of a computed EFSM slice by isolating the parts of the model that may contribute to faulty behavior. Their research of slicing techniques for state machines is thoroughly based upon control and data-flow analysis. To automate the slice

computation they proposed a tool that constitute of graphical editor, an EFSM executor and EFSM slicer. Figure 2.4 shows the EFSM model for ATM system.

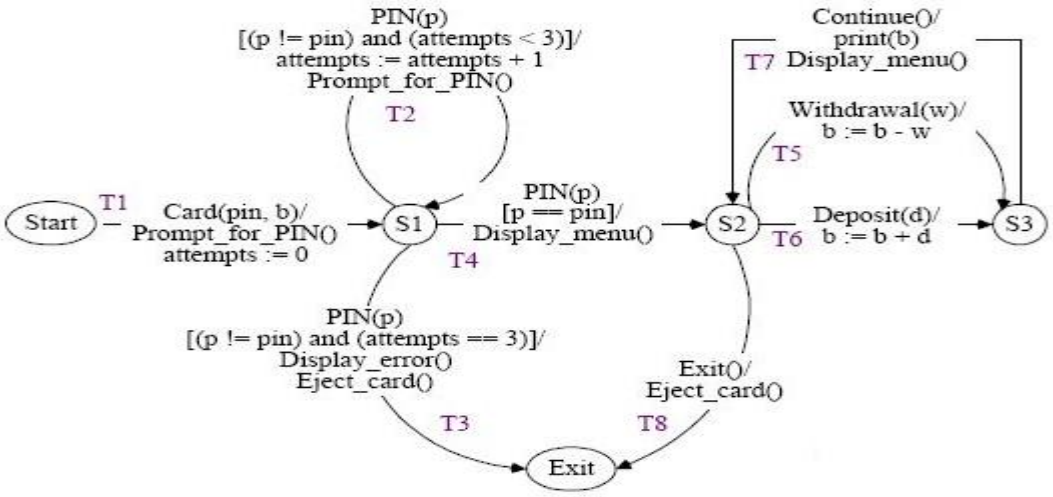

Figure 2.4 EFSM Model for ATM System [50]

Samuel et.al [51] proposed a schema known as 'Ctest' that automatically generate test cases from UML communication diagram. According to schema the first step of the approach is to construct communication tree from communication diagram on the basis of data flow and control flow. After selecting the predicate from tree, tool named UTG (UML behavioral Test case Generator) transform the predicate according to 'CTest' schema to find the test data. Communication diagram has been taken as input by tool in xml format. In this approach Document Parser class parses the XML file for the message name, arguments, sequence numbers and constructs the communication tree, while Test Data Finder uses the parsed information and finds the test data in the form of a string

Julliand et.al [52] proposed an approach based on domain abstraction for generating test cases on the basis of syntactic abstraction and variable elimination with the help of model slicing. In the proposed approach source model is taken as input with set of abstract variable then reduced by syntactic abstraction followed by semantically abstraction to generate abstract model from which symbolic tests are extracted according to selection criteria. They proposed three methods for identifying the relevant variable and generating abstract model. The first one is to consider data flow dependency only. Second one uses both data-flow and control-flow dependency. Third method is to use data flow and partial

control flow dependencies to find as much as possible strong relevant variables. Once the set of abstract variable 'Xa' is defined the next step is to define the Slice function that abstract the predicate P according to 'Xa'. The core idea of the work is to use the combination of syntactic and semantic abstraction or slicing to refine the result more precisely while generating the test cases.

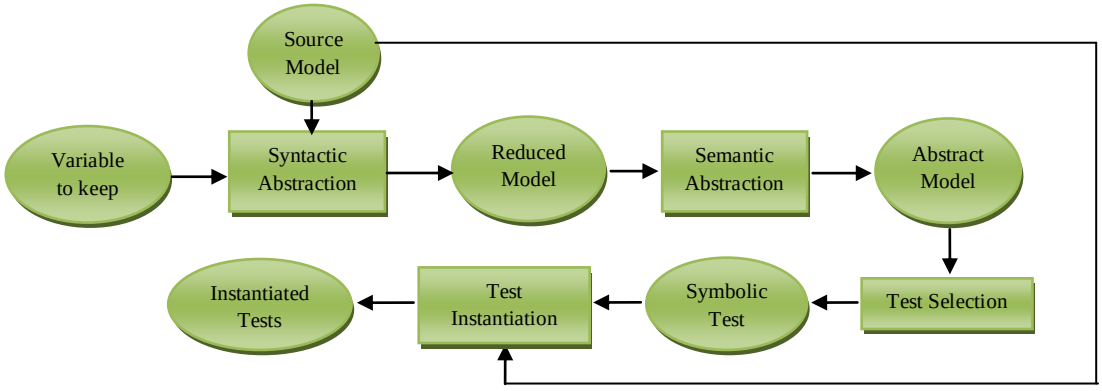

Figure 2.5 Overview of Process for B-Model Slicing & Predicate generation [52]

2.1.3 Using UML/OCL Constraints

OCL allows the definition of expressions on UML models, an expression that evaluates the true or false of class invariant, or constraint. OCL expression involves several objects from one or more classes of the model. To get a starting object, we can use the keyword 'self', which denotes an object of the context type or the method 'allInstances()', that can be used to access all objects of a given type, e.g., 'TT::allInstances()' returns a set of all objects of class TT. Given an object, OCL provides operators to read the values of its attributes and access the objects connected to it through associations (navigation). Combining these operators with arithmetic, logic and relational operators, iterators and user-defined query operations, it is possible to write complex constraints about UML models.

Shaikh et.al [25] proposed a verification technique to check the correctness of model with the help of slicing as shown in Figure 2.6. The proposed technique increases the scalability of verification by partitioning the original model into sub model. To define the binary association and inheritance relation, dependency graph and flow graph has been used in the proposed approach.

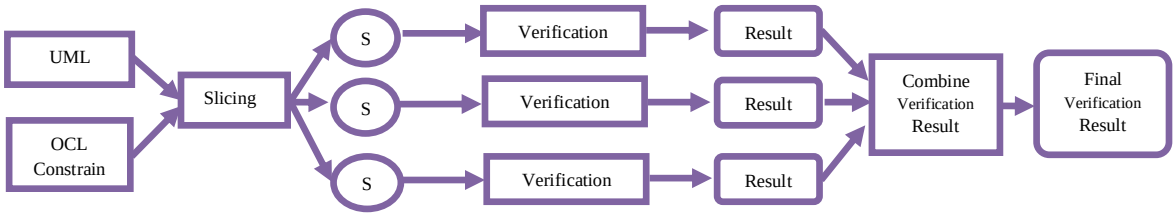

Figure 2.6 Process Model for Slicing using OCL Constraints [25]

To verify the satisfiablity of instances in models independently structure of class diagram and OCL Constraints has been taken as input. By which slicing can be done easily to decompose the model into sub models with the help of dependency graph to extract instances of models and correct component relations.

[53][54] Proposed a tool (UOST) to enable the efficient verification of UML/OCL Class diagram with the help of model slicing technique. The tool can verify the properties of the diagram with disjoint and non-disjoint sets of slicing. Tool take the class diagram as input in XMI format with text specified OCL constraints and break the file into several slices with the help of model slicing technique after parsing the file. Researcher used the eclipse solver to translate the slices into CSP and check the existence of solution to generate the respective object diagram for satisfiabile and unsatisfaible sub models with their specific invariants.

Sarna et al. [55] proposed an algorithm for automatic generation of test cases from sequence diagrams. They first transform UML sequence diagram into graphical representation named as SDG (Sequence diagram graph). They follow a graph based methodology with a depth first search algorithm to traverse the SDG and to generate test cases according to all message sequence path coverage criteria. To retrieve the information for a specification of input/output, pre and post conditions for test cases generation they use the use case template, class diagram and data dictionary and expressed in OCL.

2.1.4 Using Feature Based Criteria

Archer et al. [56] proposed a novel slicing technique on the feature model by taking cross-tree constraints into account with respect to set of features which are acting as

slicing criteria. The core idea of proposed algorithm is to compute proposition formula representing the set of configuration and rules and to apply propositional logic reasoning techniques to construct an FM (representing its hierarchy, variability information, feature groups and cross-tree constraints).

[57] also proposed the concept that how set of complementary set of operators like aggregate, merge and slice can provide practically and efficient support for separation of concerns from feature modeling. They defined that slicing process is both semantic and syntactic so they analyze the cross-cutting constraints to define the features that must be or cannot be sliced. In their proposed technique, the feature model and its cross-cutting constraints are first analyzed by transformation into predicates and then these predicates are transformed in a sliced feature model.

Hubaux et al. [58] proposed a slice feature diagram to design three different views of an input diagram to provide more flexibility to the configuration environment. The sliced diagram does not keep the same structure as the input diagram. The proposed approach does not consider cross-cutting constraints and is thus syntactic. The important property of the approach is that it should always lead to valid configurations but the problem can arises in the approach when features belong to more than one view or, more generally, when the selection of a feature in one view affect the selection of another feature in a concurrent view.

2.1.5 Using Model Languages

Kim [59][60] introduced the slicing technique called dynamic software architecture slicing (DSAS). In the 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, Kim's work is very efficient there because it’s able to generate a smaller number of components and connectors in each slice according to slicing criteria.

In this approach software architecture is first designed by using ADL (Architecture description language) and later on mapped onto program statement as executable architecture. Dynamic slicer takes slicing criterion as input, and reads the ADL source code of the architecture to identify the information of component and connector along

with the event names used in the ADL and parameter names combined with those events. The proposed algorithm filters out the events that are not relevant and passes only those which are relevant to slicing criterion and generate resulting software architecture slice as shown in Figure 2.7.

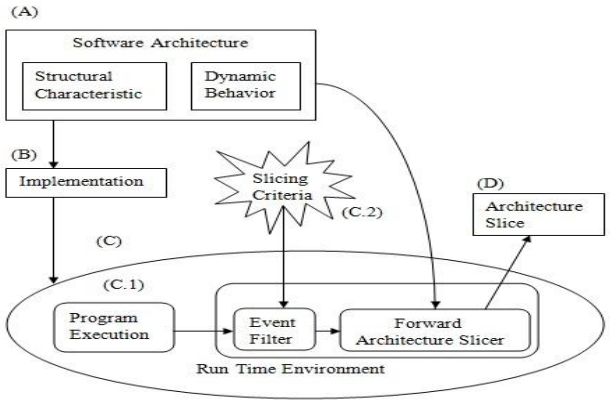

Figure 2.7 Dynamic Software Architecture Slicing Methodology [59]

Falessi et al. [61] used the concept and technique of model slicing to automate the safety inspection of system. In order to achieve this, a tool named "Safe Slicer" has been proposed that use model based slicing to enables automatic extraction of the safety-related slices (fragments) of design models. They proposed and elaborate a design methodology which ensures the traceability of links required for automated slicing. The methodology and the slicing algorithm proposed by these researchers are the basis for the Safe Slice tool. Evaluation conducted by this tool indicates that the use of design slices substantially reduces the amount of information that needs to be inspected.

Lano et al. [62] defined the technique for slicing of UML model using Model Transformation, particular for restriction of model to those parts which specifies the properties of subset within. The proposed technique use class diagrams, individual state machines and communicating sets of state machines to perform slicing of UML Model. Researcher used the client-supplier relation between different classes to form a tree structure. According to proposed technique slicing will be carried out upon class invariants and operation pre and post conditions by considering the predicates P. To slice the behavior and communicating state machines they define criteria's for a slice 'S' of a state machine 'M' are S <syn M, if S has fewer elements than M. S =sem M. This means

that any analysis which concerns the value of the slice features V in the selected state s, over all paths to this state, can be performed on the slice S, and the result will also apply to state model M. their techniques focused on structure-preserving and amorphous slicing of class diagrams and state machines by using Model transformations to perform slicing, each transformation will satisfy the <syn and =sem relations, resulting in a slice which also satisfies these relations compared to the original model.

Zoltán et.al [63][64] proposed dynamic backward slicing of model transformations technique with respect to program slicing. To slice the models they used model transformation language as a core of technique with the help of Dynamic Backward slicing by considering the Execution traces of program to generate final slice. The proposed technique take three inputs, the model transformation program, the model on which the MT program operates and the slicing criterion and generate the output as transformation slices and model slices as shown in Figure 2.8. Transformation of models into MT Language is done by three consecutive processes, Graph pattern, Graph transformation rules and control language on VIATRA2 platform. The algorithm keep the record of execution traces during graph pattern calls and GT rules, to provide traceability information between source and target models and uses these traces to slice MT programs and models simultaneously according to slicing criteria.

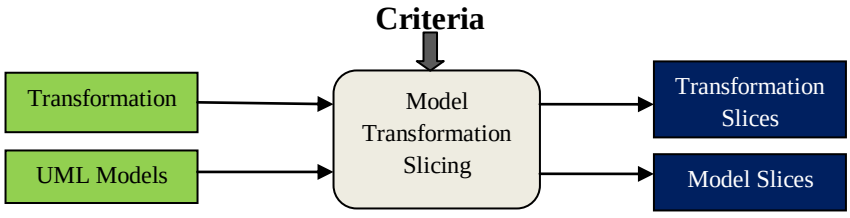

Figure 2.8 Slicing Process with Model Transformation [63, 64]

Blouin et.al [65] [66] proposed a DSML (Domain Specific Model Language) 'Kompren' to model the model slicers for particular domain. Kompren refers to the selection of the set of classes and relations from the input metamodel expressed using an object-oriented meta-language. Using Kompren, the MSM (Model Slicer Model) specifies the type of slicing criteria among the classes of input metamodel by domain expert that uses the ‘Ecore’ to describe the structure of metamodel and ‘Kermeta’, an action language to

specify the behavior of slicer. Kompren's compiler processes the MSM and automatically generates an actual model slicer function (MSF).This MSF take as input the slicing criteria, options, constraints defined in MSMs by Domain user to produce the Model Slices. Figure 2.9 shows the basic steps involved in 'Kompren' (Domain specific model language).

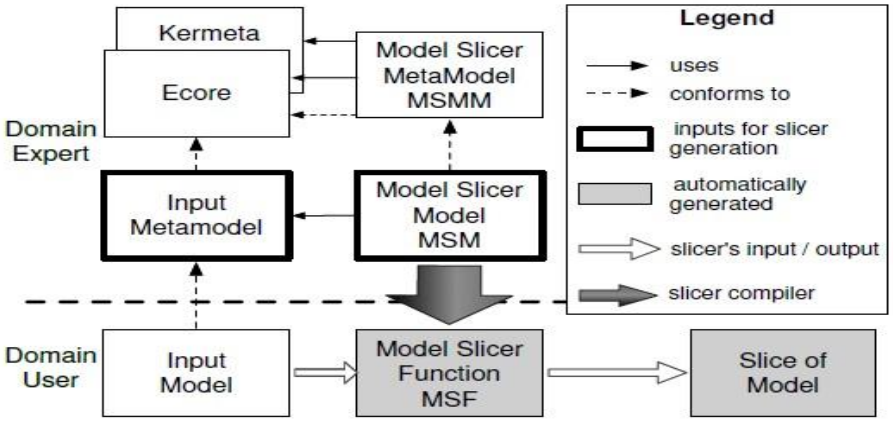

Figure 2.9 Overview for Modeling Model Slicers with Kompren [66]

Table 2.1 Tools for Model based Slicing

S.No	Year	Tool Name	Technique Used
1	2003	EFSM Slicing Tool	Control and Data flow analysis.
2	2007	UTG	Data Flow and Control Flow dependency, Communication Tree.
3	2008	SSUAM	Model Dependency Graph.
4	2008	UML Slicer	MetaModel Diagram, Key Elements.
5	2009	Reticella	B-Model dependency Graph.
6	2011	Archlice	Model Dependency Graph.
7	2011	Safe Slicer	System Model Language, Traceability Links and Rules.
8	2012	UOST	UML + OCL Constraints.

Table 2.2 Comparison of Model based Slicing Approaches/Techniques/Tools

Approach/Techniques/ Tools	**Slicing Process**		**Model**	**Usage**
	Syntactic	Semantic		
Feature model slicing [56,57,58]	✓	✓	Feature model	Separation of concerns
UML slicer [23, 41]	✓	×	UML metamodel	Modularization
UML statechart [19, 26, 28, 38]	✓	✓	UML state charts	Reactive system, Model checking
Safe slicer [61]	✓	×	System Model Language	Safety
Domain abstraction [52]	✓	✓	Control & Data flow graph	Abstraction of model
Context free UML slicing [24]	✓	×	UML class diagram	Sub model extraction
Dynamic software architecture slicing [59, 60]	✓	✓	Architecture description language	Architectural slicing
EFSM slicing [50]	✓	×	State based models	Size reduction
DSUAM [44]	✓	✓	UML	Separation of concerns
UML activity diagram [45]	✓	✓	Activity diagram	Test case generation
UML/OCL slicing [25, 53, 54]	✓	✓	UML/OCL	Verification
Metamodel pruning [42]	✓	×	Ecore based	Static analysis
Model transformation [62,63,64]	✓	✓	Graph and UML diagram	Program slicing
Behavior tree slicing [49]	✓	✓	Behavior tree	Model checking
Domain specific model language [65, 66]	✓	✓	UML metamodel	Dynamic model slicing

Chapter 3

Gap Analysis and Problem Statement

3.1 Gap analysis in Existing Work

Based on the literature review of program slicing techniques, UML, Object oriented testing, test case generation and slicing of UML diagrams, following gaps have been identified:

- No consolidate technique have been developed to extract the point of interest from architecture of software to ease the software visualization [23, 25, 43, 44, 45, 51].
- Model based test case generations have not been giving a thorough thought for procedural as well as concurrent programming system [67, 68, 69].
- There is need to simulate the environment for generating test set from architectural specification of a problem statement [12, 17, 18, 28].
- Till date no technique using conditional predicate for finding out a relative slice from sequence diagram has been designed [21, 27, 46, 47, 55].

3.2 Problem Statement

After reviewing the literature of software testing techniques, slicing techniques, software visualization and unified modeling language, it has been analyzed that slicing UML diagrams is one of the major area in which work can be extended for various constructs like sequence diagram, state transition diagram, activity diagram, class diagram, etc. Program slicing is a technique for extracting point of computer programs by tracing the program control and data flow related to particular data item. This has been categorized into code based and model based slicing.

It has been thoroughly analyzed that for the process of slicing sequence diagram no consolidate technique have been developed to extract the point of interest from architecture of software to ease the software visualization that uses conditional predicate for finding out a relative slice.

Chapter 4

Methodology

The proposed work addresses the slicing of sequence diagram to ease the software visualization by using conditional predicate for finding relevant slices.

In the proposed methodology, following steps has been followed:

1. Generation of UML (Sequence) diagram of/from a particular requirement specification.
   1.1. Visual paradigm for UML, Rational rose and Magic-draw, *etc* can be used to generate the UML diagrams.

2. Create XML from the specified UML diagram (Sequence diagram).
   2.1. Visual paradigm for UML 10.0 version provides the in-built functionality to export the diagrams into XML format.

3. Document Object Model (DOM) parser for parsing XML code and generating an output file (with .txt extension) having Object name, identifier, message name, message to & fro information.
   3.1. Java API DOM is used to parse the XML code file generated in step 2.
   3.2. DOM parser uses the function DocumentBuilderFactory ( ) to create the instance of the class to parse the file.
   3.3. DOM parser will generate a txt file having information regarding object name and its identifier. This file also contains the information related to all the messages and the objects among which the message is floating.
   3.4. All the information generated by parser will be stored in separate .txt file.

4. Passing file obtained from step 3 and slicing criteria to a .java program (which act as slicer) for getting the relative/required chunk of information in a separate .txt file.
   4.1. Slicer will take .txt file generated in step 3 as input.

4.2. Slicer will ask user to define the slicing criteria at run time to generate the chunk/slice as per specified requirements.

4.3. Computed slices will be store in separate .txt file which holds the information of messages, their guard condition and objects id's among which messages are being passed.

5. Changing object id with relative object name among which message is passing so that information can be retrieved easily (this step will only deal with sliced part).

   5.1. To ease the retrieved of information objects id's will replaced by their corresponding object name (in the file retrieved from step 4.3).

   5.2. All the information will store in separate .txt file which holds the information of messages and the objects name (among which they are communicating relative to user defined slicing criteria).

6. Passing txt file as obtained from step 5, to a .java program so that it can be converted into input file format for *Quick Sequence Diagram Editor*.

7. Tool will generate the final and relatively small sequence diagram.

   7.1. Tool will take the input format defined at step 6 as input to convert into its equivalent diagram.

   7.2. Refined slice (small sequence diagram) will be generated as final output according to slicing criteria as per requirement to ease the software visualization.

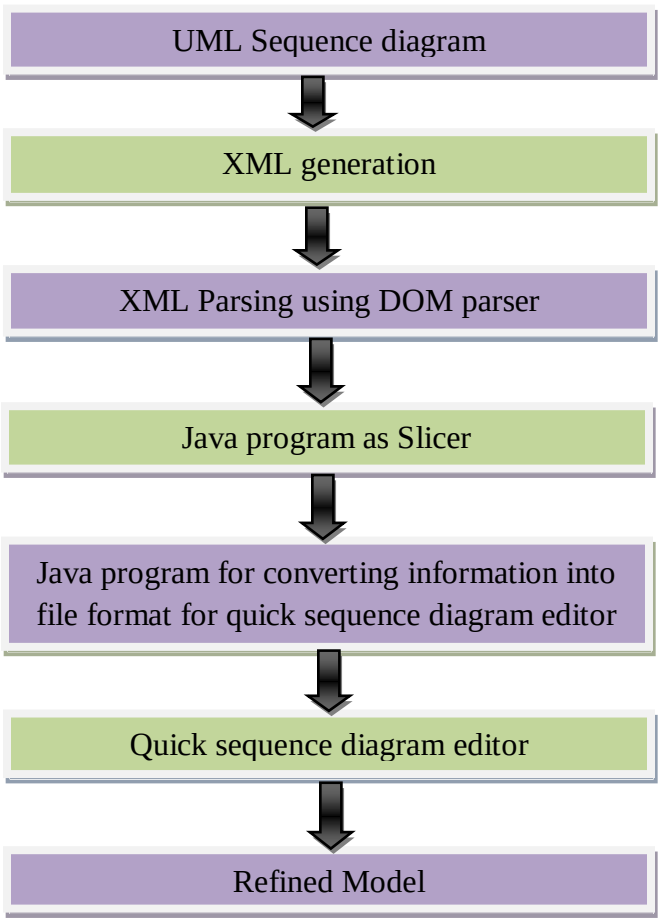

Figure 4.1: Overview of Proposed Methodology

Chapter 5

Implementation

The implementation part has been divided into various subsections. The first part is to draw the UML diagram and convert into XML file. For this existing tools such as Magic-draw, Magic-draw R convertor, Rational Rose with XML API, Visual Paradigm for UML [67] etc can be used. The tools are. In this implementation Visual Paradigm for UML 10.0 version [67] has been used to convert the sequence diagram into XML.

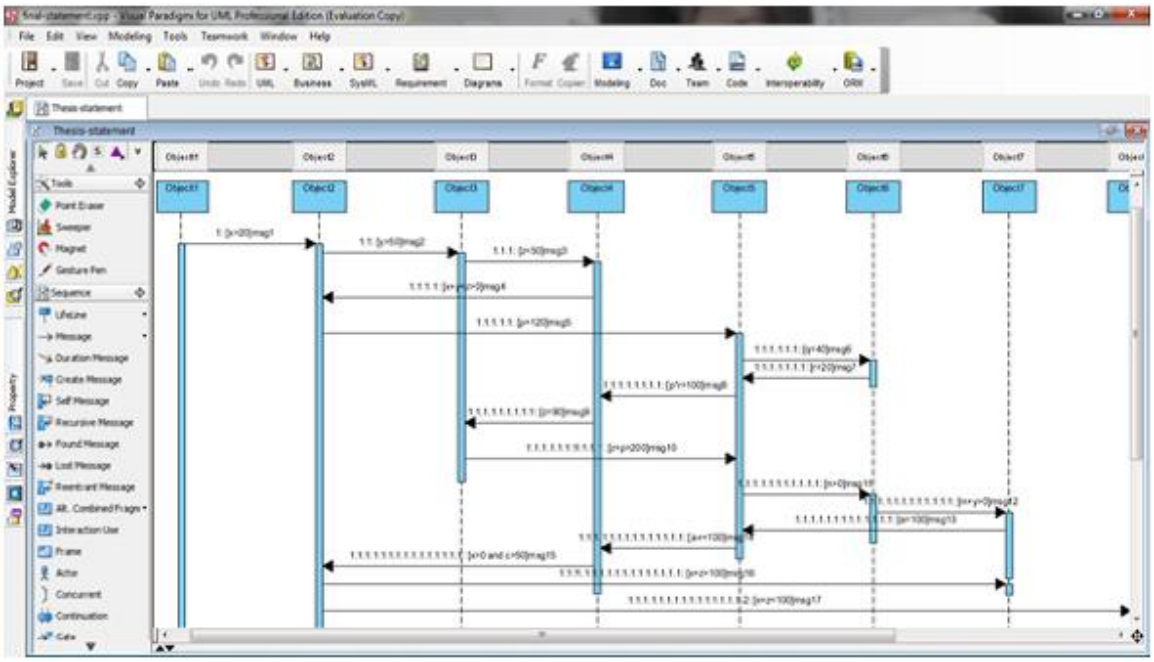

Figure 5.1: Designing sequence diagram using visual paradigm [67]

In Figure 5.1, UML sequence diagram (which tells us the functional behavior of the system) has been taken as an example to describe the scenario. In this example multiple objects are interacting and sending message to others at different time and scenario. These objects use variable and constants as guard condition to pass messages to others. Figure 5.2 will serve as an example of sequence diagram for implement of proposed methodology. In this example 10 objects have been taken into account that interacting with each other at different time and states during their life cycle using identifiers as guard condition.

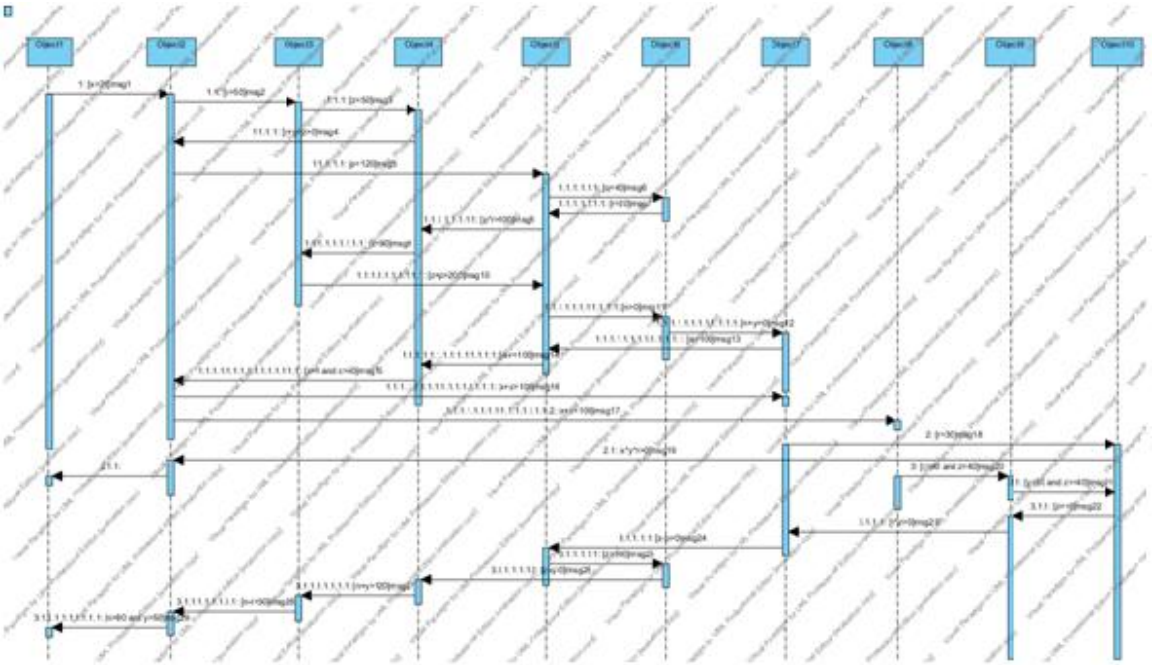

Figure 5.2 Example Sequence diagram

Next is to convert that UML diagram into XML by using the Visual Paradigm for UML 10.0 version as shown in Figure 5.3.

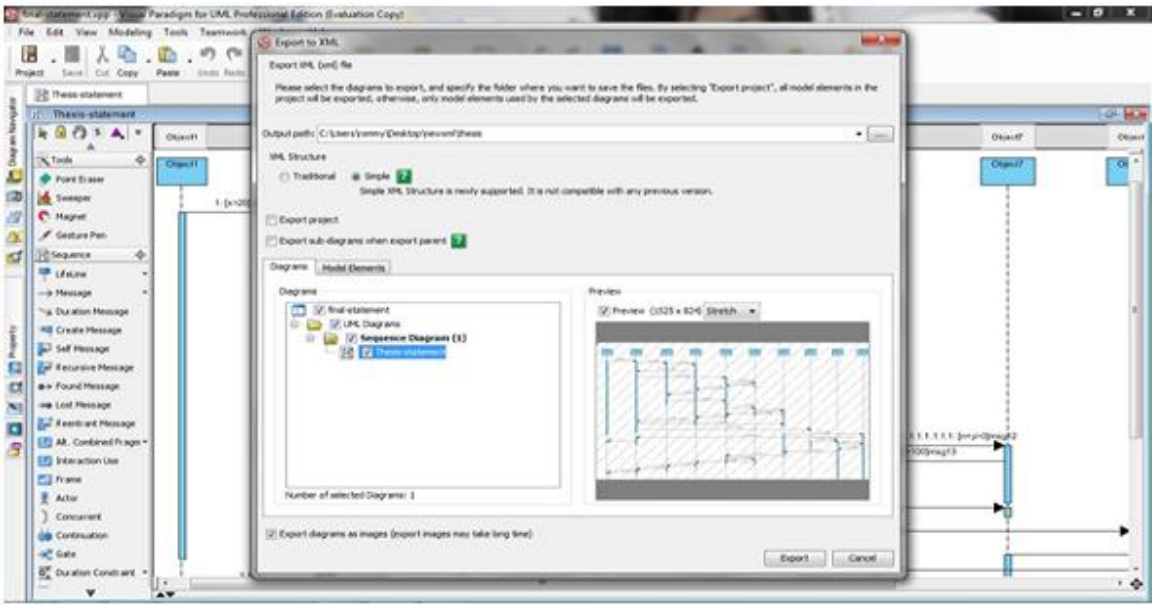

Figure 5.3: Exporting Sequence diagram for generating its specific XML

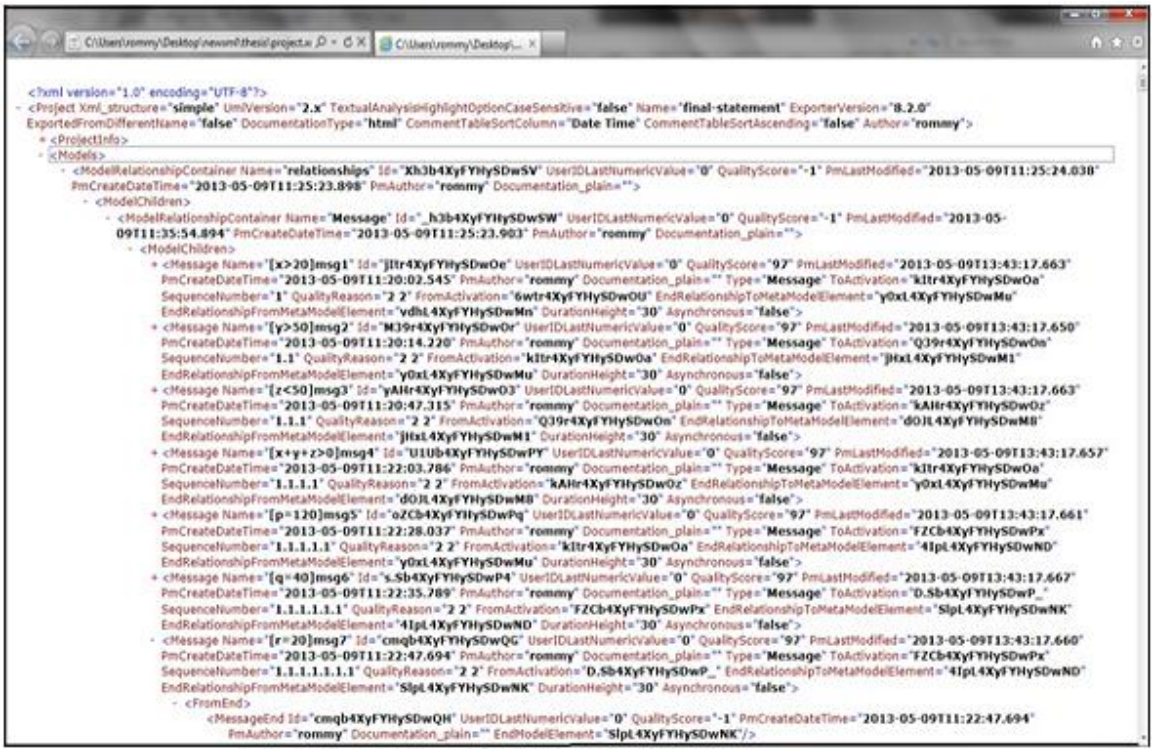

Figure 5.4: XML file of Sequence diagram

As shown in Figure 5.4, XML constitutes all the information about sequence diagram like the object name with distinct ids, the messages which they are using to transfer the data or to call the object of other classes, their attributes, etc. The purpose of converting the UML diagram into XML file is confirming platform independency.

Next step is to parse the XML file to fetch the relevant information. Figure 5.5 shows the parsing program which is using java API Document Object Model (DOM) to parse the XML file. Here in this example the xml file of sequence diagram is present. By parsing the file we can get the name of the objects and their Ids, messages which are call and pass by these objects, their relation with each other, guard conditions which they are using to traverse in sequence diagram through different objects and their respective Ids from which message is coming and to which message is going as shown in Figure 5.6.

```
import java.io.File;
import org.w3c.dom.*;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;
import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.NodeList;
import org.xml.sax.*;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.PrintStream;

public class Read
{
public static void main (String argv []){
int type_id;
   try {

         DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
        DocumentBuilder db = dbf.newDocumentBuilder();
        Document doc = db.parse (new File("project.xml"));
doc.getDocumentElement().normalize();
System.setOut(new PrintStream(new FileOutputStream("Test.txt")));
NodeList nodeList1 = doc.getElementsByTagName("InteractionLifeLine");
System.out.println("over all class tag:"+nodeList1.getLength());
        if (nodeList1 != null && nodeList1.getLength() > 0)
              {
                 for (int k = 0; k < nodeList1.getLength(); k++)
              {
    Element ell = (org.w3c.dom.Element) nodeList1.item(k);
    System.out.println("Class-Name="+ell.getAttribute("Name")+" "+"Class-Id="+ell.getAttribute("Id"));
    System.out.println("\n");
```

Figure 5.5: DOM parser to parse the XML file

Next is to define the Slicing criteria on the basis of which UML diagram has to be sliced diagrams. Here in the proposed approach variables and constants used in guard condition are taken as slicing criteria. As shown in Figure 5.6 output file as generated by parser and slicing criteria have to be passed to java slicer program; as represented by Figure 5.7.

Figure 5.8 shows the output file generated after applying slicing criteria related to variable “z”. Messages that uses z variable in their guard condition will be displayed in this file.

```
Object-Name=Object10 Object-Id=lblL4XyFYHySDwNl
Object-Name=Object9 Object-Id=rIFL4XyFYHySDwNe
Object-Name=Object8 Object-Id=rA5L4XyFYHySDwNX
Object-Name=Object7 Object-Id=ZSZL4XyFYHySDwNQ
Object-Name=Object6 Object-Id=SlpL4XyFYHySDwNJ
Object-Name=Object5 Object-Id=4lpL4XyFYHySDwNC
Object-Name=Object4 Object-Id=dOJL4XyFYHySDwM7
Object-Name=Object3 Object-Id=jHxL4XyFYHySDwM0
Object-Name=Object2 Object-Id=y0xL4XyFYHySDwMt
Object-Name=Object1 Object-Id=JdhL4XyFYHySDwMm
Message=[x>20]msg1 {TO-Object-Id= y0xL4XyFYHySDwMt & From-Object-Id= JdhL4XyFYHySDwMm}
Message=[y>50]msg2 {TO-Object-Id= jHxL4XyFYHySDwM0 & From-Object-Id= y0xL4XyFYHySDwMt}
Message=[z<50]msg3 {TO-Object-Id= dOJL4XyFYHySDwM7 & From-Object-Id= jHxL4XyFYHySDwM0}
Message=[x+y+z>0]msg4 {TO-Object-Id= y0xL4XyFYHySDwMt & From-Object-Id= dOJL4XyFYHySDwM7}
Message=[p=120]msg5 {TO-Object-Id= 4lpL4XyFYHySDwNC & From-Object-Id= y0xL4XyFYHySDwMt}
Message=[q=40]msg6 {TO-Object-Id= SlpL4XyFYHySDwNJ & From-Object-Id= 4lpL4XyFYHySDwNC}
Message=[r=20]msg7 {TO-Object-Id= 4lpL4XyFYHySDwNC & From-Object-Id= SlpL4XyFYHySDwNJ}
Message=[p*r>100]msg8 {TO-Object-Id= dOJL4XyFYHySDwM7 & From-Object-Id= 4lpL4XyFYHySDwNC}
Message=[z=90]msg9 {TO-Object-Id= jHxL4XyFYHySDwM0 & From-Object-Id= dOJL4XyFYHySDwM7}
```

Figure 5.6: Output-file generated by parser

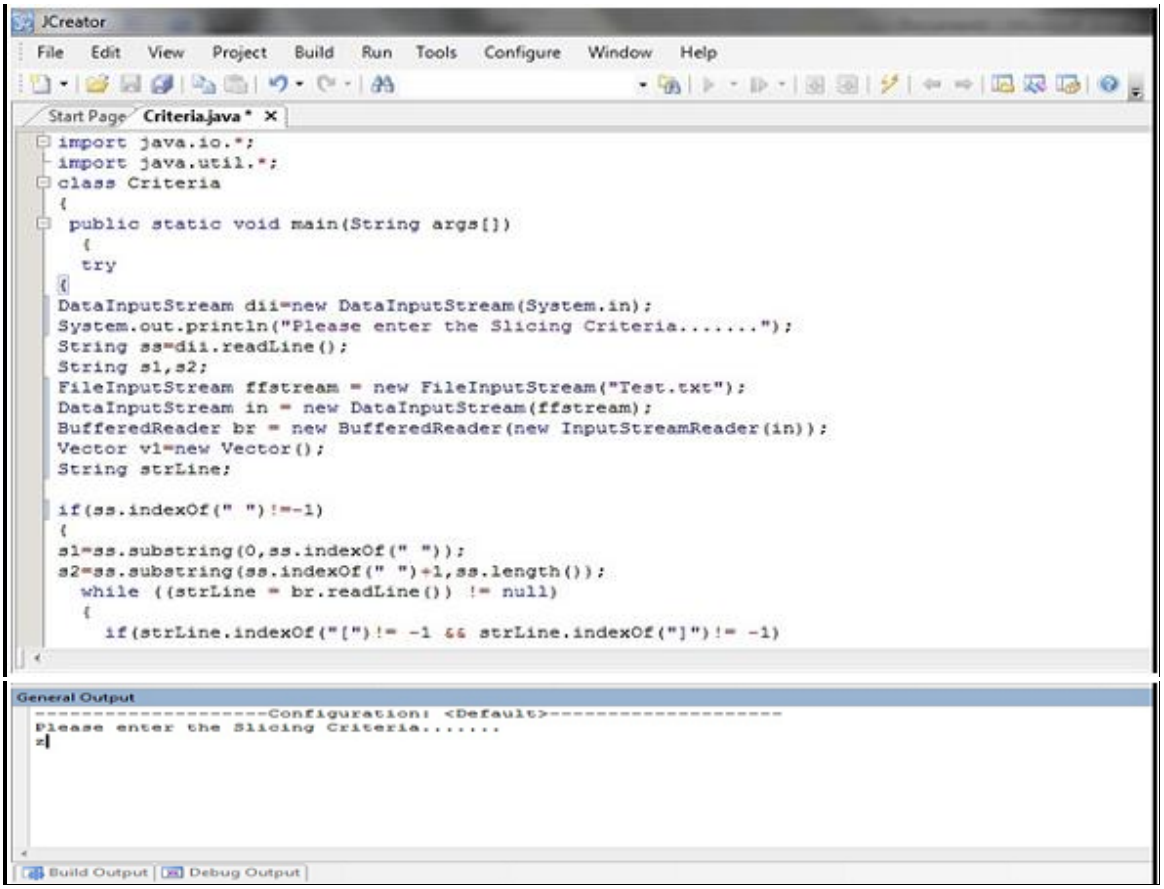

Figure 5.7: Java program for finding out the specified chunk

```
Message=[z<50]msg3 {TO-Object-Id= dOJL4XyFYHySDwM7 & From-Object-Id= jHxL4XyFYHySDwM0}
Message=[x+y+z>0]msg4 {TO-Object-Id= y0xL4XyFYHySDwMt & From-Object-Id= dOJL4XyFYHySDwM7}
Message=[z=90]msg9 {TO-Object-Id= jHxL4XyFYHySDwM0 & From-Object-Id= dOJL4XyFYHySDwM7}
Message=[z+p>200]msg10 {TO-Object-Id= 4IpL4XyFYHySDwNC & From-Object-Id= jHxL4XyFYHySDwM0}
Message=[x+z>100]msg16 {TO-Object-Id= ZSZL4XyFYHySDwNQ & From-Object-Id= y0xL4XyFYHySDwMt}
Message=[x+z<100]msg17 {TO-Object-Id= rA5L4XyFYHySDwNX & From-Object-Id= y0xL4XyFYHySDwMt}
Message=[x=40 and z=40]msg20 {TO-Object-Id= rIFL4XyFYHySDwNe & From-Object-Id= rA5L4XyFYHySDwNX}
Message=[y=50 and z>=40]msg21 {TO-Object-Id= IblL4XyFYHySDwNl & From-Object-Id= rIFL4XyFYHySDwNe}
Message=[z>=0]msg22 {TO-Object-Id= rIFL4XyFYHySDwNe & From-Object-Id= IblL4XyFYHySDwNl}
Message=[y*z>0]msg23 {TO-Object-Id= ZSZL4XyFYHySDwNQ & From-Object-Id= rIFL4XyFYHySDwNe}
Message=[z-p>0]msg24 {TO-Object-Id= 4IpL4XyFYHySDwNC & From-Object-Id= ZSZL4XyFYHySDwNQ}
Message=[z>100]msg25 {TO-Object-Id= SlpL4XyFYHySDwNJ & From-Object-Id= 4IpL4XyFYHySDwNC}
Message=[z-q>0]msg26 {TO-Object-Id= dOJL4XyFYHySDwM7 & From-Object-Id= SlpL4XyFYHySDwNJ}
```

Figure 5.8: output file generated after applying slicing

```
Message=[z<50]msg3 {TO-Object= Object4 & From-Object= Object3}
Message=[x+y+z>0]msg4 {TO-Object= Object2 & From-Object= Object4}
Message=[z=90]msg9 {TO-Object= Object3 & From-Object= Object4}
Message=[z+p>200]msg10 {TO-Object= Object5 & From-Object= Object3}
Message=[x+z>100]msg16 {TO-Object= Object7 & From-Object= Object2}
Message=[x+z<100]msg17 {TO-Object= Object8 & From-Object= Object2}
Message=[x=40 and z=40]msg20 {TO-Object= Object9 & From-Object= Object8}
Message=[y=50 and z>=40]msg21 {TO-Object= Object10 & From-Object= Object9}
Message=[z>=0]msg22 {TO-Object= Object9 & From-Object= Object10}
Message=[y*z>0]msg23 {TO-Object= Object7 & From-Object= Object9}
Message=[z-p>0]msg24 {TO-Object= Object5 & From-Object= Object7}
Message=[z>100]msg25 {TO-Object= Object6 & From-Object= Object5}
Message=[z-q>0]msg26 {TO-Object= Object4 & From-Object= Object6}
```

Figure 5.9: computed slice after the conversion of object-id to object-name

Figure 5.9 represents the file, when objects are present in Figure 5.8 will be replaced with corresponding object name. Here in this example according to the slicing criteria all the messages that use ‘z’ variable in their guard condition are present with the information of those objects among which message is floating.

```
import java.io.*;
import java.util.*;
class FileRead
{
 public static void main(String args[])
  {
  try
{FileInputStream ffstream = new FileInputStream("rupi.txt");
 DataInputStream in = new DataInputStream(ffstream);
 BufferedReader br = new BufferedReader(new InputStreamReader(in));
  String strLine;
  Vector v=new Vector();
  Vector v2=new Vector();
  while ((strLine = br.readLine()) != null)
  { String s ;
if (strLine.indexOf("TO-Object") != -1)
 { int i=strLine.indexOf("TO-Object")+10;
    s=strLine.substring(i,strLine.indexOf(" ",(i+2)));
     v.addElement(s);
     if(strLine.indexOf("From-Object") != -1)
      {
        //v2.addElement(s);
        int ii=strLine.indexOf("From-Object")+12;
        String ss=strLine.substring(ii,strLine.indexOf("}",(ii+2)));
        v2.addElement(ss);
        v2.addElement(s);
        ii=strLine.indexOf("Message=")+8;
        ss=strLine.substring(ii,strLine.indexOf("{",(ii+2)));
        v2.addElement(ss);
        }
  }
if(strLine.indexOf("From-Object") != -1 )
{
int i=strLine.indexOf("From-Object")+12;
    s=strLine.substring(i,strLine.indexOf("}",(i+2)));
      v.addElement(s);
}}
  Vector v1=new Vector();
```

Figure 5.10 Program for the conversion of information format

Java program as shown in Figure 10 has been used to convert the information into the input format of *Quick sequence diagram editor* [70] tool.

```
< Object4>:<int>[v]
< Object3>:<int>[v]
< Object2>:<int>[v]
< Object5>:<int>[v]
< Object7>:<int>[v]
< Object8>:<int>[v]
< Object9>:<int>[v]
< Object10>:<int>[v]
< Object6>:<int>[v]


< Object3>:< Object4>.<[z<50]msg3 >
< Object4>:< Object2>.<[x+y+z>0]msg4 >
< Object4>:< Object3>.<[z=90]msg9 >
< Object3>:< Object5>.<[z+p>200]msg10 >
< Object2>:< Object7>.<[x+z>100]msg16 >
< Object2>:< Object8>.<[x+z<100]msg17 >
< Object8>:< Object9>.<[x=40 and z=40]msg20 >
< Object9>:< Object10>.<[y=50 and z>=40]msg21 >
< Object10>:< Object9>.<[z>=0]msg22 >
< Object9>:< Object7>.<[y*z>0]msg23 >
< Object7>:< Object5>.<[z-p>0]msg24 >
< Object5>:< Object6>.<[z>100]msg25 >
< Object6>:< Object4>.<[z-q>0]msg26 >
```

Figure 5.11: Input file for quick sequence diagram editor

Figure 5.11 represent the input file specifically for the tool named *Quick Sequence diagram editor* [71]. File represented by Figure 5.9 has to be converted into the format as specified in Figure 5.11.

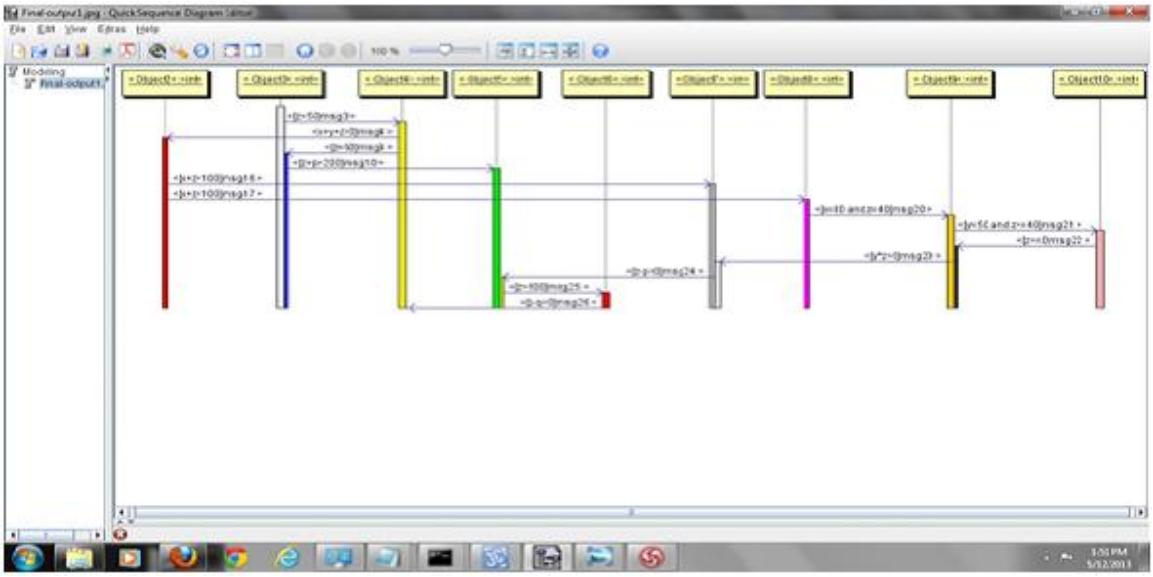

Figure 5.12: Quick Sequence Diagram Editor

Figure 5.12 and 5.13 shows the final & required sliced sequence diagram as obtained from the information available in Figure 5.11.

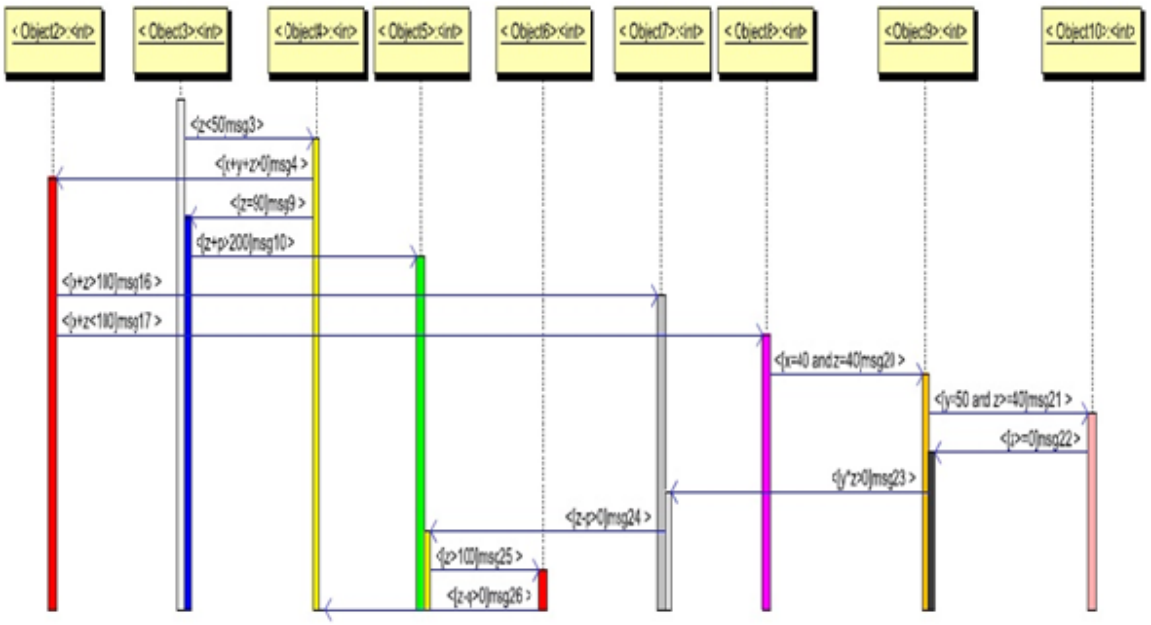

Figure 5.13: Computed Sliced Sequence diagram

Chapter 6

Conclusion and Future Scope

A new technique for model based slicing has been proposed that will extract the submodel from architecture of software to ease the software visualization.

6.1 Conclusion

The major contribution of this work is given below:

- The key contribution of the technique is to generate the refined model slices related to slicing criteria using conditional predicate in sequence diagram.
- The foundation of the proposed technique is 'UML' and 'Slicing'. With this, the problem of visualization of large and complex software can be handled efficiently.

6.2 Future Work

The proposed technique has focused on the generation of chunk using model based slicing but still there are the following points that can be explored further.

- The proposed technique can be extended further to handle the generation of test case and test set through model slices according to slicing criteria as per requirement.
- Technique can be explored further for concurrent and distributed programming.

References

[1] A. P. Mathur, "Foundation of Software Testing", Pearson/Addison Wesley, 2008.

[2] IEEE Standard 829-1998, "IEEE Standard for Software Test Documentation", pp.1-52, IEEE Computer Society, 1998.

[3] IEEE Standard 1059-1993, "IEEE Guide for Software Verification and Validation Plans", pp.1-87, Computer Society, 1993.

[4] R. S. Pressman, "Software Engineering – A Practitioner's Approach", McGraw Hill Education Asia, 2005.

[5] R. D. Craig, S. P. Jaskiel, "Systematic Software Testing", Artech House Publishers, Boston-London, 2002.

[6] ANSI/IEEE Standard 1008-1987, "IEEE Standard for Software Unit Testing", pp.1-23, IEEE Computer Society, 1997.

[7] S. R. Rakitin, "Software Verification and Validation for Practitioners and Managers", Artech House Publishers, Boston-London, 2001.

[8] I. Somerville, "Software Engineering", Sixth Edition, Addison-Wesley Publishers, 2001.

[9] Richard Torkar, "Towards Automated Software Testing Techniques, Classifications and Frameworks", Doctoral Dissertation, Department of Systems and Software Engineering, Blekinge Institute of Technology, pp.1-235, Series No 2006:04, Sweden, 2006.

[10] Quality assurance and testing [Online]. Available at: http://www.hanusoftware.com/services/quality-assurance-testing [As accessed on September 2012].

[11] F. Tip, "A Survey of Program Slicing Techniques", Journal of Programming Languages, vol. 3, No.3, pp. 121-189, 1995.

[12] G.B. Mund, R. Mall, and S. Sarkar, "An Efficient Dynamic Program Slicing Technique," Information and Software Technology, vol. 44, no. 2, pp. 123-132, 2002.

[13] M. Weiser, "Program Slices: Formal Psychological, and Practical Investigations of an Automatic Program Abstraction Method", PhD thesis, Computer and Communication Sciences Department, Univ. of Michigan, 1979.

[14] M. Weiser, "Program Slicing" Proceeding of 5th IEEE International Conference on Software Engineering, pp. 439-449, 1981.

[15] M. Weiser, "Program Slicing," IEEE Trans. Software Eng., vol. 10, No. 4, pp. 352-357, July 1984.

[16] B. Korel and J. Laski, "Dynamic Program Slicing," Information Processing Letters, vol. 29, No. 3, pp. 155-163, Oct. 1988.

[17] J.T. Lallchandani and R. Mall, "Slicing UML architectural models," ACM SIGSOFT Software Engineering Notes, vol.33, No.3, pp. 1–9, 2008.

[18] J.T. Lallchandani and R. Mall, "Integrated state-based dynamic slicing technique for UML models", Software, IET, vol. 4, No. 1, pp. 55–78, 2010.

[19] V. Ojala, "A slicer for UML state machines" Helsinki University of Technology, 2007.

[20] J. Qian and B. Xu, "Program slicing under UML scenario models", ACM SIGPLAN NOTICES, vol. 43, No. 2, 2008.

[21] P. Samuel and R. Mall, "A Novel Test Case Design Technique Using Dynamic Slicing of UML Sequence Diagrams", e-Informatica Software Engineering Journal Selected full texts, vol. 2, No. 1, pp. 61–77, 2008.

[22] K. Androutsopoulos, D. Clark, M. Harman, Z. Li, and L. Tratt, "Control dependence for extended finite state machines", Fundamental Approaches to Software Engineering, pp. 216–230, 2009.

[23] J.H. Bae, K.M. Lee, and H.S. Chae., "Modularization of the UML metamodel using model slicing", IEEE 5th International Conference on Information Technology: New Generation., pp. 1253–1254, 2008.

[24] H. Kagdi, J.I. Maletic, and A. Sutton, "Context-Free Slicing of UML Class Models", In Proceeding of 21st IEEE International Conference on Software Maintenance, pp. 635-638, 2005.

[25] A. Shaikh, R. Clarisó, U.K. Wiil, and N. Memon. Verification-driven slicing of UML/OCL models. In Proceedings of the IEEE/ACM International Conference on Automated software engineering, pages 185–194, ACM, 2010.

[26] Kevin Lano Crest, "Slicing of UML State Machines", In Proceedings of the 9th

WSEAS International Conference on Applied Informatics and Communications, pp.63-69, 2009.

[27] P. Samuel, R. Mall, and S. Sahoo, “UML Sequence Diagram Based Testing Using Slicing”, IEEE Indicon 2005 Conference, pp. 176–178, 2005.

[28] S. Van Langenhove, “Towards the Correctness of Software Behavior in UML: A Model Checking Approach Based on Slicing”, Dissertation, Department of Mathematics, Ghent University, 2006.

[29] Grady Booch, Ivar Jacobson & James Rumbaugh, “OMG Unified Modeling Language Specification”, Publisher: Addison Wesley, Version 1.3, First Edition: October 20, 1998.

[30] Object Management Group, "OMG Unified Modeling Language (OMG UML), Superstructure”, Version 2.4.1, August 2011, [Retrieved on 28-01-2013].

[31] Object Management Group, “UML Superstructure Specification,” Version 2.2, February 2009, [Retrieved on 28-01-2013].

[32] Scott W. Ambler, “UML 2 Class Diagrams”, Webdoc 2003-2009, Available at http://www. agilemodeling.com/artifacts, [As accessed on May 5, 2013].

[33] "Extensible Markup Language (XML) 1.0 (Fifth Edition)". W3.org. 2008-11-26. Retrieved 2013-05-05.

[34] James Duncan Davidson et al. Sun Microsystems. “Java API for XML Parsing”, version 1.0 final release, 2000.

[35] Grady Booch, James Rumbaugh, Ivar Jacobson, “The Unified Modeling Language User Guide", 2nd Edition, May 2005, Publisher. Addison Wesley.

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

[37] Jianjun Zhao, “Applying slicing technique to software architectures,” In Fourth IEEE International Conference on Engineering of Complex Computer Systems, pp.87 –98, 1998.

[38] J. Wang, Wei Dong, and Zhichang Qi, “Slicing Hierarchical Automata for Model Checking UML Statechart,” In Proceeding of 4th International Conference of Formal Engineering Methods and Software Engineering, pp. 435-446, Oct. 2002.

[39] W. Fangjun and Y. Tong, “Dependence Analysis for UML Class Diagrams”, Journal of Electronics (China), vol. 21, No. 3, pp. 249-254, May 2004.

[40] S. Van Langehove, “Internal Broadcasting to Slice UML State Charts”, In Proceeding of Theory and Practice of Software Verification, Oct.2005.

[41] Jung Ho Bae and Heung Seok Chae, “UMLSlicer: A tool for modularizing the UML metamodel using slicing”, In 8th IEEE International Conference on Computer and Information Technology, pp.772–777, 2008.

[42] Sagar Sen, Naouel Moha, Benoit Baudry, and Jean Marc Jézéquel, "Meta-model Pruning", In 12th International Conference on Model Driven Engineering Languages and Systems, pp.32-46, 2009.

[43] Jaiprakash T. Lallchandani, R. Mall, "Static Slicing of UML Architectural Models", Journal of Object Technology, vol. 8, No. 1, pp.159-188, January-February 2009.

[44] J. Lallchandani and R. Mall, “A Dynamic Slicing Technique for UML Architectural Models”, IEEE Transaction on Software Engineering, vol. 37, No. 6, NOV/DEC 2011.

[45] Philip Samuel, Rajib Mall, “Slicing-Based Test Case Generation from UML Activity Diagrams,” ACM SIGSOFT Software Engineering Notes, vol. 34, No. 6, November 2009.

[46] Kunihiro Noda, Takashi Kobayashi, Kiyoshi Agusa, Shinichiro Yamamoto, “Sequence Diagram Slicing”, In Proceeding of 16th Asia-Pacific Software Engineering Conference, IEEE, pp.291-298, 2009.

[47] Ranjita Kumari Swain , Vikas Panthi, Prafulla Kumar Behera, “Test Case Design Using Slicing of UML Interaction Diagram”, In Proceeding 2nd International Conference on communication, computing and security, Elsevier, pp.136-144, 2012.

[48] Hyeon-Jeong Kim , Doo-Hwan Bae, Vidroha Debroy, W. Eric Wong, “Deriving Data Dependence from UML State Machine Diagrams,” In Proceeding 5th IEEE International Conference on Secure Software Integration and Reliability Improvement, pp.118-126, 2011.

[49] Nisansala Yatapanage, Kirsten Winter and Saad Zafar, “Slicing behavior tree models for verification”, In IFIP Advances in Information and Communication Technology, pp.125–139, 2010.

[50] B. Korel, I. Singh, L. Tahat, and B. Vaysburg, “Slicing of State Based Models”, In Proceeding of International Conference of Software Maintenance, pp.34-43, 2003.

[51] Philip Samuel , Rajib Mall, Pratyush Kanth, “Automatic test case generation from UML communication diagrams”, Information and Software Technology (ELSEVIER), vol.44, No. 2, pp.158-171, 2007.

[52] J. Julliand, N. Stouls, P-C. Bue, P-A. Masson, “B model slicing and predicate abstraction to generate tests,” Software Quality Journal, vol. 21, pp.127-158, 2013.

[53] Asadullah Shaikh, Uffe Kock Wiil, and Nasrullah Memon, "UOST: UML/OCL aggressive slicing technique for efficient verification of models", In 6th International Workshop on System Analysis and Modeling, pp. 173–192, 2010.

[54] Asadullah Shaikh, Uffe Kock Wiil, and Nasrullah Memon, "Evaluation of tools and slicing techniques for efficient verification of UML/OCL class diagrams", Advances in Software Engineering, vol.18, pp 173-192, 2011.

[55] Monalisa Sarma, Debasish Kundu, Rajib Mall, “Automatic Test Case Generation from UML Sequence Diagrams,” 15th IEEE International Conference on Advanced Computing and Communications, pp. 60-65, 2007.

[56] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert France, “Slicing feature models,” In Proceeding 26th IEEE/ACM International Conference on Automated Software Engineering, pp. 424-427, 2011.

[57] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert France, “Separation of Concerns in Feature Modeling: Support and Applications,” In Proceedings of the Aspect-Oriented Software Development (AOSD'12), pp.1-12, ACM, March 2012.

[58] Arnaud Hubaux, Patrick Heymans, Pierre-Yves Schobbens, Ebrahim Khalil Abbasi, and Dirk Deridder, “Supporting multiple perspectives in feature-based

configuration", Software and Systems Modeling, vol. 12, No. 3, pp.641-663, 2013.

[59] T. Kim, Y.-T. Song, L. Chung, and D.T. Huynh, "Dynamic Software Architecture Slicing", In Proceeding 23rd International conference on Computer Software and Applications , pp. 61-66, 1999.

[60] T. Kim, Y.-T. Song, L. Chung, and D.T. Huynh, "Software Architecture Analysis: A Dynamic Slicing Approach", Journal of Computer and Information Science, vol. 1, No. 2, pp. 91-103, 2000.

[61] Davide Falessi, Shiva Nejati, Mehrdad Sabetzadeh, Lionel Briand, and Antonio Messina, "SafeSlice: A model slicing and design safety inspection tool for SysML", In Proceeding 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering and 13rd European Software Engineering Conference, ACM, 2011.

[62] Kevin Lano and Shekoufeh K. Rahimi, "Slicing of UML Models Using Model Transformations," Model Driven Engineering Languages and Systems, 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010**,** Lecture Notes of Computer Science, Vol. 6395, pp. 228-242, Springer, 2010.

[63] Zoltán Ujhelyi, Ákos Horváth, and Dániel Varró, "Towards dynamic backward slicing of model transformations," In Proceeding 26th IEEE/ACM International Conference on Automated Software Engineering, pp.404–407, IEEE, 2011.

[64] Zoltán Ujhelyi, Ákos Horváth, and Dániel Varró, "Dynamic Backward Slicing of Model Transformations", In Proceeding IEEE 5th International Conference on Software Testing, Verification and Validation, pp. 1-10, 2012.

[65] A. Blouin, B. Combemale, B. Baudry, O. Beaudoux, "Modeling model slicers," In Proceedings of the 14th IEEE/ACM International conference on Model driven engineering languages and systems, pp.62-76, 2011.

[66] A. Blouin, B. Combemale, B. Baudry, O. Beaudoux, "Kompren Modeling and Generating Model Slicers," Journal of Software and System Modeling, Springer, 2012.

[67] M. Khandai, A.A. Acharya, D.P. Mohapatra, "A Novel Approach of Test Case Generation for Concurrent Systems Using UML Sequence Diagram", In

Proceeding of 3rd International Conference on Electronics Computer Technology (ICECT), pp. 157 – 161, 8-10 April 2011.

[68] Mahesh Shirole, Rajeev Kumar, "Testing for Concurrency in UML Diagrams", ACM SIGSOFT Software Engineering Notes, vol. 37, No. 5, pp.1-8, 2012.

[69] Vinay Arora, Rajesh Kumar Bhatia and Maninder Singh, "Evaluation of Flow Graph and Dependence Graphs for Program Representation", International Journal of Computer Applications, vol. 56, Issue 14, pp. 18-23, October 2012.

[70] Tool Visual Paradigm for UML, Available at: http://www.visual-paradigm.com/download/vpuml.jsp [As Accessed on 25th January 2013].

[71] M. Strauch, "Quick Sequence Diagram Editor," Available at: http://sdedit.sourceforge.net/ [As Accessed on 25th February 2013].

List of Publication

Published:

[1] Rupinder Singh, Vinay Arora, "Literature Analysis on Model based Slicing", International Journal of Computer Applications, Vol. 70, Issue.16, pp. 45-51, May 2013.

[2] Rupinder Singh, Vinay Arora, "Technique for Extracting Subpart from UML Sequence Diagram", International Journal of Advanced Research in Computer Science and Software Engineering, Vol. 3, Issue 6, pp. 593-596, June 2013.

[3] Rupinder Singh, Vinay Arora, "A practical approach for model based slicing", IOSR Journal of Computer Engineering , Vol. 12, Issue. 4, pp. 18-26, July 2013.