AN INTEGRATED UML BASED MODEL FOR DESIGN ANALYSIS

By

ADAM MCDONALD

A thesis submitted in partial fulfillment of
the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

WASHINGTON STATE UNIVERSITY
School of Engineering and Computer Science

May 2010

To the Faculty of Washington State University:

The members of the Committee appointed to examine the thesis of ADAM MCDONALD find it satisfactory and recommend that it be accepted.

Orest Jacob Pilskalns, Ph.D., Chair

Charles Lang, Ph.D.

Wayne Cochran, Ph.D.

ACKNOWLEDGMENTS

This work would not have been possible without the support and encouragement of the Washington State University computer science faculty and my advisor Dr. Orest Pilskalns.

AN INTEGRATED UML BASED MODEL FOR DESIGN ANALYSIS

Abstract

by Adam McDonald, M.S.
Washington State University
May 2010

Chair: Orest Pilskalns

In software engineering, there is a strong movement towards "Design First" and "Test Driven Development". With these approaches it is imperative to ensure that design documents are valid and consistent both internally and externally. This thesis discusses the current state of design testing documents for validity and proposes a new approach. Although much research effort has been dedicated to software design validation, none of the current solutions provide an effective, efficient, and automatic approach that includes a wide variety of UML design document types. To remedy this, we present an new approach which attempts to address the downfalls of the other solutions. To demonstrate this approach we apply our techniques to a case study. The case study is based around designing a canonical web application for blogging. By first designing the project in a variety of UML design documents and then running those documents through our proposed approach, we were able to pinpoint numerous design faults and inconsistencies between the diagrams. Using our approach, software faults are discovered early in the development lifecycle and therefore reduce software maintenance time and costs overall.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software developers are beginning to shift their focus from implementation testing to design analysis and design testing. The hope is that finding errors earlier in the software life cycle will result in overall savings. Quality requirements, sometime referred to as non-functional requirements are often overlooked in the design phase, because they can only be evaluated by merging aspects of the design and simulating execution of the design. Software testing research has provided a wide array of test and test generation methods, particularly for code but not for designs. More recently, testing methods have been proposed for designs. Given that the *Unified Modeling Language* (UML) [17] is the de-facto standard for design artifacts, most of the testing approaches involve designs using various UML diagrams. UML has made it possible to describe designs with a uniform notation at a variety of design levels from conceptual to detailed design [3].

There are two basic ways to use UML design artifacts for testing; either for testing an implementation against its design (e.g. [4]), or to test the designs themselves to evaluate their quality (e.g. [14]). Complex systems, such as telecommunication systems, can lead to hundreds of pages of UML designs in various notations. These diagrams commonly interact in ways whose validity and correctness are far from easily determined. Given the complexity and multiple views through multiple models like Class Diagrams, Sequence Diagrams, constraints in *Object Constraint Language* (OCL) [17] and the like, designs can be difficult to evaluate.

This thesis aims to provide a solution to the following problem:

- Effectively, efficiently and automatically testing the design of a software represented by UML diagrams in order to find software faults as early as possible during the software development lifecycle.

1.1 Dissertation outline

This thesis addresses the problem of design testing across various UML diagrams including OCL. Chapter 2 is dedicated to proposing a solution to the the testing problem and is organized as follows:

- The first section provides a survey of the current state of work related to testing software with respect to its design.
- The second section explains our approach in detail while demonstrating it on generic UML diagrams.

Chapter 3 illustrates our approach on a case study and summarizes the results.

Finally, chapter 4 concludes and gives directions for future work.

1.2 Abbreviations

ASSL	A Snapshot Sequence Language
CA	Conditional Attribute
CCT	Class Constraint Tuple
CE	Conditional Edges
DG	Directed Graph
DIG	Design Interaction Graph
DUT	Design Under Test
EA	Event Attribute
EDUT	Executable DUT
ID	Implementation Dependency
JAL	Java-like Action Language
MVC	Model View Controller
OCL	Object Constraint Language
OMDAG	Object Method Directed Acyclic Graph
PL	Path Length
SCT	System Configuration Table
SD	Sequence Diagram
SST	System Structure Table
TAM	Testable Aggregate Model
TDUT	Testable DUT
UML	Unified Modeling Language
XML	eXtended Markup Language

Chapter 2

Behavior Interaction Testing

2.1 Related Works

This section surveys the current state with respect to testing and validation of UML designs themselves. As of now, a comprehensive set of testing criteria, approaches, and tools does not exist. This section gives a brief introduction to what is currently available. Table 2.1 shows existing testing approaches, denoting their similarities and differences.

2.1.1 State Validation via Snapshot Creation

Gogolla et al. [9] propose an approach to validate system states. These states, called snapshots, are represented by object diagrams which consist of objects, attribute values for each object and links among objects. Snapshot validation is done using a tool called **USE** which the authors developed earlier. The goal here is to facilitate the snapshot generation by defining the properties that they need to satisfy. For this

Table 2.1: Related Works Comparison

	Our Approach	Gogolla 2003 [9]	Trong 2003 [19]	Pilskalns 2003 / 2007 [14][15]
Automated	No, future work	Yes, USE tool	Yes, EPTUD plugin	Yes
Supported Diagrams	Use Case, Class, State, Sequence, Activity, Interaction	Class	Class, Activity	Class, Sequence
Fault Detection	Static: generalization/association/navigation violations, invalid calling/called states, missing behavior; Dynamic: OCL violations, path faults, multiplicity violations	OCL violations based on a given 'state snapshot'	OCL violations by comparing observed and actual behavior	OCL violations, path faults, multiplicity violations

purpose, Gogolla et. al developed *A Snapshot Sequence Language* (ASSL) that allows the generation of desired snapshots by specifying their properties.

ASSL defines the sequence of operations needed to generate a snapshot with **USE**. In other words, the properties of snapshots are integrated into ASSL procedures. Test cases examine the system with respect to desired properties that need to be satisfied by class diagrams. Each property in the test case is specified using *dynamic* invariants as opposed to *present* invariants that need to be satisfied globally. Dynamic and present invariants are defined using OCL expressions. When a dynamic invariant is loaded into **USE**, it will make sure that there exists a snapshot that satisfies both the dynamic invariant and the present invariant. This is done by showing that there exists no valid snapshot that satisfies the present invariant as well as the negation of the dynamic invariant. Figure 2.1 gives a general view of the USE approach.

2.1.2 Test Execution via JAL/EPTUD

Trong et al. [19] introduced a testing approach in which executable forms of UML design models are exercised with test inputs generated from the class diagrams and activity diagrams. Later, the expected behavior defined with OCL constraints and the observed behavior are compared and failures are reported. Their approach is supported by a prototype tool, *Eclipse Plugin Test Under Test* (EPTUD). Class diagrams are used to characterize a set of valid object configurations while activity diagrams help define class operations. A *Java-like Action Language (JAL)*[12] is employed to describe the semantics of actions. The testing process begins by introducing the *Design Under Test (DUT)* to the testing system. *DUT* is transformed into Executable *DUT (EDUT)*. *EDUT* is a program that simulates the behavior modeled in the *DUT*. *EDUT* contains two parts: a static structure representing the runtime configuration of the *DUT*, and a simulation engine.

The static structure is derived from the class diagrams while the simulation engine

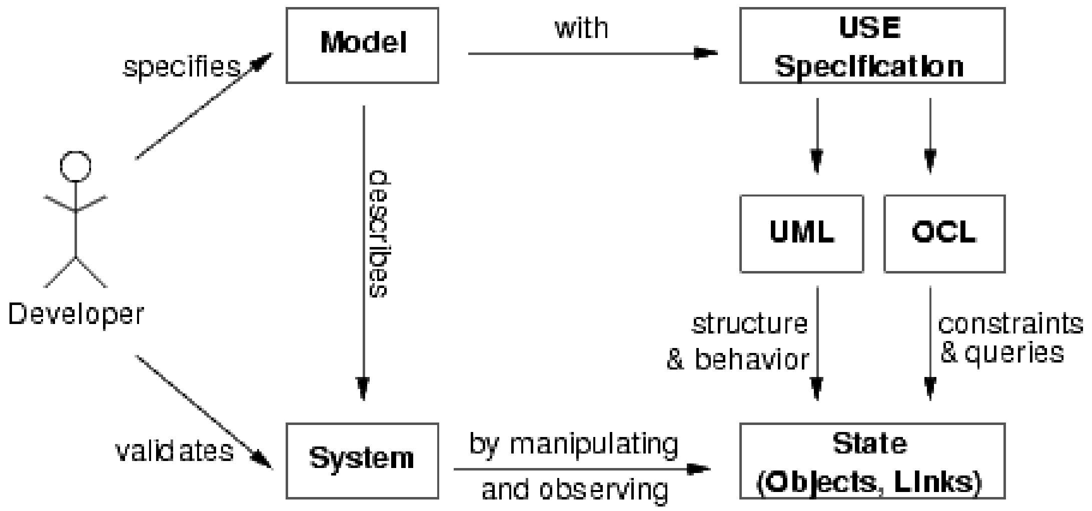


Figure 2.1: USE Overview.

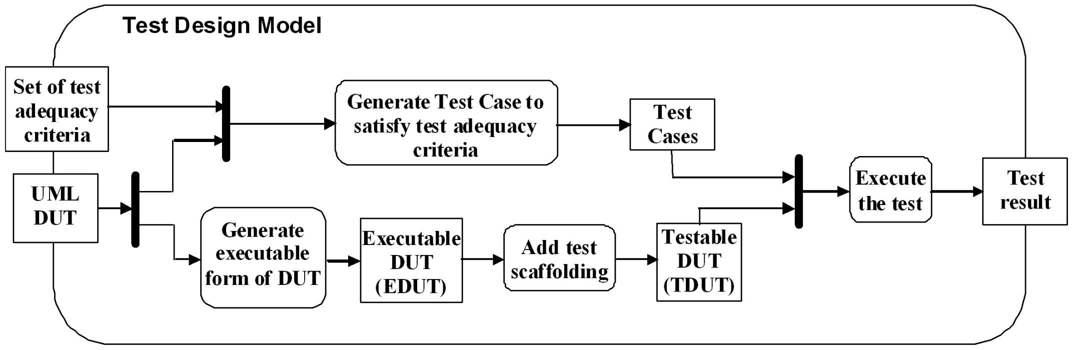


Figure 2.2: Overview of the testing process.

is generated from the activity diagrams. Test scaffolding is added to *EDUT* to perform failure checks *(TDUT)*. Test cases are implemented on the *TDUT* and results are reported by an observer class. Figure 2.2 illustrates an overview of this approach.

2.1.3 Behavioral and Structural Testing via TAM

Most existing testing approaches for UML designs provide simple static analysis capabilities that can check model consistency. This can be done by validating structural views (e.g. class diagrams) against invariants represented by OCL expressions, or validating behavioral views (e.g. sequence diagrams) against pre/post conditions represented by OCL expressions. However, these approaches do not validate dependencies between views. In other words, they don't address the problem of revealing inconsistencies among behavioral and structural views.

Pilskalns et al. [14][15] address this problem by introducing a framework to test behavioral and structural aspects of UML designs by integrating the two views into a single representation called *Testable Aggregate Model* (TAM)[14] (its earlier version was called *Object Method Directed Acyclic Graph (OMDAG)*[15]). They provide a framework to generate and execute test cases using TAM and to validate test results by comparing them against OCL expressions.

The TAM is constructed by combining the behavioral information of sequence diagrams with the structural information of class diagrams. This aggregation of sequence and class diagrams makes this approach different from [9] as it allows validation of multiple types of diagrams at the same time, effectively testing for cross-diagram defects. The approach consists of the following steps:

1. Build TAM using UML models.
   (a) Construct a *Directed Graph* (DG) from each sequence diagram.
   (b) Construct *Class Constraint Tuples* (CCT) from class diagram and OCL expressions.
   (c) Combine DG and CCT into TAM.
2. Determine input model and generate test cases.
   (a) Determine which attributes need partitioning.
   (b) Partition attributes with domain analysis [2] to generate test cases.
3. Execute the tests.
   (a) For each test:
      i. Record potential faults.
      ii. Validate test results.

A DG is represented by the tuple $G = \langle V, E, s \rangle$ where V is a set of vertices, E is the set of edges, and s is the starting vertex. A vertex in DG can be a simple vertex representing a message or a Sub-DG representing a *combined fragment* hence representing several levels of abstraction. *Combined fragments* allow the developer to describe the control flow of messages with conditions. In the context of [14], three kinds of combined fragments are considered. These are *option* (i.e. 'if' statement), *alternative* (i.e. 'switch' statement), and *loop.* The loop fragment may contain a boolean guard condition, as well as a minimum and maximum number of iterations.

Test cases consist of values for variables or attributes that enable traversing a path in the TAM. Thus, variables and attributes that are present in conditional statements constitute the input model. The set of values that they can obtain defines the input domain. The following steps define the input model:

1. Identify the set of variables that occur in conditions.
2. Determine the range based on type of variable. Use one of the combinatorial techniques in [2] to determine partitions and combinations of partitions.
3. Select test values based on the combinatorial techniques used in step 2.

To validate the results, the changes to the system during test execution must be recorded. This is facilitated by use of an instance and trace table. The instance table keeps count of the number of instances for each class. The trace table records each message, each object, and every attribute assigned values. The instance table is updated as execution proceeds for the class type of the object as well as all of its super classes (if available). For each conditional vertex v_i attribute values are assigned and recorded based on the executed test case.

With the trace table and instance table, two types of faults can be revealed. The first type is an *OCL fault.* It occurs when states recorded in the execution trace or instance tables violate OCL constraints. The second type is classified as *path fault* where a path may not be traversable or may not exist. This can be caused by calling a private, abstract, or non-existing operation. In addition to detecting OCL and path faults, the instance table can be used to detect association end multiplicity violations.

2.1.4 Analysis of Current State

Based on the existing works and their deficiencies, it can be concluded that the existing works on testing software designs have limited applicability in terms of both effectiveness and flexibility. The limited scope of detectable faults would jeopardize the effectiveness while the minimal support for a variety of UML diagram types puts doubt on the flexibility of the solution. The main motivation behind our approach is to invent a simple yet complete approach that meets both effectiveness and flexibility standards of an optimal testing approach.

2.2 Approach

Our testing technique can be summarized in four subsequent phases:

1. Constructing a model referred to as the *Design Interaction Graph* (DIG) from the usecase, sequence, activity and state diagrams.
2. Deriving the *System Structure Table* (SST) from the class diagram.
3. Deriving the test cases from the design interaction graph (DIG).
4. Execute test cases by instantiating (DIG) to find design faults.

In this section we explain each phase starting from the construction of the DIG.

2.2.1 Constructing the Design Interaction Graph (DIG)

Similar to other graph presentations, we represent DIG as: $DIG = \langle V, E \rangle$ where V is the set of vertices and E is the set of edges. During the construction process a vertex can be either a usecase, a class, or a state within a class while an edge can represent either the control flow between different usecases, message calls or events with optional pre/post conditions and guards and actions that are executed after the message calls. Note that in special situations an edge might have no message or event on it. These special edges are shown by dashed lines with an ϵ symbol appearing on them and will be addressed later. A formal definition of edges and vertices will be given in a later section. We construct the *Design Interaction Graph* (DIG) using an incremental process. This process is shown in Figure 2.3 using an activity diagram. The construction process is composed of five actions. Each action uses a specific type of UML diagram and adds the information from that diagram to the design interaction graph model. In the subsequent sections we elaborate how these diagrams are added to the model and we demonstrate every action in Figure 2.3 for each UML diagram on a generic diagram of that type.

Convert Usecase Sequential Constraint Into a Directed Graph

In some situations, the execution of some usecases might be dependent upon the execution of others. For example, a remove course usecase can only take place if that course has already been added to the system. Therefore, in general there might be a relation between the execution of different usecases. Briand et al. [4] use a variation of the activity diagram to represent the execution dependencies of usecases with each other. The activity diagram enables the designer to describe different sequences of execution constraints in a single diagram hence calling it usecase sequential constraint

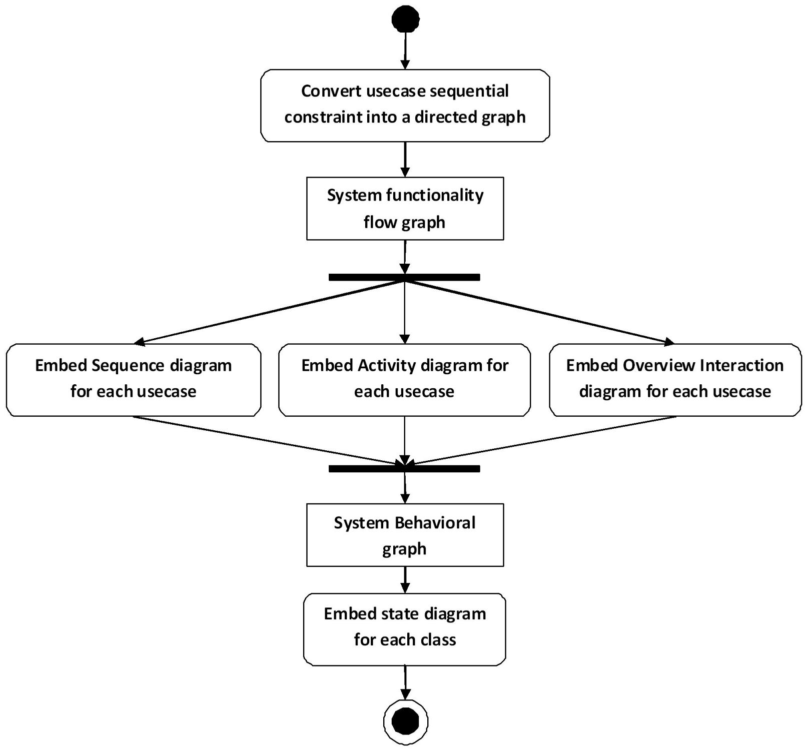

Convert usecase sequential constraint into a directed graph
System functionality flow graph
Embed Sequence diagram for each usecase
Embed Activity diagram for each usecase
Embed Overview Interaction diagram for each usecase
System Behavioral graph
Embed state diagram for each class

Figure 2.3: Overview of the *DIG* construction acitivity.

diagram.

The first stage of our construction process takes the usecase sequential constraint diagram and converts it into a directed graph. This is done by replacing all the activities and all the transitions in the usecase sequential constraint diagram with vertices and edges respectively. The start and end nodes will be translated into start and end vertices in the directed graph while the fork and join vertices will be ignored. Finally, the decision nodes will be translated into conditions on the transitions that are coming out of each decision node. Figure 2.4 illustrates the process of transforming a generic form of usecase sequential constraint diagram into a directed graph. This directed graph represents a high level functionality flow of the system where the start and ending vertices show the start and ending points of this flow. Therefore, we refer to this directed graph as *system functionality flow graph.* Note that in the system functionality flow graph, all the transitions are ϵ transitions. This means that these transitions are not conveying any messages or events and only represent a flow of functionality between different usecases.

Embedding Sequence Diagram for Each Usecase Scenario

The internal behavior of a usecase can either be modeled by a sequence diagram or an activity diagram. In case a sequence diagram is used to model a usecase scenario, the sequence diagram information should somehow be added to the system functionality flow graph. In this subsection we explain how the information in a sequence diagram for a particular usecase will be added to the system functionality flow graph.

A sequence diagram can be viewed as a directed graph with each lifeline/class representing a vertex and each message representing an edge. Each guard condition that is present on a message will be transformed into a condition on the corresponding edge. The process of transforming combined fragments, i.e. alt, opt, break, par, loop is illustrated in Figures 2.5, 2.6, 2.7, 2.8, and 2.9, while the generic sequence diagram in Figure 2.10 is transformed into the graph in Figure 2.11. The variable PL is the *Length of the Path* traversed in the graph. That is why it is being incremented on each edge traversal. The condition on PL enforces the correct sequence of messages in the graph traversal. Note that for a *loop* fragment we use two other variables in addition to PL. These variables represent the total number of edge traversals within the loop (k) and the number of edge traversals on each execution of the loop (i). These two variables are important in the sense that they enforce the correct sequence of traversal within the loop i.e. variable i, and after the execution of the loop, i.e. k.

During the process of converting the sequence diagram into a directed graph, we

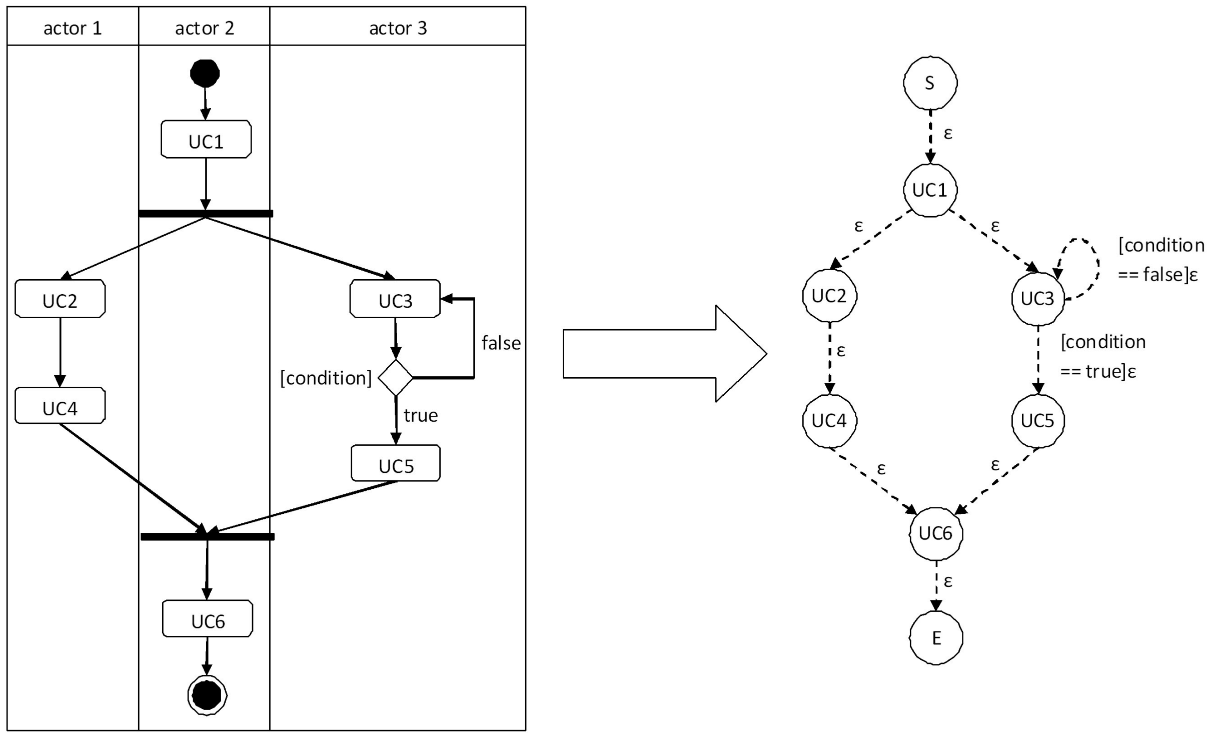


Figure 2.4: Converting a usecase sequential constraint diagram to a directed graph.

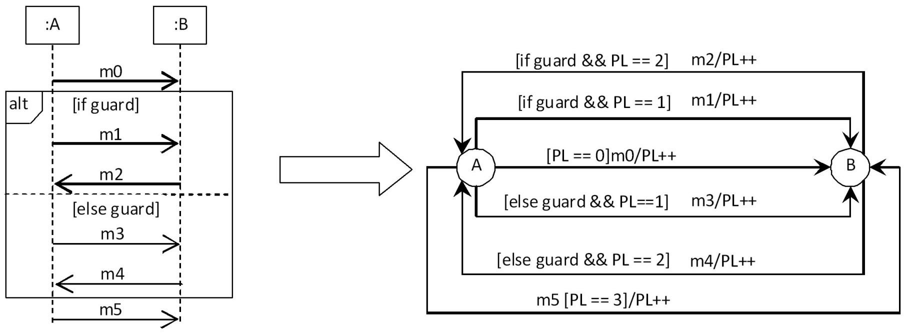

:A
:B
m0
alt
[if guard]
m1
m2
[else guard]
m3
m4
m5
[if guard && PL == 2] m2/PL++
[if guard && PL == 1] m1/PL++
[PL == 0]m0/PL++
A
B
[else guard && PL==1] m3/PL++
[else guard && PL == 2] m4/PL++
m5 [PL == 3]/PL++

Figure 2.5: Converting an *alt* fragment in sequence diagram into a directed graph.

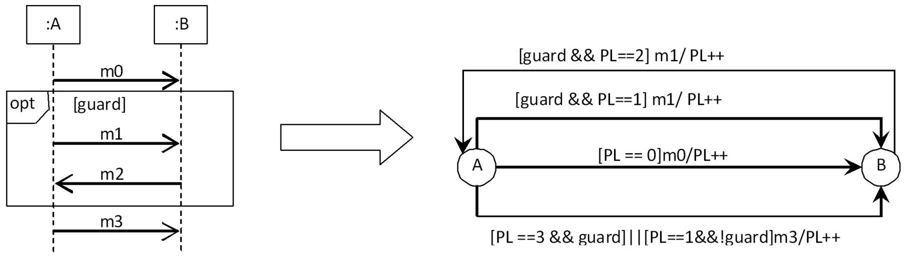

:A
:B
m0
opt
[guard]
m1
m2
m3
[guard && PL==2] m1/ PL++
[guard && PL==1] m1/ PL++
[PL == 0]m0/PL++
A
B
[PL ==3 && guard]||[PL==1&&!guard]m3/PL++

Figure 2.6: Converting an *opt* fragment in sequence diagram into a directed graph.

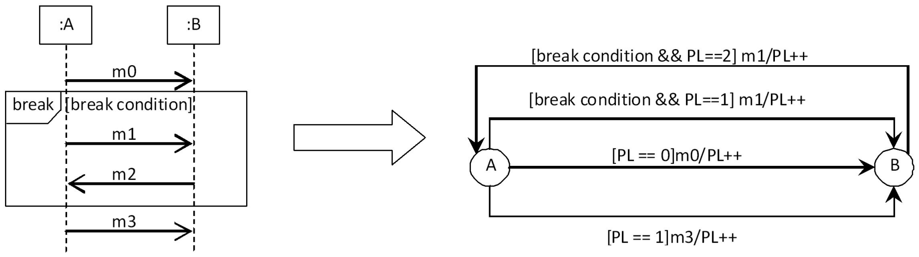

:A
:B
m0
break
[break condition]
m1
m2
m3
[break condition && PL==2] m1/PL++
[break condition && PL==1] m1/PL++
[PL == 0]m0/PL++
A
B
[PL == 1]m3/PL++

Figure 2.7: Converting a *break* fragment in sequence diagram into a directed graph.

identify the very first lifeline/class that invokes a message in the sequence diagram as the start lifeline/class of the sequence diagram. Respectively, the lifeline/class that receives the last message is identified as the ending lifeline/class. These two lifelines/classes are identified as start and end vertices in the graph of the sequence

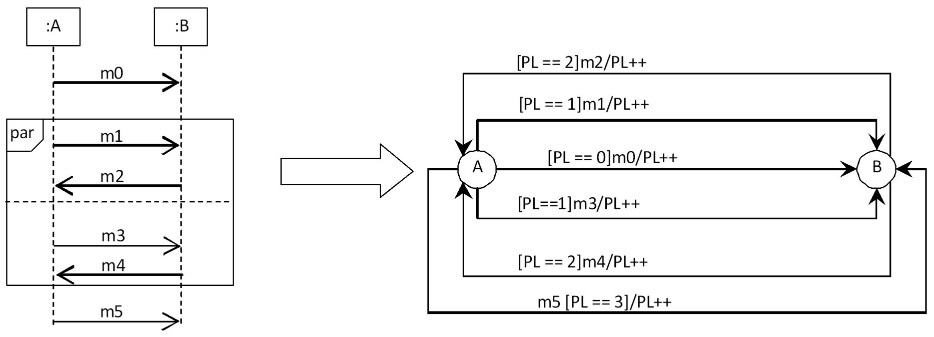


Figure 2.8: Converting a *par* fragment in sequence diagram into a directed graph.

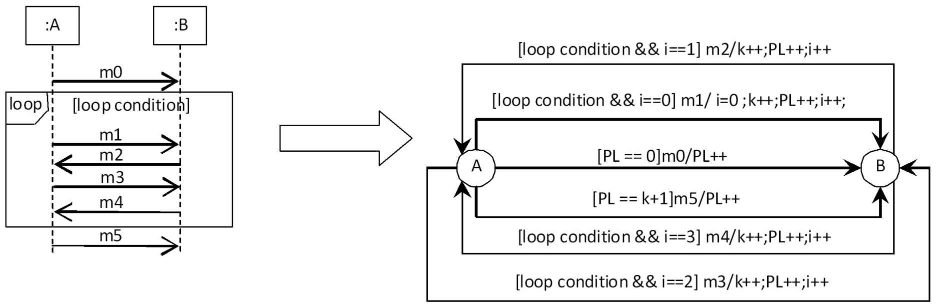

:A
:B
m0
loop
[loop condition]
m1
m2
m3
m4
m5
[loop condition && i==1] m2/k++;PL++;i++
[loop condition && i==0] m1/ i=0 ;k++;PL++;i++;
[PL == 0]m0/PL++
A
B
[PL == k+1]m5/PL++
[loop condition && i==3] m4/k++;PL++;i++
[loop condition && i==2] m3/k++;PL++;i++

Figure 2.9: Converting a *loop* fragment in sequence diagram into a directed graph.

diagram. In case the ending and starting lifelines/classes of two consecutive messages are not the same we connect the ending and the starting lifelines of these two messages with an ϵ edge in the graph as is demonstrated in Figure 2.11.

Now that the sequence diagram is converted into a graph, we need to embed this graph into the system functionality flow graph. This process is done by replacing the usecase vertex with the corresponding graph of its sequence diagram. Note that during this process, all the incoming and outgoing edges of the usecase vertex should point to the start and end vertices of the graph of the sequence diagram. Figure 2.12 illustrates how the the generic sequence diagram of Figure 2.10 is embedded into the system functionality flow graph in Figure 2.4 while replacing $u1$.

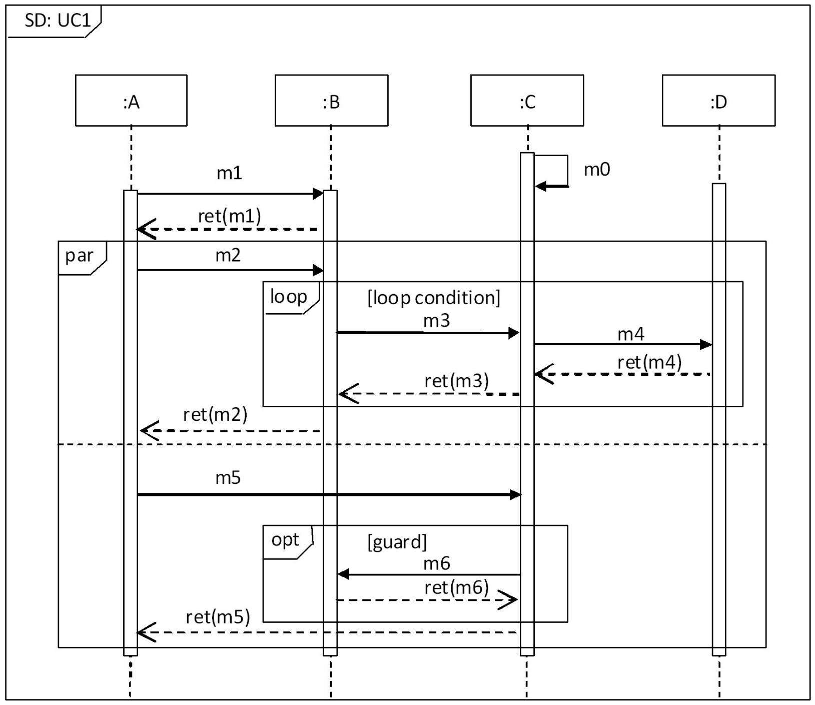


Figure 2.10: An example of a generic form of sequence diagram.

Embedding Activity Diagram for Each Usecase

Another way of modeling the internal behavior of a usecase is through the use of activity diagrams. An activity diagram can be used to show the actual flow of control between classes (in the form of swimlanes) in the usecase. In order to add the information in the activity diagram to the system functionality flow graph, we first need to convert the activity diagram into a directed graph and then embed this directed graph into the system functionality flow graph. Before converting the activity diagram into a directed graph, we need to flatten the activity diagram by replacing all the sub-activities with their corresponding activity diagrams.

We convert an activity diagram into a directed graph by representing each class in the swimlanes with a single vertex and each activity with an edge. Since an

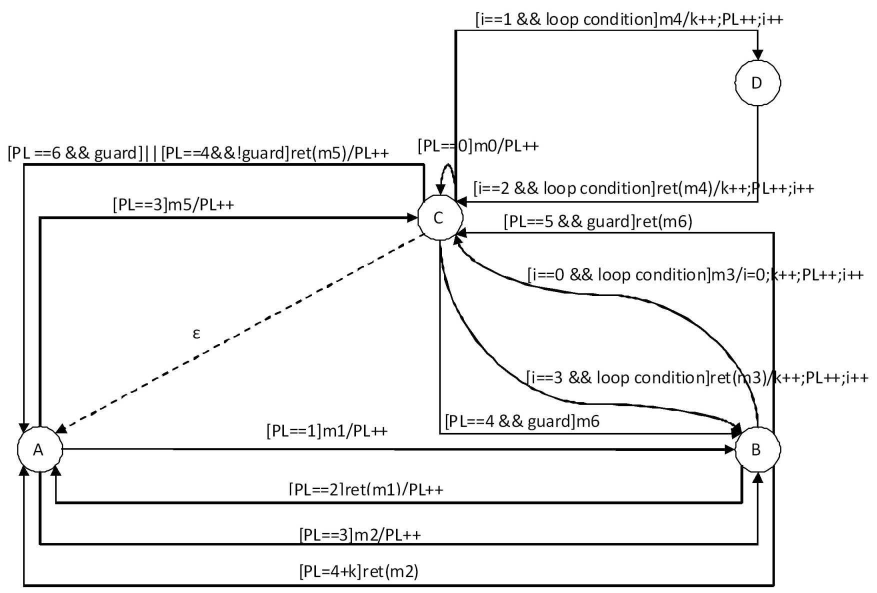

[i==1 && loop condition]m4/k++;PL++;i++
D
[PL ==6 && guard]||[PL==4&&!guard]ret(m5)/PL++
[PL==0]m0/PL++
[i==2 && loop condition]ret(m4)/k++;PL++;i++
[PL==3]m5/PL++
C
[PL==5 && guard]ret(m6)
[i==0 && loop condition]m3/i=0;k++;PL++;i++
ε
[i==3 && loop condition]ret(m3)/k++;PL++;i++
[PL==4 && guard]m6
[PL==1]m1/PL++
A
B
[PL==2]ret(m1)/PL++
[PL==3]m2/PL++
[PL=4+k]ret(m2)

Figure 2.11: Sequence diagram of Figure 2.10 converted into a directed graph.

activity diagram shows the flow of control between different classes, we use ϵ edges that connect the corresponding vertices to represent these flow of controls. Currently we only support the transformation of the activity nodes that are present in the generic activity diagram of Figure 2.13. The corresponding directed graph of the activity diagram in Figure 2.13 is shown in Figure 2.14. Assuming that the activity diagram AD models the internal behavior of usecase UC, the directed graph DG that corresponds to AD will be embedded into the system functionality flow graph by first replacing UC with DG and then removing the start vertex of DG while connecting all the outgoing edges of the start vertex to all vertices that had an outgoing edge to UC in the system functionality flow graph. Also, the end vertex of DG is removed and all the incoming edges of the end vertex will be connected to all the vertices in the system functionality flow graph that had an incoming edge from UC. Figure 2.15 illustrates how the directed graph of Figure 2.14 is embedded into the system

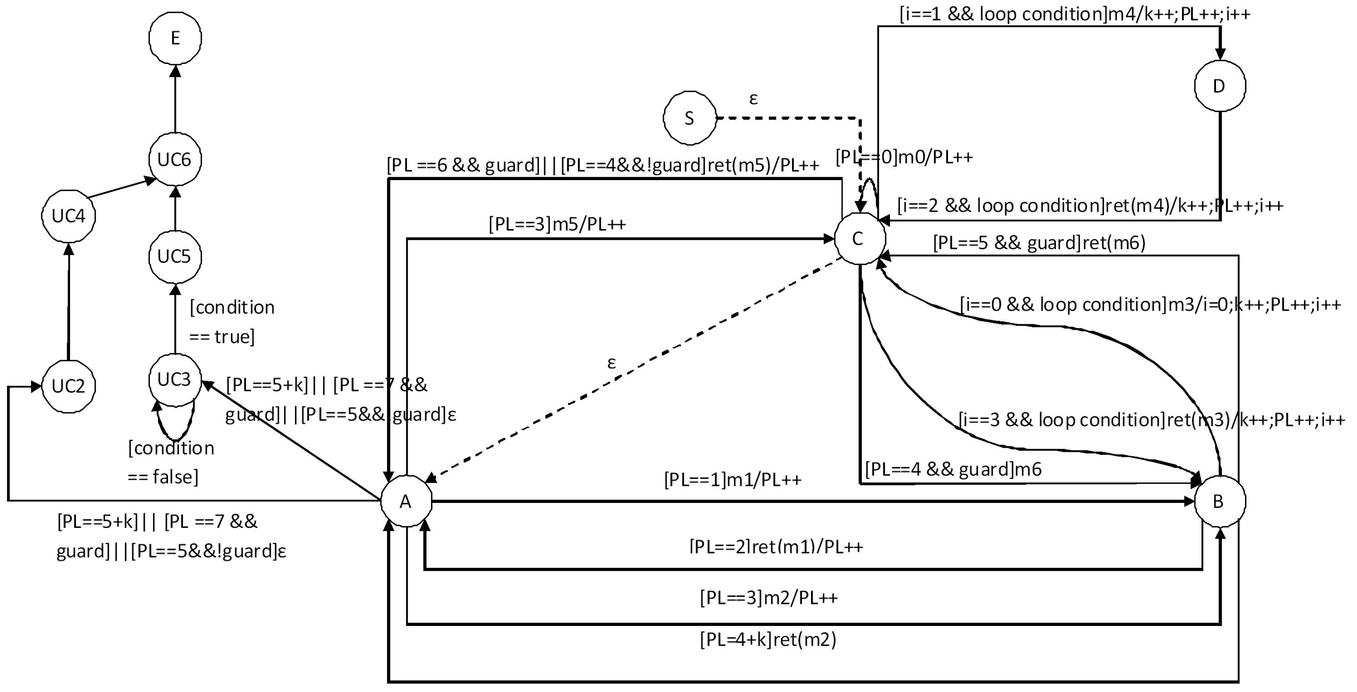


Figure 2.12: System functionality flow graph after embedding the directed graph for the sequence diagram of Figure 2.10.

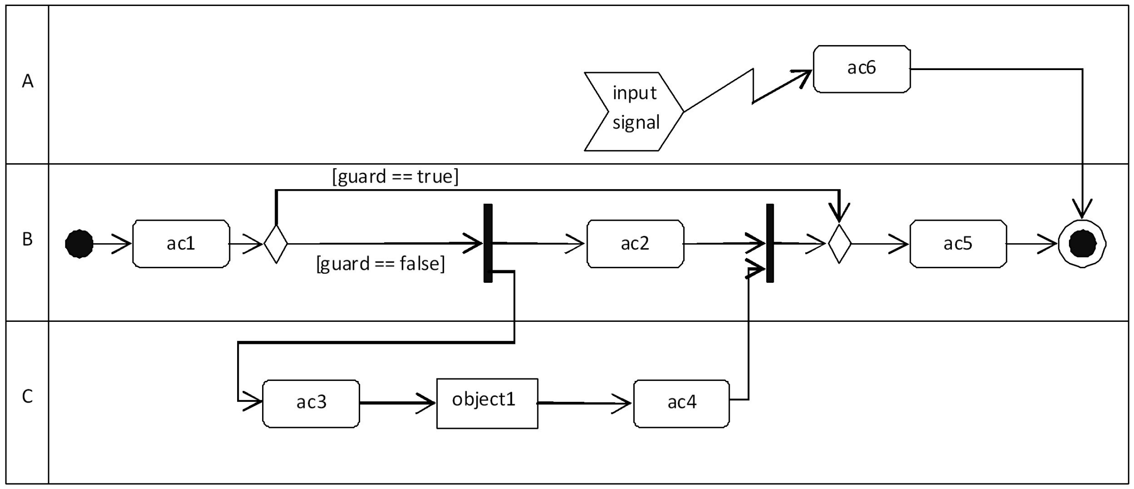


Figure 2.13: An example of a generic form activity diagram.

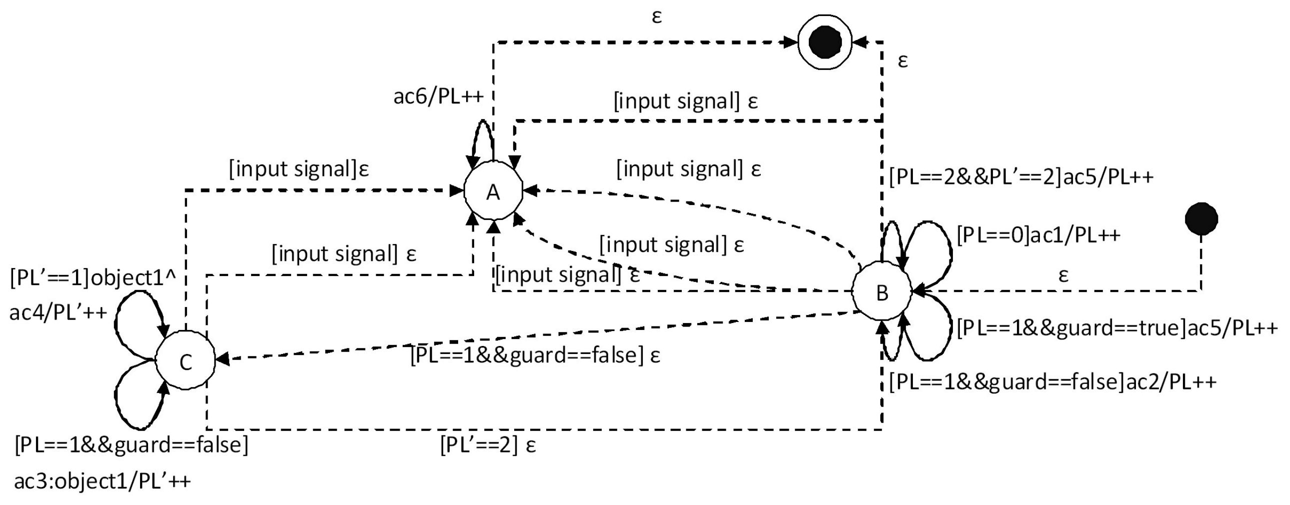


Figure 2.14: Corresponding directed graph of the activity diagram in Figure 2.13.

functionality flow graph.

Embedding Interaction Overview Diagram for Each Usecase

Interaction overview diagrams are a new type of diagram introduced in UML 2.0 that define interactions through variants of activity diagrams in a way that promotes overview of control flow. Interaction overview diagrams is very similar to an activity diagram and the only difference is that the actions in the activity diagram are replaced with UML sequence diagrams or activity invocations represented by UML sequence diagrams. The interaction overview diagrams are a very good candidates to represent the internal behavior of usecases.

In order to combine the information represented by the UML interaction overview diagram with the system functionality graph, we first propose a top-down approach to convert the interaction overview diagram into a directed graph and then we embed this directed graph with a somewhat similar approach into the system functionality flow graph.

The top-down approach starts by first treating each UML sequence diagram within the interaction overview diagram as a single vertex and converting the interaction overview diagram which is a variation of activity diagram into a directed graph. This process is explained when embedding activity diagrams into directed graphs. Next we convert each sequence diagram into a directed graph as previously explained. The final step is to connect the start and end vertices of each sequence diagram to the appropriate start and end vertices of other sequence diagrams based on the transitions in the interaction overview diagram. Figure 2.16 shows a generic interaction overview diagram and Figure 2.17 represents the directed graph that is generated after applying the explained procedure to the interaction overview diagram of Figure 2.16. Finally,

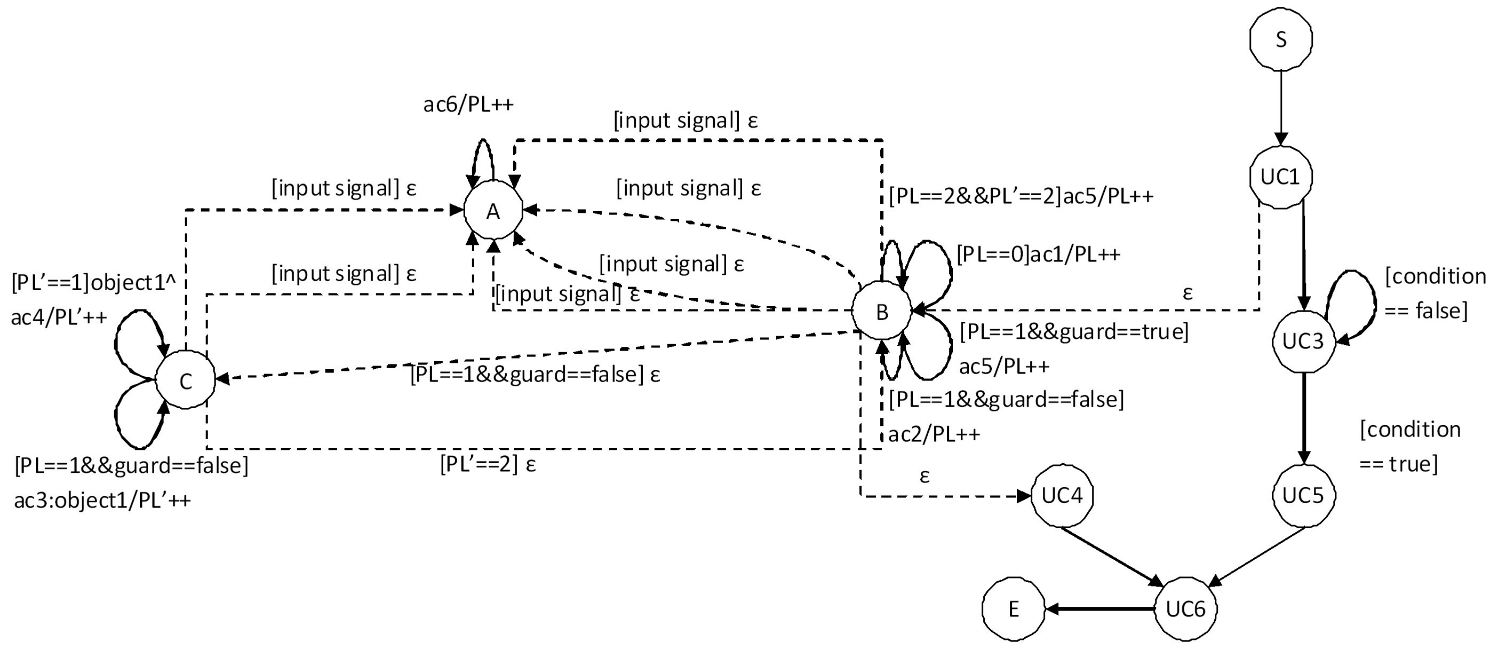


Figure 2.15: Directed graph of Figure 2.14 embedded into the system functionality flow graph.

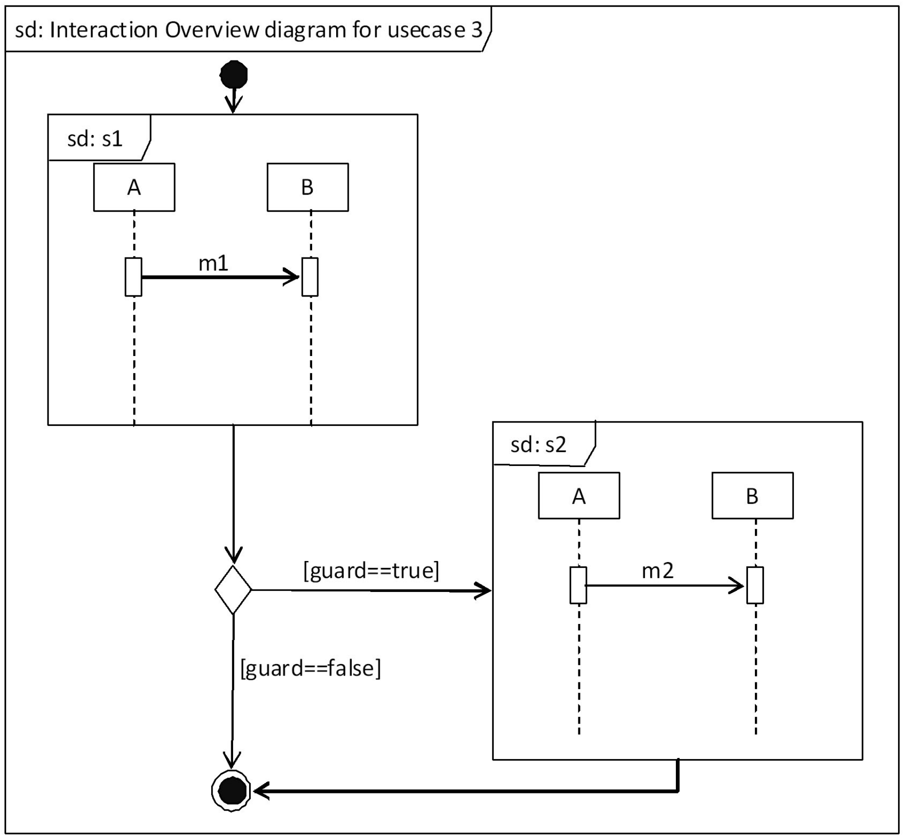


Figure 2.16: Generic form of an Interaction Overview Diagram.

the directed graph is embedded into the system functionality flow graph using a similar approach explained in the previous subsection as shown in Figure 2.18.

Embedding State Diagram for Each Individual Class

At this stage, all the vertices in the system functionality flow graph are replaced with vertices that represent individual classes in the system and from this point on we refer to this model as *System Behavioral Graph.* Now is the time to add the information from individual class state diagrams to our model. This stage is done in two consecutive steps. First, the state diagram for the class is transformed into a

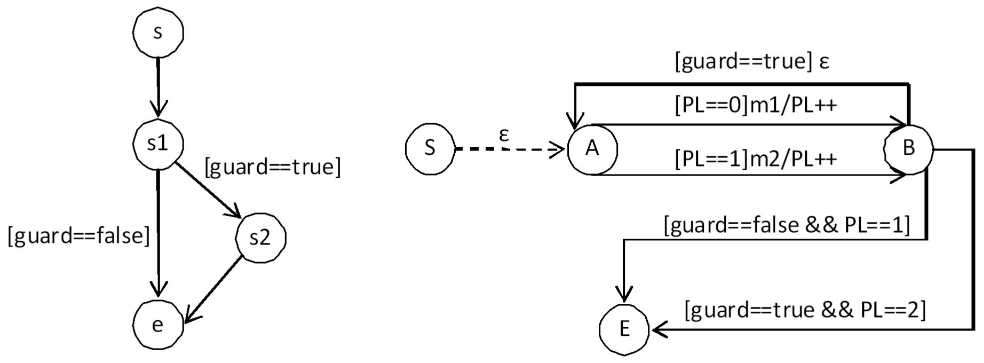


Figure 2.17: The result of transforming the interaction overview diagram of Figure 2.16 into a directed graph.

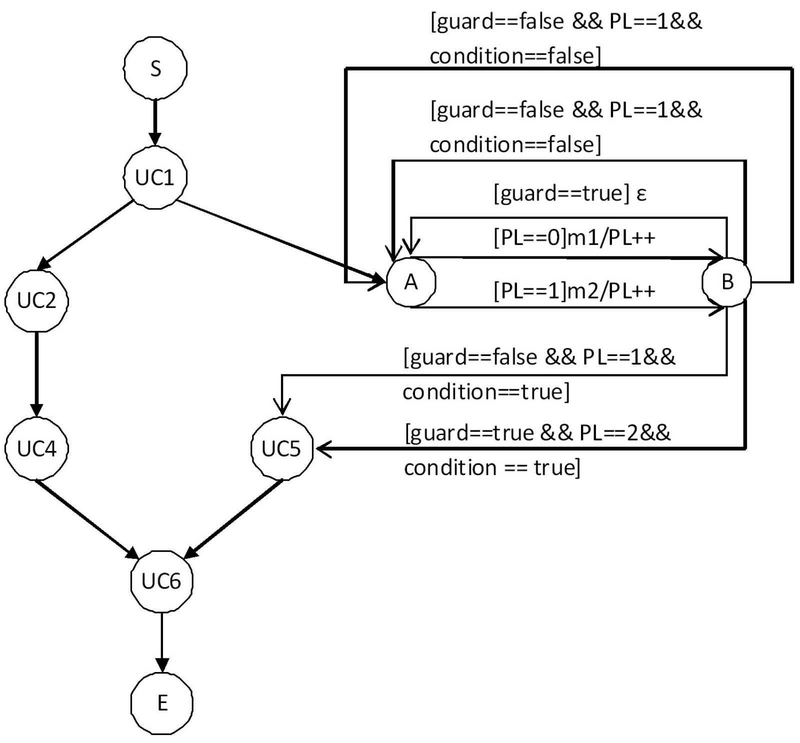

S
UC1
UC2
UC4
UC5
UC6
E
A
B
[guard==false && PL==1&& condition==false]
[guard==false && PL==1&& condition==false]
[guard==true] ε
[PL==0]m1/PL++
[PL==1]m2/PL++
[guard==false && PL==1&& condition==true]
[guard==true && PL==2&& condition == true]

Figure 2.18: The result of embedding the directed graph of the interaction overview diagram for usecase 3 in the system functionality flow graph.

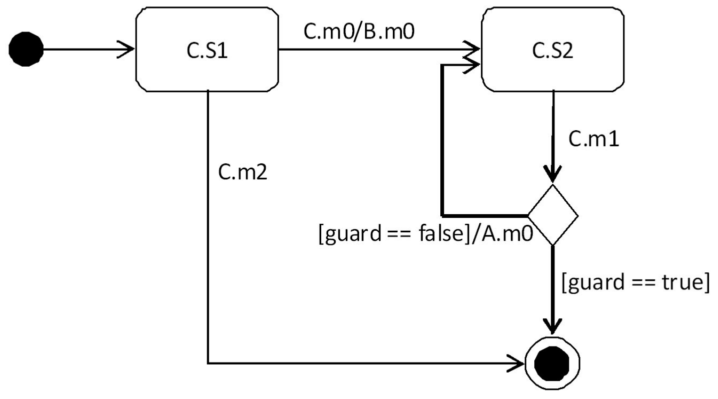


Figure 2.19: An example of a generic state diagram.

directed graph and then this graph is embedded into our model. Figure 2.19 shows a generic form of state diagram for class C. We currently only support the node types that are present in Figure 2.19. Note that we assume that the state diagram is always flattened first, i.e. all the substates are replaced by their state diagrams.

The process of converting a state diagram into a directed graph starts by creating a single vertex for each state. Then for each transition in the state diagram that starts at state $S1$ and ends in state $S2$, we add an edge entering the vertex representing $S1$ and an ϵ edge between the vertices representing $S1$ and $S2$, which will be representing the state transition from $S1$ to $S2$. Finally, in case an action is executed after the transition from $S1$ to $S2$, we represent the action with an edge that goes out of the vertex that represents $S2$. Figure 2.20 illustrates the above steps and shows the directed graph of the state diagram in Figure 2.19.

The second step is to embed the directed graph into the system behavioral graph. Assuming we are embedding the directed graph of the state diagram for class C, we first locate the vertex representing class C in the system behavioral graph. Then we replace this single vertex with the directed graph of its state diagram. Now each

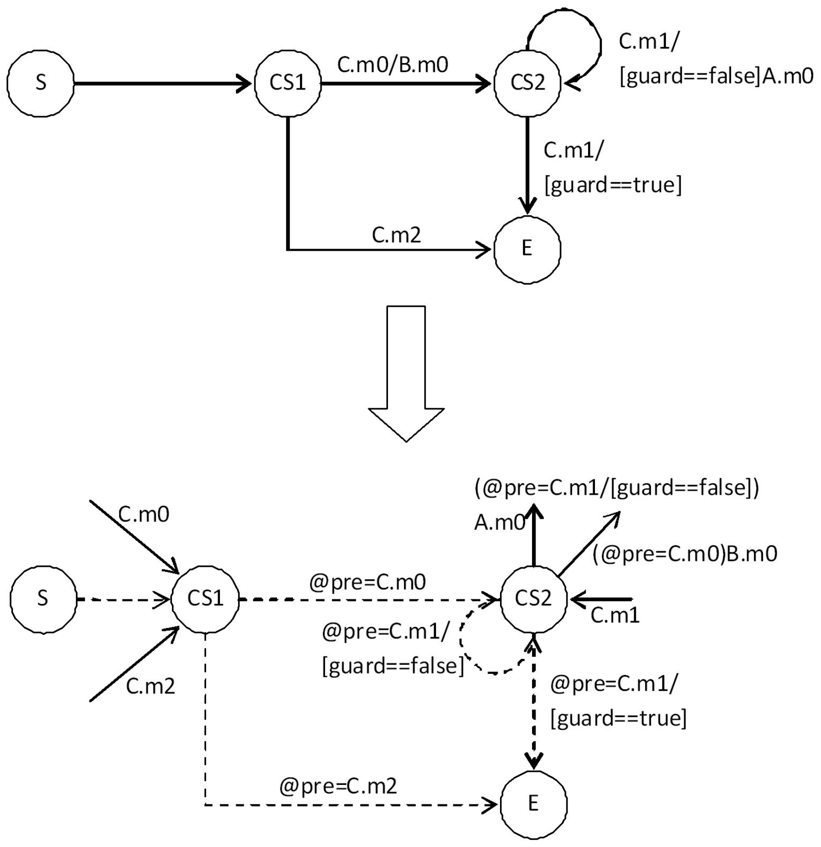


Figure 2.20: Directed graph of the generic state diagram of Figure 2.19.

incoming edge of the original vertex will be replaced by the appropriate edge in the directed graph of the state diagram based on the name of the message part of the edge. Figure 2.21 shows the result of embedding the directed graph of Figure 2.20 into the system behavioral graph.

2.2.2 Constructing the System Structure Table

As its name implies, the *System Structure Table* (*SST*) stores the information regarding the relationships among classes and their structure. These information can be classified as follows:

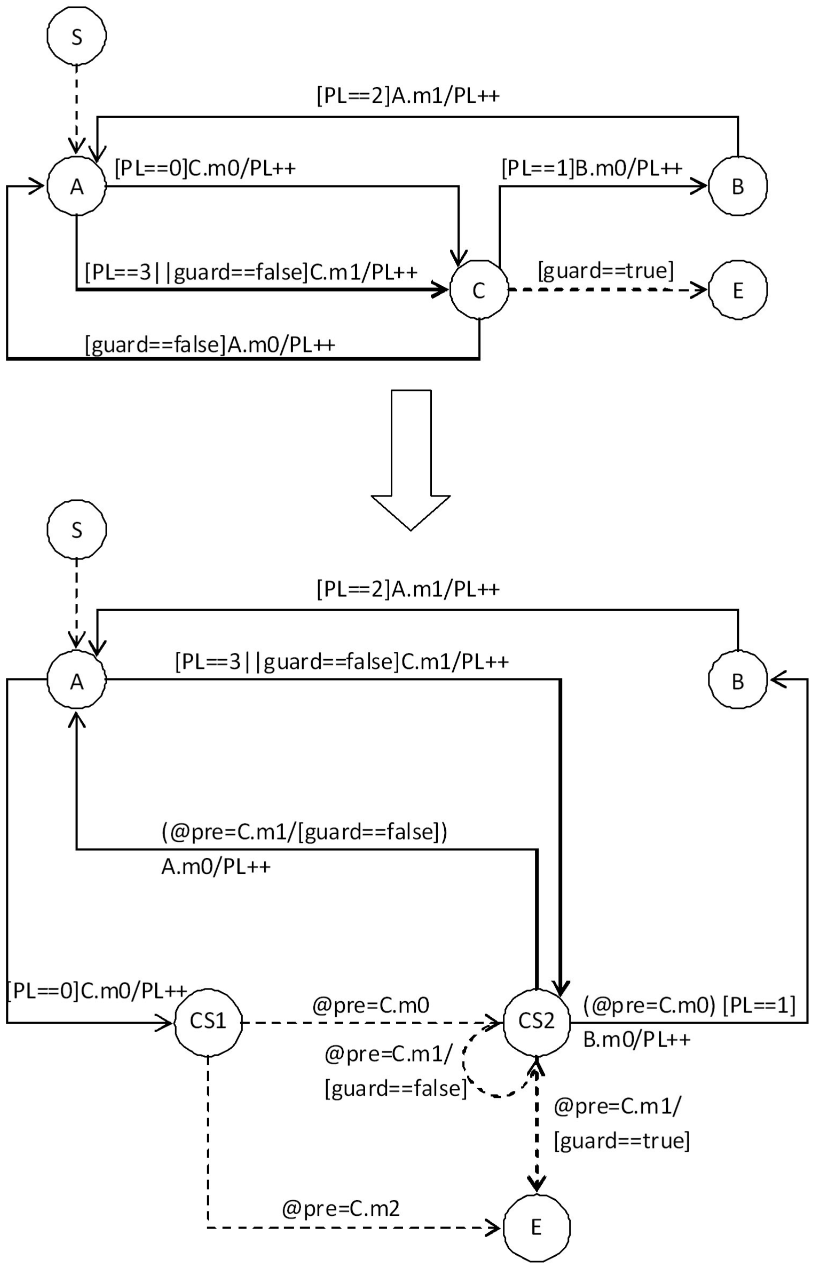


Figure 2.21: Embedding the directed graph of Figure 2.20 into the system behavioral graph.

- Instance-Level Relationships
  - Association: If class A is allowed to invoke at least one method of class B, then this relationship is represented by an association link shown as a direct arc from A to B. In case only A is allowed to invoke a method in B, the association would be considered unidirectional as opposed to a bidirectional association where B is also capable of invoking methods in A.
  - Aggregation: Aggregation occurs when both of the following conditions hold:
    1. A class is a collection or container of other classes.
    2. The contained classes do not have a strong life cycle dependency on the container. The contents will continue to exist even though their container is destroyed.
  - Composition: The strong variant of the aggregation. The contents cease to exist after the container is destroyed.
- Class Level Relationships
  - Generalization-Specialization: Represent the inheritance among classes.
  - Realization: A relationship between two model elements, in which one model element (the client) realizes the behavior that the other model element (the supplier) specifies.
- General Relationship
  - Dependency: A dependency exists between two defined classes if a change

to the definition of one would result in a change to the other. This is indicated by a dashed arrow pointing from the dependent to the independent class.

- Multiplicity: The cardinality constraints on the number of instances of each of the classes participating in an association relationship.

Realization relationships are treated as associations due to the fact that a realization is essentially a unidirectional association between supplier class and the client class. To represent this information we propose Table 2.2 as a template for the SST. For each relationship in the class diagram, we provide an entry in the SST in which the first column corresponds to the label of the relationship. The second column represents the type of the relationship (i.e. Unidirectional or Bidirectional Association, Aggregation, Composition, or Generalization). The third and fourth column represent the two classes acting as the head/container/super class and tail/content/sub class. The last two column denote the multiplicities associated with each end of the relationship. The multiplicities are represented in the form of integer intervals, i.e. $[lb, ub]$ where lb and ub correspond to the lower and upper bounds respectively.

2.2.3 Deriving the Test Suite

A test suite is a set of test cases, usually created based on some test adequacy criterion. A test adequacy criterion defines some property e.g. path coverage, statement coverage, etc., that must be covered in the entire program before testing can be stopped. A test suite that satisfies a test adequacy criterion consists of test cases that together exercise the property described by the test adequacy criterion on the entire program. Hence, it can be concluded that the size of the test suite (number of

Table 2.2: The proposed template for *System Structure Table SST*

Name of the Relationship	Type of the Relationship	Super/Head /Container class	Sub/Tail /Content class	Multiplicity of Super/-Head /Container class	Multiplicity of Sub/-Tail/Content class
A_B	(Uni/Bi)-Directional Association or Aggregation or Composition or Generalization	A	B	$[lb_A, ub_A]$	$[lb_B, ub_B]$

test cases within the test suite) is a function of the underlying test adequacy criterion. Test adequacy criteria have different fault detection potential. In general, the size of a test suite has a straight relation with the fault detection potential of its underlying adequacy criterion. A larger test suite requires the allocation of more resources in the testing phase. As a result, a tester always faces a tradeoff between a higher fault detection potential and a smaller test suite.

Generally, a test case consists of a set of inputs. The application is tested by applying the inputs and comparing the outputs of the application with the expected outputs provided by the test oracle. The test oracle can be embedded into the test case by including the set of expected outputs in the test case. The set of inputs is defined as the values for each element of the input domain.

Traditionally, the elements of the input domain were defined as variables or attributes that have an effect on the output of the program. Respectively, the input domain included all possible values for each of these variables/attributes. Based on this definition a test case TC for program p with n different input variables v_i could be formalized as: $TC = \langle (v_1, c_1), (v_2, c_2), ..., (v_{n-1}, c_{n-1}), (v_n, c_n) \rangle$, where each c_i represents a value. Test adequacy criterion is used to guide the process of assigning values to input variables so that in the end, the set of derived test cases would satisfy the criterion.

Object oriented programming introduced new concepts such as classes, objects, encapsulation, etc. Many of these new concepts are not expressible in terms of attributes. For example, the number of objects of each class or the relationships between different classes can not be expressed in terms of attributes. Hence if the above definition of test case is to be used, we should redefine the input domain to include these new features.

We separate the elements of the input domain into three groups:

1. The attributes that are input arguments of events. We refer to these as *event input attributes*(EA).

2. The attributes that appear in the guard conditions. We refer to these as *condition attributes*(CA).

3. The number of instances (objects) of each class.

An attribute can be both an event input attribute and condition attribute. We assume that the values of all condition attributes are derived from the values of event input attributes. Furthermore, we assume that the relation between each condition attribute and its corresponding event input attribute(s) is known. This becomes important because a condition attribute can only be assigned through the values of its corresponding event input attribute(s). We store all these relations in a *attribute relations* table which consists of two columns. The first column indicates the condition attribute, while the second column denotes the relation between the condition attribute and its corresponding event input attributes.

UML diagrams are grouped based on whether they express the behavior of the application or its structure. Similarly, a test case derived from UML diagrams can test the behavior or structure of the application or both. Separating the testing of behavior from structure can hide some of the faults caused by the inconsistency among behavioral and structural diagrams. Hence we choose to define our test cases based on both aspects of design. We derive a test case by following these steps:

1. Deriving the behavioral part of the test case from DIG.
2. Deriving the structural part of the test case from SST.
3. Combining the two parts into a single test case.

The behavioral and structural parts of the test case are derived using the following process:

1. Select an appropriate test adequacy criterion.
2. Identify the input domain elements.
3. Assign values to input domain elements based on the test adequacy criterion.

The following subsections explain the above process for behavioral and structural parts of the test case in more detail. Later we explain how we combine these two parts into a consistent test case.

Deriving Behavioral Part of the Test Cases

In general, the behavior of a system refers to what must happen in the system and includes the control flow, data flow, and work flow amongst the parts of the system and the environment surrounding the system. The behavior in UML is described by interaction, use case, activity or state diagrams. These diagrams are all captured in the Design Interaction Graph (DIG) as explained earlier. As a result DIG consists of all the flows that can be shown by the UML behavioral diagrams.

We derive the behavioral part of the test case based on the flows in DIG. A flow is resembled by a path in DIG that starts at the *Start* vertex and ends at the *End* vertex. We represent a path p by the tuple $\langle e_1, ...e_i, e_{i+1}, ..., e_m \rangle$ where e_1 corresponds to the first edge of the path that starts in the *Start* vertex, e_i and e_{i+1} are two consecutive edges in the path and e_m is the last edge of the path that ends in the *End* vertex. We use $l(p)$ to denote the length of path p. As mentioned before, the process of deriving the behavioral part of the test case starts by first selecting a test adequacy criterion. There already exist several test adequacy criteria that are based on control flow or data flow graphs e.g. path coverage, statement coverage, definition usage coverage. These criteria measure the test coverage based on the path in the graph that each test case traverses. Due to the similarity of DIG with data flow and control flow graphs, these criteria can easily be adopted to derive the set of test cases in DIG. Although all these criteria are widely used, they lack some important properties and therefore they are considered to be weak.

Ideally, we can define a test adequacy criterion that requires all paths in DIG to be traversed. In general, the problem of finding all paths in a graph is NP complete. This means that a test adequacy criterion that requires all paths in the DIG to be traversed is practically inapplicable. We propose the *All Condition Paths* test adequacy criterion as an alternative and we emphasize that it is ultimately up to the tester to select the appropriate test adequacy criterion based on the available resources. A path in DIG consists of zero or more path realization conditions. The *All Condition Paths* states that for all valid combinations of path realization conditions, their corresponding paths needs to be traversed.

In order to find these combinations for each conditional edge e we should find the set of all conditional edges that can be reached from e. This problem can be modified and converted to the problem of finding the transitive closure of a graph. First, we define the transitive closure of a graph and then we state how we can use the transitive closure to derive all combination of conditions.

Definition 2.2.1. *The transitive closure of graph $G = (V, E)$ is a graph $G^+ = (V, E^+)$ such that E^+ contains an edge (v, w) iff G contains a path $v \rightsquigarrow w$.*

The set of all vertices that are reachable from vertex v is known as the successor set of v and is defined as: $Succ(v) = \{w|(v, w) \in E^+\}$.

Similarly, the set of all vertices that can reach vertex v is known as the predecessor set of v and is defined as: $Pred(v) = \{u|(u, v) \in E^+\}$.

In the above definition, the successor and predecessor sets are defined for vertices but we can define the successor and predecessor sets for edges. We denote an edge that connects vertex u to vertex v by $u \rightarrow v$. We define the edge successor set of $u \rightarrow v$ based on the successor sets of u and v.

Definition 2.2.2. $Edge\_Succ(u \rightarrow v) = \{(x \rightarrow y)^+|x \in Succ(u) \wedge x \in Succ(v) \wedge y \in Succ(u) \wedge y \in Succ(v)\}$

Because we are dealing with a multi directed graph[1], we use $(x \rightarrow y)^+$ to represent the set of all edges that connect vertex x to vertex y. Similarly, we define the edge predecessor set of $u \rightarrow v$ as follows:

Definition 2.2.3. $Edge\_Pred(u \rightarrow v) = \{(x \rightarrow y)^+|x \in Pred(u) \wedge x \in Pred(v) \wedge y \in Pred(u) \wedge y \in Pred(v)\}$

[1]There can be several edges from vertex x to vertex y

The problem of finding all combination of path conditions is solved in two steps:

1. Find the edge successor and predecessor sets for all conditional edges.

2. Remove all edges that are derived by traversing an invalid path, from all successor and predecessor sets. We refer to these edges as invalid edges. Although, these edges can be reached or reach based on path traversal, the path length conditions would render these paths invalid.

3. Remove all non-conditional edges from all successor and predecessor sets. Here we interpret an edge with only path length conditions as a non-conditional edge.

4. For each pair of conditional edges e and f, derive all valid combinations of conditions based on the conditional edges that appear in $Edge\_Succ(e)$ and $Edge\_Succ(f)$ or $Edge\_Succ(e)$ and $Edge\_Pred(f)$ or $Edge\_Succ(f)$ and $Edge\_Pred(e)$.

The first step consists of two stages:

1. We derive the transitive closure graph of DIG. This can be done by executing one of the algorithms introduced in [6, 7, 10, 11, 13, 16, 18]. All of these algorithms have a running complexity of $O(|E|.|V|)$ where $|E|$ is the number of edges and $|V|$ is the number of vertices in DIG[2].

2. We derive the edge successor set for all conditional edges based on definition 2.

To avoid further complexity, we remove the invalid edges as part of the first step. We denote the path length condition value on the edge e by $cv_{PL}(e)$ and we introduce a boolean variable for each edge which is set to valid if the edge is a valid edge in

[2]$O(|E|.|V|)$ is a relaxed upper bound since DIG is a multi directed graph and $|E|$ is an over estimate of the number of edges that are really traversed in the algorithm.

the successor/predecessor edge set being constructed. We execute Algorithm 2.1 to remove all the invalid edges during the process of constructing the edge successor set. This algorithm traverses the successor edge set in a depth first search order and examines each edge in line 9 to decide whether they are valid or not. The algorithm 2.1 can be easily employed to remove the invalid edges from the edge predecessor sets by replacing all the successor edge set with predecessor edge sets and changing the condition in line 9.

Algorithm 2.1 Removing all invalid edges from the edge successor set

```
 1: removeInvalidEdges(edge $e$, edge successor set $Edge\_Succ(e)$){
 2: for all ($f \in Edge\_Succ(e)$) do
 3:    $f$.valid == false
 4: end for
 5: if ($Edge\_Succ(e) == \emptyset$) then
 6:    return
 7: end if
 8: for all ($f \in Edge\_Succ(e)$) do
 9:    if ($cv_{PL}(f) == cv_{PL}(e) + 1 \| cv_{PL}(f) == \emptyset$) then
10:       $f$.valid == true
11:       removeInvalidEdges($f$, $Edge\_Succ(f)$)
12:    end if
13: end for
14: for all ($f$.valid != true) do
15:    remove $f$ from $Edge\_Succ(e)$
16: end for
    }
```

The third step is done by a simple pass of the edge successor/predecessor sets for each conditional edge and removing the edges that are non-conditional. At this point each edge successor/predecessor set includes all the conditional edges that are valid. Now we can proceed to the final stage which gives us all valid combinations of path realization conditions. Each valid combination of conditions defines a single path in the graph, this is why the combination of conditions is known as path realization conditions. Thus, in order to find all valid path realization conditions, first we need to find all paths between every pair of conditional edges, then we derive the path realization conditions as a set of all the conditional edges along the path.

Let's say we want to find all path realization conditions between edges e and f. The set of conditional edges that are included in those path are simply those that are in the edge successor set of e and edge predecessor of f, or those that are in the edge predecessor of e and edge successor set of f: $CE(e, f) = (Edge\_Succ(e) \cap Edge\_Pred(f)) \cup$ $(Edge\_Pred(e) \cap Edge\_Succ(f))$. Each $CE(e, f)$ is a super set of a valid combination of conditional edges. This is because $CE(e, f)$ usually corresponds to several paths between e and f.

Earlier we mentioned that a test case corresponds to a path in DIG and a path is composed of edges. We define an edge e as:

$$\langle Vertex(src), Vertex(dst), guard \| \varepsilon, \{event\} \| \langle \varepsilon \rangle \rangle.$$

We divide the guard conditions appearing on an edge into two groups based on whether they are path length (PL) conditions or path realization conditions. We use $e\_g_i(p)$ to denote the path realization condition on the edge i participating in the path p. Assuming there is a path realization condition $e\_g_i(p)$ on the edge e_i, the guard condition $e\_g_i(p)$ can include a single conditional attribute or several conditional attributes.

Let $ca_k(e\_g_i(p))$ represent a single conditional attribute participating in $e\_g_i(p)$.

Definition 2.2.4. *The set of all conditional attributes that participate in $e\_g_i(p)$ is defined as:*

$CA(e\_g_i(p)) = \bigcup_k ca_k(e\_g_i(p))$.

Definition 2.2.5. $\bigcup_i CA(e\_g_i(p))$ *represents the set of all conditional attributes that appear in at least one of the path realization conditions along path* p.

Based on our all condition path criterion, we derived $CE(e, f)$ which includes conditional edges that participate in all paths between e and f. Assuming path p connects e to f or vice versa, we should replace $e\_g_i(p)$ with each individual edge in $CE(e, f)$ in definitions 4 and 5. $CE(e, f) \subset CE(g, h)$ if both e and f appear on a path between g and h. Hence if $CE(g, h)$ is computed there is no need to compute $CE(e, f)$. This results in a very important optimization which states that we should start the construction of CE sets from those pair of conditional edges that are far from each other. This can be done by using heuristic algorithms that can predict those pairs of edges that are farthest away.

As mentioned earlier, the values of the conditional attributes is derived based on their relations with their corresponding input event attributes. These input event attributes are derived from the *attribute relations* table. We treat these input event attributes as input domain elements. For each test case we should define the values for the input domain elements. The values for the input domain elements are derived based on boundary value analysis of the condition attributes they relate to. Because a single input event attribute can relate to several condition attributes it is important to detect the combinations of conditional attribute values that result in a single input event attribute having different values in the same test case. These conditional attribute value combinations are infeasible because they try to traverse a path that is not valid. This is due to the fact that we are using the super set $CE(e, f)$ of a path instead of a single path.

In addition to the set of input event attribute and value pairs (denoted by $EA(e\_g_i(p))$ and $C(e\_g_i(p))$ respectively) derived from the boundary value analysis of conditional attributes in $\bigcup_i CA(e\_g_i(p))$, the behavioral part of the test case also includes the sequence of events that are fed into the system by its surrounding environment e.g. actors, interfaces. This sequence of events is used in scenario based testing where some interference from the environment is needed. Hence we formalize the behavioral part of the test case denoted by BTC as follows:

$$BTC = \langle\langle \bigcup_i (EA(e\_g_i(p)), C(e\_g_i(p)))\rangle, \langle ev_1, ..., ev_k\rangle\rangle$$

Where $\bigcup_i (EA(e\_g_i(p)), C(e\_g_i(p)))$ represents the set of all attribute, value pairs realizing path p and $\langle ev_1, ..., ev_k\rangle$ represents the sequence of events that needs to be fed to the system during test case execution.

Deriving Structural Part of the Test Cases

We represent the structural part of the test case as a table called the *System Configuration Table* (SCT) that has four columns. The first column indicates the name of the instance of the relationship. The second column indicates the type of the relationship. Finally, the third and fourth columns state the Super/Head/Container object name, and Sub/Tail/Content object name respectively.

The process of deriving the SCT can be summarized in the following steps:

1. Based on the path that is traversed by the attribute values in the behavioral part, we identify all the edges that create an object, and the class type of the object being created. We represent an edge that creates an instance of class i with $e_c(i)$.

2. For each edge $e_c(i)$, we derive the number of times the edge is traversed in the path based on the values of condition attributes. This gives us the total number

of objects of each class type that are executed during the path traversal based on the values of the condition attributes.

3. We take one of the following approaches depending on whether we want to start the test case execution in the system configuration described by SCT or we want to end the test case in the system configuration corresponding to SCT.

   - Using the SCT as the start configuration: We create SCT using SST by specifying the number of instances of each relation, and their corresponding objects explicitly. The process of instantiation is guided by the testing criteria defined in Table 2.3.

   - Using the SCT as the end configuration: First we use a unique SCT to describe the start system configuration[3]. Next we instantiate SCT by a similar approach as described above. Then for each object type k that is instantiated along the path (identified by the behavioral part), we subtract the number of objects included in the SCT by the number of times the corresponding edge $e_c(k)$ is traversed. In case the number of objects of a class becomes negative we need to generate a new instance of SCT.

An important observation is that a behavioral part can be combined with several instances of SCT. The combined behavioral and structural parts give us the final test case.

2.2.4 Executing Test Cases on DIG

After instantiating the DIG based on the initial system configuration represented by the system configuration table (SCT) we start the process of executing the test cases

[3]This instance of SCT is independent of the test case

Table 2.3: Definition of testing criteria to derive the structural test case

Association-end multiplicity (AEM) criterion Given a test set T and a system model SM, T must cause each representative multiplicity-pair in SM to be created. (Adopted from [1].)
Generalization (GN) criterion Given a test set T and a system model SM, T must cause every specialization defined in an generalization relationship to be created. (Adopted from [1].)
Aggregation (AG) criterion Given a test set T and a system model SM, T must cause every association defined in an aggregation relationship to be created.
Composition (CM) criterion Given a test set T and a system model SM, T must cause every association defined in a composition relationship to be created.

by traversing the instantiated DIG from the start vertex using the input attribute values and events given for each test case. Similar to [14], in order to detect faults during the execution of the test cases we use an *instance table* to keep track of every object that exists in the system and their relationships amongst each other at each point in time and a *trace table* to keep track of the values of attributes, the calling objects and the messages that are passed. Together they will help us to detect design faults that are undetectable otherwise.

The *instance table* can be used to detect multiplicity violations, i.e. if the number of instances of a class can violate the multiplicity constraints represented in the class diagram. The *trace table* on the other hand can be used to detect several kinds of faults. An important observation is that because we can be at more than one vertex during the execution of a test case (due to fork nodes in the activity diagrams), it is important to keep an individual trace table for each sub-path whenever there are

Table 2.4: Structure of an *instance table*

Object_1's Name	Object_1's Class	Relationship	Object_2's Name	Object_2's Class

Table 2.5: Structure of a *trace table*

Calling Object	Message	Attribute Values

more than one traversable paths corresponding to a test case.

In the next subsection we list the different kinds of faults that are detectable. Table 2.4 and 2.5 represent the structure of the instance table and trace table respectively.

2.2.5 The Fault Model

In this section we list the kinds of faults that can be detected with our testing technique and for each kind of fault we explain how it can be detected. In general there are two groups of faults that can be detected with our approach. The first group are *static faults*, where the faults can be detected without having to executing the test cases. These faults are as follows:

- Generalization violation: This fault occurs when the assumption is that there is a generalization relationship between two classes where in fact this relationship does not appear in the class diagram. The way to detect this fault is to check every method call against its class during the construction of DIG. If a method does not belong to the class that is being called on, then either the method call is wrong or a generalization relationship is missing.

- Association violation: This fault occurs when one class uses a method from another class while no association relationship exists between the two in the class diagram. This fault can also be detected during the construction of DIG. For every edge being created in the DIG, we need to make sure that if the method call is between two different classes then the corresponding association relationship appears in the class diagram.

- Navigation violation: This fault occurs when an association relationship is directed. This means only the class at the starting point of an edge resembling a directed association relationship in class diagram can call the methods in the class at the ending point. As for the association violation fault, we can apply the same approach during the construction of DIG to detect these faults.

- Invalid calling state or called state: This kind of fault occurs when an object in a particular state is calling a method from another class while the state diagram does not allow the call in that state. Another variation of this problem appears when a method from an object in a particular state is being called while the state diagram of the object does not allow the method to be called while the object is in that state. These faults can easily be detected while embedding the state diagram into the system behavioral graph.

- Missing behavior for a call behavior in activity diagram: This fault occurs when a call behavior appears in the activity diagram while it has not been defined in any other diagrams i.e. sequence diagrams. Detecting this fault is fairly easy and it can be done during the construction of DIG. If a behavior is missing, then the transition that represents that call behavior will not be converted into a subgraph.

The second group of faults are *dynamic faults*. These faults can only be detected by executing the test cases and analyzing the contents of the *trace* and *input* tables.

These faults are as follows:

- Multiplicity violations: This fault occurs when at any given time during the running time of the system, the number of objects of a class participating in a relationship with other objects from a different class is not in the range defined by the corresponding multiplicity of that class in the class diagram. This fault can easily be detected from the instance table. Whenever an object is created or destroyed, for every relationship that the object's class type has in the class diagram, one needs to check the total number of the objects of that class per each relationship with objects of other classes. This can be done with a simple query in the instance table.

- Violation of OCL constraints: We assume that all the pre/post conditions and all the invariants are expressed using OCL. These conditions are termed as OCL constraints. OCL constraints can also be used to embed the oracle's information into the testing model. Violation of any of these constraints during the execution of a test case is rendered as the failure of that test case. These failures can be detected using the trace table. Since we store the calling object, the method being called and the values of the passing arguments and attributes before and after each edge traversal in DIG, we can easily check the corresponding OCL constraint for each object, method or attribute.

- Untraversable or non-existent path: A path is untraversable when for no combination of input attributes can it be traversed. Notice that detecting an untraversable path is very much dependent upon the test adequacy criterion and the input partitioning technique that is being used. Therefore while a test suite might not traverse a path and hence detecting it as untraversable another test suite might be able to traverse this path. Because we use every possible combination of conditions occurring on every path in DIG, our test suite will detect any untraversable path. Non-existent paths will be detected when during the execution of a test case, the system has no defined way to handle the input. This usually occurs when the designer has not handled every exception case.

Chapter 3

Case Study

3.1 The Canonical Blog

For our case study, we decided to apply our approach to a project based around the canonical blog. Examples of the blog can be found throughout the World Wide Web. A blog is [5],

> a web page that contains brief, discrete hunks of information called posts. These posts are arranged in reverse-chronological order (the most recent posts come first).

Blogging has been around since 1998 and therefore has a well defined domain. The core aspects of a blog can be narrowed down to authors, posts, and comments. This simplistic, well defined project made a perfect candidate for our case study. The core entities of a blog are easily translated into UML/OCL and no new domain specific language is needed. Also, the popularity of blogs allow this case study to be digested by a wide audience.

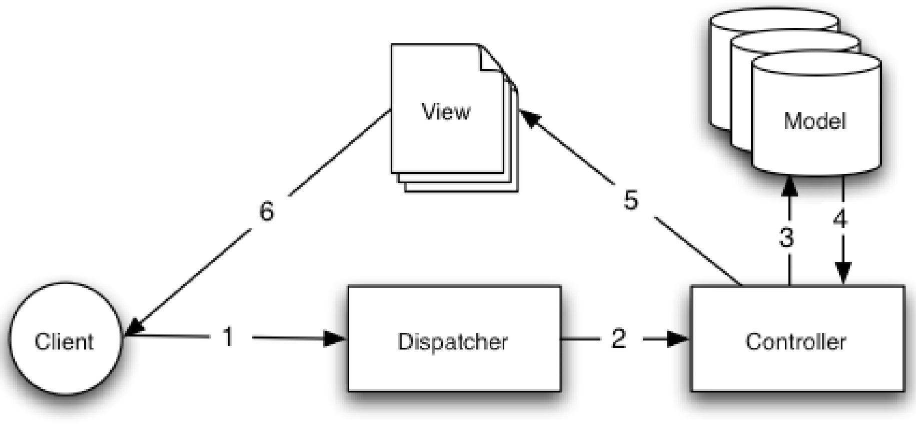


Figure 3.1: Model-View-Controller Architecture.

Before we could progress with the case study, we had to decide on a software architecture to base our designs around. We decided to follow the *Model-View-Controller* (MVC) architecture seen in Figure 3.1, and loosely base it around a Ruby on Rails implementation [8]. The various aspects of the architecture consists of:

- Model - The model represents data and the rules that govern access to and updates of this data.
- View - The view renders the contents of a model. It accesses data through the model and specifies how that data should be presented. It is the view's responsibility to maintain consistency in its presentation when the model changes.
- Controller - The controller translates interactions with the view into actions to be performed by the model. In a stand-alone GUI client, user interactions could be button clicks or menu selections, whereas in a Web application, they appear as GET and POST HTTP requests.

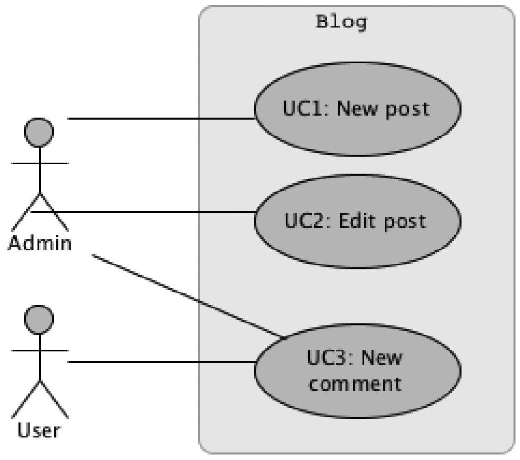


Figure 3.2: Blog - Use Cases.

In addition to Ruby on Rails, a couple other application were used for this project. We used Visual Paradigm UML Modeler Edition for the majority of the UML diagrams. This application worked extremely well for modeling in UML but lacked features to create good directed graphs. Therefore, we did all of our graph construction in Zengobi Curio, which excels in graph creation.

3.1.1 Use Cases

We started by defining our use cases. We focused on the most common blogging actions, such as creating a new post, editing a post, and creating a new comment for a post. These use cases are described in detail below in Use Cases 1-3. We could have easily incorporated additional use cases but limited it to only three to keep the diagrams understandable. These use cases and their associated actors can be seen in Figure 3.2.

UC1. CREATING A NEW POST

Goal: Successfully create a new post

Actor: Admin

Preconditions:

- User has authenticated as an Admin

Main Success Story:

1. The admin navigates to the new post view
2. The admin fills in the required post fields
3. The admin submits the new post
4. The server system validates the new post request
5. The server system stores the new post
6. The server system redirects the admin to the post view, displaying success message

Extensions:

4a. Validation fails for new post

1. The server system re-renders the new post view, displaying the validation errors to the admin

Use Case 1. Creating a new post

UC2. EDITING AN EXISTING POST

Goal: Successfully modify an existing post

Actor: Admin

Preconditions:

- User has authenticated as an Admin
- A post exists to edit

Main Success Story:

1. The admin navigates to the edit post view
2. The admin modifies the content of the existing post
3. The admin submits the modified post
4. The server system validates the modified post
5. The server system stores the updated post
6. The server system redirects the admin to the post view, displaying success message

Extensions:

4a. Validation fails for modified post

1. The server system re-renders the edit post view, displaying the validation errors to the admin

Use Case 2. Editing an existing post

UC3. CREATING A NEW COMMENT

Goal: Successfully add a comment to an existing post

Actor: User

Preconditions:

- A post exists to comment on

Main Success Story:

1. The user navigates to the post view
2. The user fills in the required comment fields
3. The user submits the new comment
4. The server system validates the new comment request
5. The server system stores the new comment
6. The server system redirects the user to the post view, displaying new comment and success message

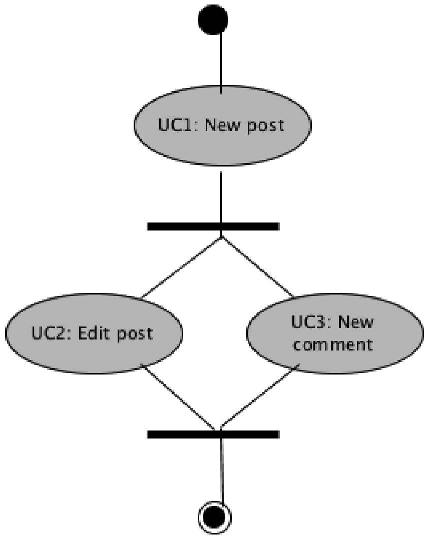
Extensions:

4a. Validation fails for new comment

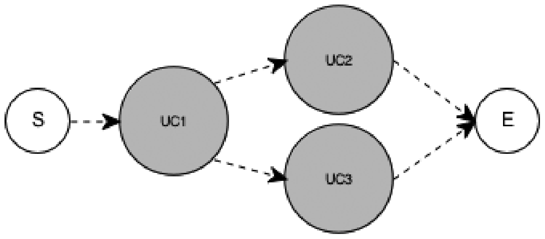
1. The server system re-renders the post view, displaying the validation errors to the user

Use Case 3. Creating a new comment

These use cases were then converted into the Use Case Sequential Constraint diagram, and finally the constraint diagrams was converted into the *System Functionality Flow Graph* (SFFG). These two diagrams are shown in Figure 3.3. It is beneficial



(a) Use Case Sequential Constraint



(b) System Functionality Flow Graph

Figure 3.3: Blog - Use Case Conversion.

to note that a post must exist (UC1) before it can be edited (UC2) or commented on (UC3). Once a post has been created, editing and commenting can be done in parallel.

3.1.2 Class Diagram

When constructing the class diagram, we chose to omit the View portion of the MVC architecture since they are not classes, just templates for rending content in the browser. All diagrams follow Ruby on Rails conventions, such as how class names are organized. As illustrated in Figure 3.4, the classes defined for our models were User, Post, and Comment. Each model has an associated controller which is similarly named, UsersController, PostsController, and CommentsController. There is only a single instance of each controller which interacts with many models. The communication between the controllers and models is unidirectional ($Controller \Rightarrow Model$).

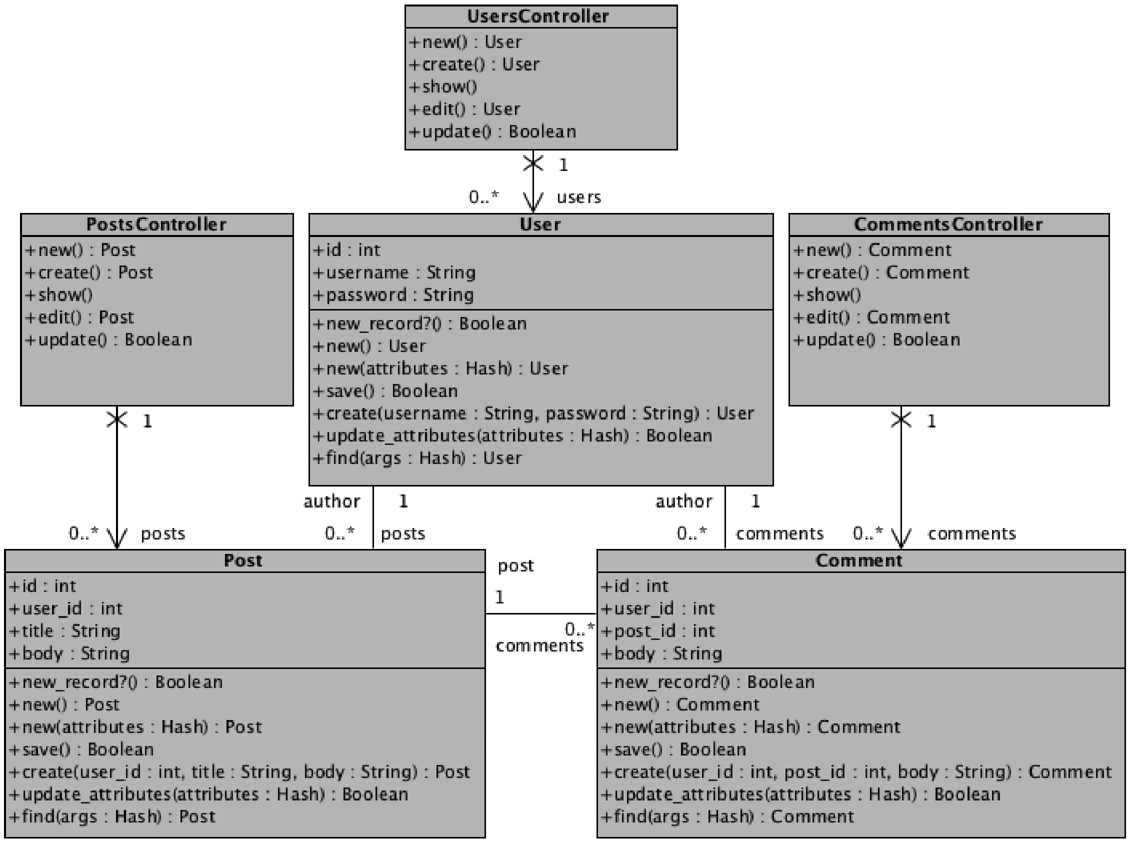


Figure 3.4: Blog - Class Diagram.

Models has a one-to-many relationship with every other model.

3.1.3 Object Constraint Language

For this case study we decided to impose constraints on all the models using the Object Constraint Language (OCL). Constraints were not added to controllers since they simply act as a intermediary between the user/browser and the models in this specific project. The constraints are outlined below in Tables 3.1.3, 3.1.3, and 3.1.3. We chose to seed our case study with OCL faults to ensure this type of a fault was properly detected by our approach. The seeded faults were:

1. Use Case 1 - Creating a new post with an invalid Post.user_id attribute
2. Use Case 2 - Editing an existing post with an empty Post.title attribute
3. Use Case 3 - Commenting on a Post that doesnt exist (invalid Comment.post_id)

3.1.4 State Diagrams

State diagrams were created for all of the controllers and models, as well as a diagram from the user/browser perspective. This additional state diagram was needed to fully model the interactions in the MVC architecture. Figure 3.5 illustrates the diagrams for the models and user/browser, while Figure 3.6 illustrates the diagrams for the controllers. The user/browser diagram only has a single state which represents the waiting for a given action to occur. Two examples of actions are waiting for a page to render in the browser, and waiting for a HTTP request to be sent to the blog application. Controllers only have a single state, where they are waiting for incoming HTTP requests. Models on the other hand, contain two states, 1) a new record, not yet saved to the database, and 2) an existing record which is stored in the database.

3.1.5 Sequence Diagrams

Taking our three previously defined use cases, we converted their logic into sequence diagrams, illustrated in Figure 3.7. We elected to go with sequence diagrams over the other approaches proposed, due to the some of the aspects of object oriented languages this diagram type provides, such as object lifelines, and the ability to show method returns.

```
context User
  inv valid_id: id > 0
  inv valid_username: username->size() > 0
  inv valid_password: password->size() > 0

context User::new()
  post let message : oclMessage = self^new_record()
  message.hasReturned()
  and
  message.result() = true

context User::save(): Boolean
  // save successful
  post let message: oclMessage = self^create(self.username, self.password)
    message.hasReturned()
    and
    message.result() = false

context User::create(username : String, password : String): User
  pre valid_post: username->size() > 0 and
                  password->size() > 0
  // valid existing record
  post  let message1 : oclMessage = self^createNew(username, password)
    message1.hasReturned()
    and
    let message2 : oclMessage = self^new_record();
    message2.result()  = false

context User::update_attributes(attributes : Hash): Boolean
  pre valid_attrs: attributes.username->size() > 0 and
                   attributes.password->size() > 0
  post update_successful:
    post let message: oclMessage = self^update_attributes(attributes)
    message.result() = true

context User::find(args : Hash): User
  pre valid_args:
    if args->exists(id) then
      args.id > 0
    endif
    and
    if args->exists(username) then
      args.username->size() > 0
    endif
    and
    if args->exists(password) then
      args.password->size)() > 0
    endif
  post user_found:
    post let message: oclMessage = self^find(args)
    let result = message.result();
    and
    result != nil
```

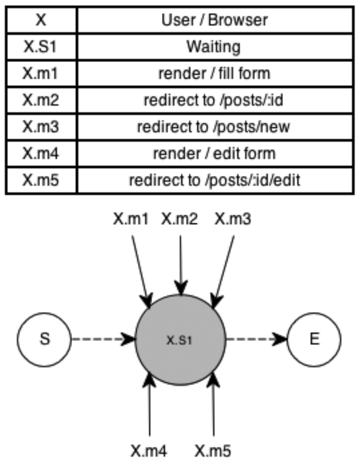
Table 3.1: OCL - User Constraints

```
context Post
  inv valid_id: id > 0
  inv valid_user: user_id > 0
  inv valid_title: title->size() > 0
  inv valid_body: body->size() > 0

context Post::new(): Post
   post let message : oclMessage = self^new_record()
  message.hasReturned()
  and
  message.result() = true

context Post::save(): Boolean
  post let message: oclMessage = self^create(self.user_id, self.body)
    message.hasReturned() and
    message.result() = false

context Post::create(user_id : Integer, title : String, body : String): Post
  pre valid_post: user_id > 0 and
                  title->size() > 0 and
                  body->size() > 0
  // valid existing record
  post  let message1 : oclMessage = self^createNew(user_id, title, body)
    message1.hasReturned()
    and
    let message2 : oclMessage = self^new_record();
    message2.result  = false

context Post::update_attributes(attributes : Hash): Boolean
  pre valid_attrs: attributes.title->size() > 0 and
                   attributes.body->size() > 0
  // update_successful
  post : let message : oclMessage = self-^update_attributes(attributes)
  oclMessage.hasReturned() and
  oclMessage.result() = true

context Post::find(args : Hash): Post
  pre valid_args:
    if args->exists(id) then
      args.id > 0
    endif
    and
    if args->exists(user_id) then
      args.user_id > 0
    endif
    and
    if args->exists(title) then
      args.title->size() > 0
    endif
    and
    if args->exists(body) then
      args.body->size)() > 0
    endif
  post post_found:
    post let message: oclMessage = self^find(args)
    let result = message.result();
    and
    result != nil
```

Table 3.2: OCL - Post Constraints

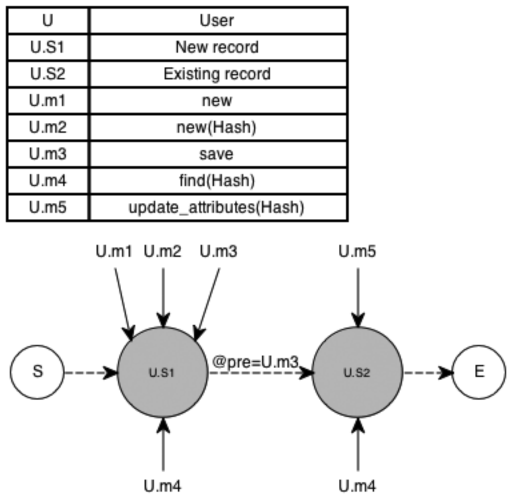
```
context Comment
  inv valid_id: id > 0
  inv valid_user: user_id > 0
  inv valid_post: post_id > 0
  inv valid_body: body->size() > 0

context Comment::new(): Comment
  post let message : oclMessage = self^new_record()
  message.hasReturned()
  and
  message.result() = true

context Comment::save(): Boolean
  post let message: oclMessage = self^create(self.user_id, self.body)
    message.hasReturned() and
    message.result() = false

context Comment::create(user_id : Integer, post_id : Integer, body : String): Comment
  pre valid_comment: user_id > 0 and
                     post_id > 0 and
                     body->size() > 0
  post valid_existing_record:
    post  let message1 : oclMessage = self^createNew(user_id, post_id, body)
    message1.hasReturned()
    and
    let message2 : oclMessage = self^new_record();
    message2.result  = false

context Comment::update_attributes(attributes : Hash): Boolean
  pre valid_attrs: attributes.body->size() > 0
  // update_successful:
  post : let message : oclMessage = self-^update_attributes(attributes)
    oclMessage.hasReturned() and
    oclMessage.result() = true

context Comment::find(args : Hash): Comment
  pre valid_args:
    if args->exists(id) then
      args.id > 0
    endif
    and
    if args->exists(user_id) then
      args.user_id > 0
    endif
    and
    if args->exists(post_id) then
      args.post_id > 0
    endif
    and
    if args->exists(body) then
      args.body->size)() > 0
    endif
  post comment_found:
    post let message: oclMessage = self^find(args)
    let result = message.result();
    and
    result != nil
```

Table 3.3: OCL - Comment Constraints



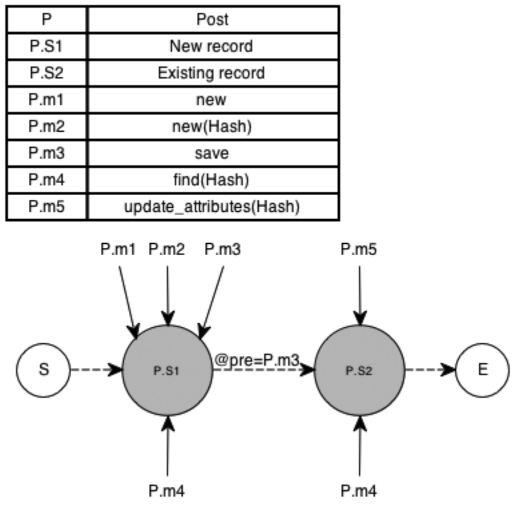
X	User / Browser
X.S1	Waiting
X.m1	render / fill form
X.m2	redirect to /posts/:id
X.m3	redirect to /posts/new
X.m4	render / edit form
X.m5	redirect to /posts/:id/edit

(a) User/Browser



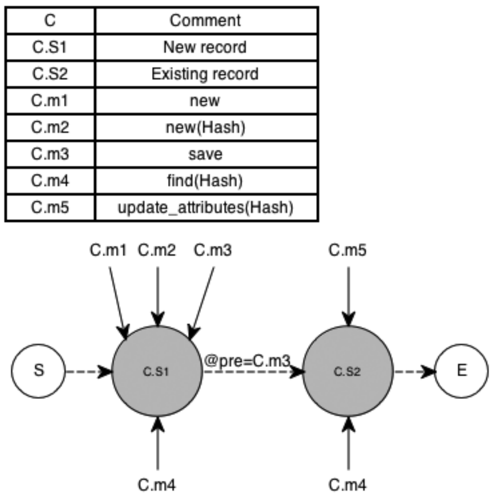
U	User
U.S1	New record
U.S2	Existing record
U.m1	new
U.m2	new(Hash)
U.m3	save
U.m4	find(Hash)
U.m5	update_attributes(Hash)

(b) User



P	Post
P.S1	New record
P.S2	Existing record
P.m1	new
P.m2	new(Hash)
P.m3	save
P.m4	find(Hash)
P.m5	update_attributes(Hash)

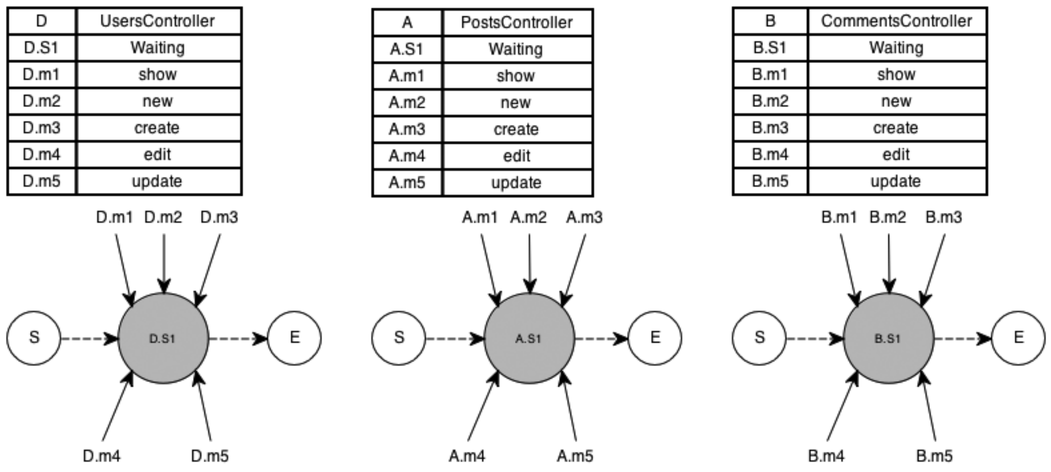
(c) Post



C	Comment
C.S1	New record
C.S2	Existing record
C.m1	new
C.m2	new(Hash)
C.m3	save
C.m4	find(Hash)
C.m5	update_attributes(Hash)

(d) Comment

Figure 3.5: Blog - Model State Diagrams.



D	UsersController
D.S1	Waiting
D.m1	show
D.m2	new
D.m3	create
D.m4	edit
D.m5	update

A	PostsController
A.S1	Waiting
A.m1	show
A.m2	new
A.m3	create
A.m4	edit
A.m5	update

B	CommentsController
B.S1	Waiting
B.m1	show
B.m2	new
B.m3	create
B.m4	edit
B.m5	update

(a) UsersController (b) PostsController (c) CommentsController

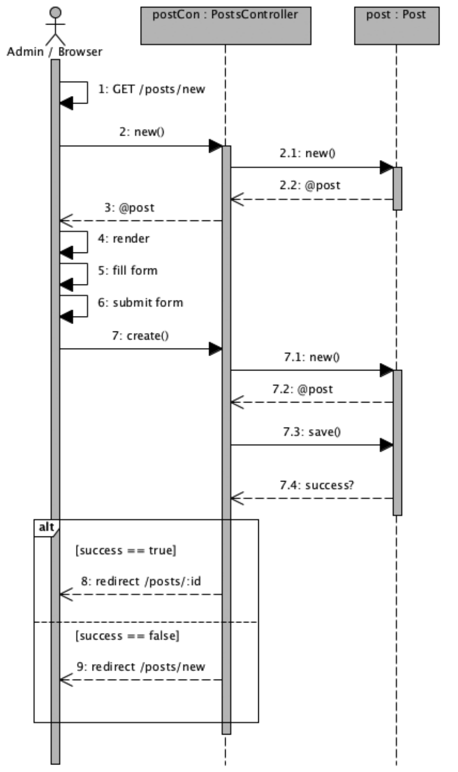
Figure 3.6: Blog - Controller State Diagrams.

3.1.6 System Structure Table

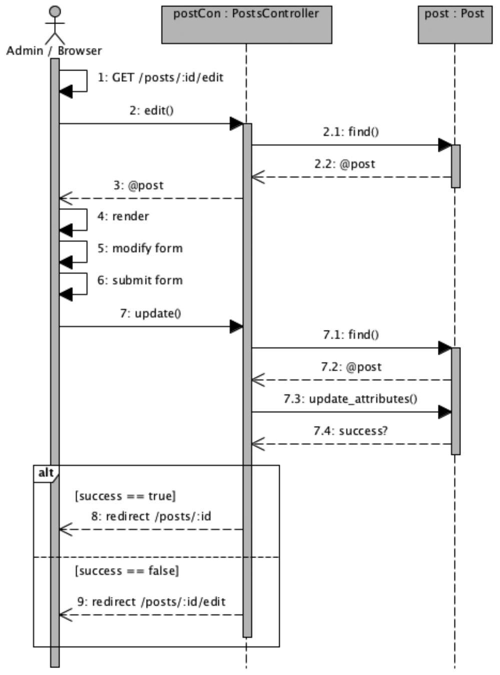
Before moving onto DIG construction, we built the SST based on the relationships between classes and their structure. Table 3.4 reveals these relationships. This was a fairly straightforward process due to the simplicity of the domain.

3.1.7 System Behavioral Graph

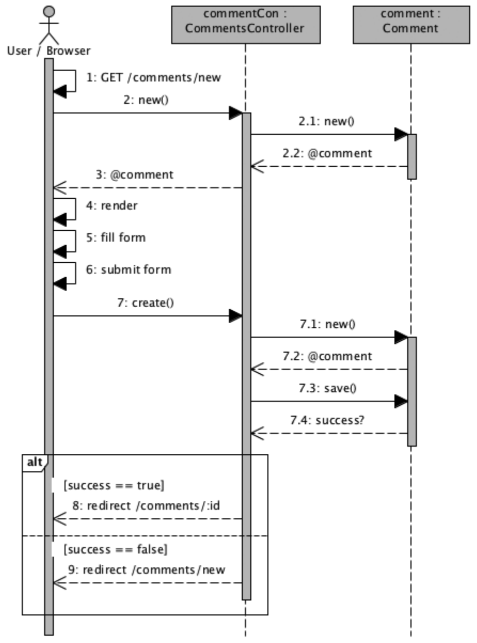
The next step consisted of flattening the UML sequence diagrams into a directed graph and embed them into the SFFG. To simplify this process, we actually created a separate SBG per use case, representing all paths for the given use case. The results of these separate SBGs are illustrated in Tables 3.5, 3.6, and 3.7, along with the associated Figures 3.8, 3.9, and 3.10. Once all three SBGs were created, they were embedded into the SFFG. This stage was the most time consuming process in our approach and quickly resulted in fairly large diagrams. In the diagrams, solid lines are method calls and dashed lines represent the method returns, along with any



(a) UC1: New post



(b) UC2: Edit post



(c) UC3: New comment

Figure 3.7: Blog - Sequence Diagrams.

Relationship Name	Relationship Type	SHC Class	STC Class	Mult. of SHC	Mult. of STC
UsersController_User	Unidirectional Association	UsersController	User	[1,1]	[0,*]
PostsController_Post	Unidirectional Association	PostsController	Post	[1,1]	[0,*]
CommentsController_Comment	Unidirectional Association	CommentsController	Comment	[1,1]	[0,*]
User_Post	Bidirectional Association	User	Post	[0,*]	[0,*]
User_Comment	Bidirectional Association	User	Comment	[0,*]	[0,*]
Post_Comment	Bidirectional Association	Post	Comment	[0,*]	[0,*]

Table 3.4: Blog - System Structure Table.

X	Admin/Browser
A	PostsController
P	Post

Table 3.5: Blog - UC1 SBG Class Mapping.

X	Admin/Browser
A	PostsController
P	Post

Table 3.6: Blog - UC2 SBG Class Mapping.

returned values.

3.1.8 Design Interaction Graph

The final step in the DIG construction was to embed our state diagrams into our previously created SBGs. This was an easy process due to most classes only having

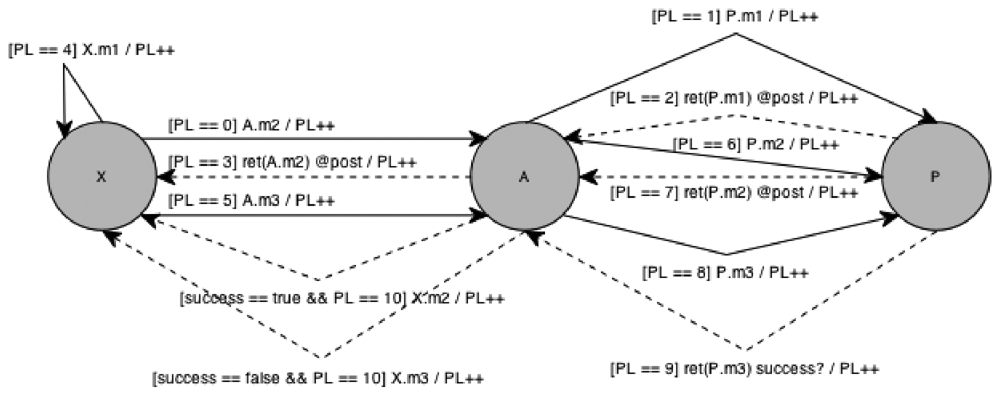

[PL == 1] P.m1 / PL++
[PL == 4] X.m1 / PL++
[PL == 2] ret(P.m1) @post / PL++
[PL == 0] A.m2 / PL++
[PL == 6] P.m2 / PL++
[PL == 3] ret(A.m2) @post / PL++
X
A
P
[PL == 7] ret(P.m2) @post / PL++
[PL == 5] A.m3 / PL++
[PL == 8] P.m3 / PL++
[success == true && PL == 10] X.m2 / PL++
[success == false && PL == 10] X.m3 / PL++
[PL == 9] ret(P.m3) success? / PL++

Figure 3.8: Blog - UC1 System Behavioral Graph.

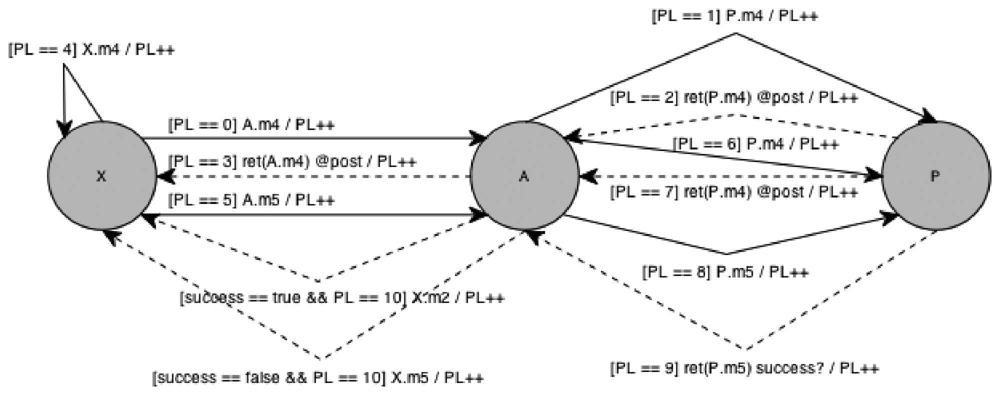

[PL == 1] P.m4 / PL++
[PL == 4] X.m4 / PL++
[PL == 2] ret(P.m4) @post / PL++
[PL == 0] A.m4 / PL++
[PL == 6] P.m4 / PL++
[PL == 3] ret(A.m4) @post / PL++
X
A
P
[PL == 7] ret(P.m4) @post / PL++
[PL == 5] A.m5 / PL++
[PL == 8] P.m5 / PL++
[success == true && PL == 10] X.m2 / PL++
[success == false && PL == 10] X.m5 / PL++
[PL == 9] ret(P.m5) success? / PL++

Figure 3.9: Blog - UC2 System Behavioral Graph.

X	User/Browser
B	CommentsController
C	Comment

Table 3.7: Blog - UC3 SBG Class Mapping.

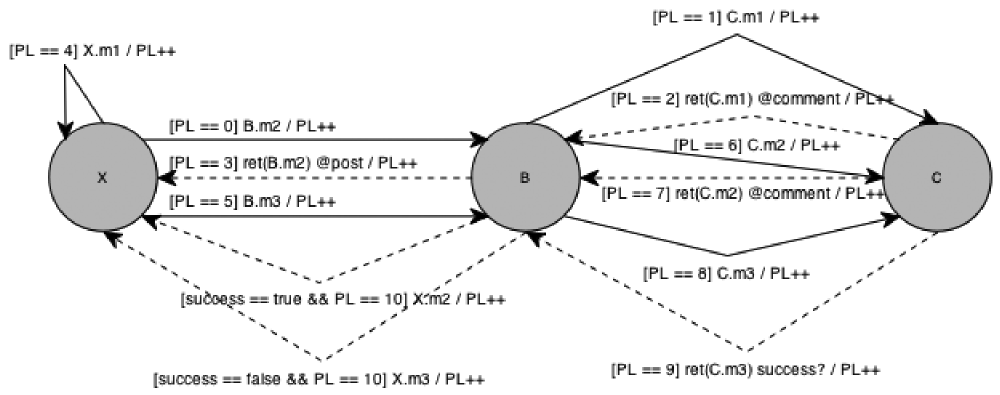

[PL == 4] X.m1 / PL++
[PL == 1] C.m1 / PL++
[PL == 2] ret(C.m1) @comment / PL++
[PL == 0] B.m2 / PL++
[PL == 6] C.m2 / PL++
[PL == 3] ret(B.m2) @post / PL++
X
B
C
[PL == 7] ret(C.m2) @comment / PL++
[PL == 5] B.m3 / PL++
[PL == 8] C.m3 / PL++
[success == true && PL == 10] X.m2 / PL++
[success == false && PL == 10] X.m3 / PL++
[PL == 9] ret(C.m3) success? / PL++

Figure 3.10: Blog - UC3 System Behavioral Graph.

one or two states. The complexity of this step increases dramatically with the more states represented in the classes. The resulting DIG is illustrated in Figure 3.11.

3.1.9 Hybrid Instance-Trace Table

During table construction, we noticed were able to merge the contents of the instance and trace tables into a single table for simplicity and easier reference to the data collected. These hybrid tables, in combination with the formula used to determine the transitive closure of a graph seen in Definition 2.2.1, enumerate all unique paths through the DIG, covering all conditional statements. For each unique path, a separate hybrid table was constructed. These tables are shown in Tables 3.8, 3.9, 3.10, 3.11, 3.12, and 3.13. The tables assist in detecting multiplicity violations within the DIG, as well as pinpoint Object Constraint Language (OCL) violations. The following is the list of the six unique paths we discovered, grouped by the related use case scenario. Each ⇒ symbol signifies a method call or a transition to another state. In both the path lists and hybrid tables, the boldface font denotes the path differences between the related cases. As an example, Use Case 1 / Path 1 and Use Case 1 / Path

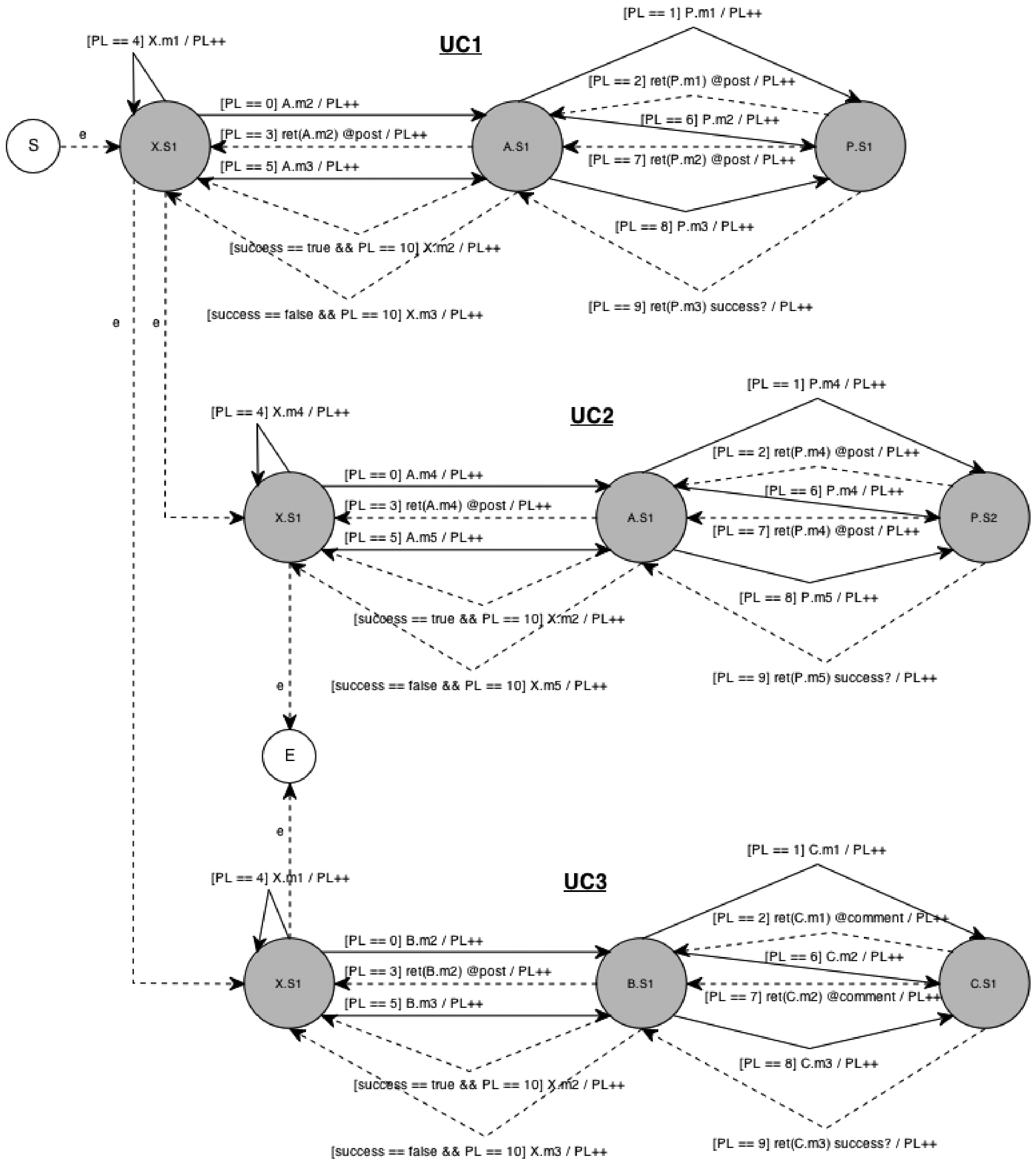

UC1
[PL == 4] X.m1 / PL++
[PL == 1] P.m1 / PL++
[PL == 2] ret(P.m1) @post / PL++
[PL == 0] A.m2 / PL++
[PL == 6] P.m2 / PL++
[PL == 3] ret(A.m2) @post / PL++
[PL == 7] ret(P.m2) @post / PL++
[PL == 5] A.m3 / PL++
S
e
X.S1
A.S1
P.S1
[PL == 8] P.m3 / PL++
[success == true && PL == 10] X.m2 / PL++
[success == false && PL == 10] X.m3 / PL++
[PL == 9] ret(P.m3) success? / PL++
e
e
UC2
[PL == 4] X.m4 / PL++
[PL == 1] P.m4 / PL++
[PL == 2] ret(P.m4) @post / PL++
[PL == 0] A.m4 / PL++
[PL == 6] P.m4 / PL++
[PL == 3] ret(A.m4) @post / PL++
[PL == 7] ret(P.m4) @post / PL++
[PL == 5] A.m5 / PL++
X.S1
A.S1
P.S2
[PL == 8] P.m5 / PL++
[success == true && PL == 10] X.m2 / PL++
[success == false && PL == 10] X.m5 / PL++
[PL == 9] ret(P.m5) success? / PL++
e
E
e
UC3
[PL == 4] X.m1 / PL++
[PL == 1] C.m1 / PL++
[PL == 2] ret(C.m1) @comment / PL++
[PL == 0] B.m2 / PL++
[PL == 6] C.m2 / PL++
[PL == 3] ret(B.m2) @post / PL++
[PL == 7] ret(C.m2) @comment / PL++
[PL == 5] B.m3 / PL++
X.S1
B.S1
C.S1
[PL == 8] C.m3 / PL++
[success == true && PL == 10] X.m2 / PL++
[success == false && PL == 10] X.m3 / PL++
[PL == 9] ret(C.m3) success? / PL++

Figure 3.11: Blog - Design Interaction Graph.

2 differ towards the end of their paths. Path 1 transitions to state X.S1 through the method X.m2, whereas Path 2 transitions to state X.S1 through the method X.m3.

- Use Case 1, Path 1 - New post, successful

  $S \Rightarrow e \Rightarrow$

  $X.S1 \Rightarrow A.m2 \Rightarrow$

  $A.S1 \Rightarrow P.m1 \Rightarrow$

  $P.S1 \Rightarrow ret(P.m1) \Rightarrow$

  $A.S1 \Rightarrow ret(A.m2) \Rightarrow$

  $X.S1 \Rightarrow X.m1 \Rightarrow$

  $X.S1 \Rightarrow A.m3 \Rightarrow$

  $A.S1 \Rightarrow P.m2 \Rightarrow$

  $P.S1 \Rightarrow ret(P.m2) \Rightarrow$

  $A.S1 \Rightarrow P.m3 \Rightarrow$

  $P.S1 \Rightarrow ret(P.m3) \Rightarrow$

  $A.S1 \Rightarrow \mathbf{X.m2} \Rightarrow$

  $X.S1$

- Use Case 1, Path 2 - New post, failed

  $S \Rightarrow e \Rightarrow$

  $X.S1 \Rightarrow A.m2 \Rightarrow$

  $A.S1 \Rightarrow P.m1 \Rightarrow$

  $P.S1 \Rightarrow ret(P.m1) \Rightarrow$

  $A.S1 \Rightarrow ret(A.m2) \Rightarrow$

  $X.S1 \Rightarrow X.m1 \Rightarrow$

  $X.S1 \Rightarrow A.m3 \Rightarrow$

$A.S1 \Rightarrow P.m2 \Rightarrow$

$P.S1 \Rightarrow ret(P.m2) \Rightarrow$

$A.S1 \Rightarrow P.m3 \Rightarrow$

$P.S1 \Rightarrow ret(P.m3) \Rightarrow$

$A.S1 \Rightarrow \mathbf{X.m3} \Rightarrow$

$X.S1$

- Use Case 2, Path 1 - Edit post, successful

  $UC1\_* \Rightarrow e \Rightarrow$

  $X.S1 \Rightarrow A.m4 \Rightarrow$

  $A.S1 \Rightarrow P.m4 \Rightarrow$

  $P.S2 \Rightarrow ret(P.m4) \Rightarrow$

  $A.S1 \Rightarrow ret(A.m4) \Rightarrow$

  $X.S1 \Rightarrow X.m4 \Rightarrow$

  $X.S1 \Rightarrow A.m5 \Rightarrow$

  $A.S1 \Rightarrow P.m4 \Rightarrow$

  $P.S2 \Rightarrow ret(P.m4) \Rightarrow$

  $A.S1 \Rightarrow P.m5 \Rightarrow$

  $P.S2 \Rightarrow ret(P.m5) \Rightarrow$

  $A.S1 \Rightarrow \mathbf{X.m2} \Rightarrow$

  $X.S1$

- Use Case 2, Path 2 - Edit post, failed

  $UC1\_* \Rightarrow e \Rightarrow$

  $X.S1 \Rightarrow A.m4 \Rightarrow$

  $A.S1 \Rightarrow P.m4 \Rightarrow$

$P.S2 \Rightarrow ret(P.m4) \Rightarrow$

$A.S1 \Rightarrow ret(A.m4) \Rightarrow$

$X.S1 \Rightarrow X.m4 \Rightarrow$

$X.S1 \Rightarrow A.m5 \Rightarrow$

$A.S1 \Rightarrow P.m4 \Rightarrow$

$P.S2 \Rightarrow ret(P.m4) \Rightarrow$

$A.S1 \Rightarrow P.m5 \Rightarrow$

$P.S2 \Rightarrow ret(P.m5) \Rightarrow$

$A.S1 \Rightarrow \mathbf{X.m5} \Rightarrow$

$X.S1$

- Use Case 3, Path 1 - New comment, successful

  $UC1\_* \Rightarrow e \Rightarrow$

  $X.S1 \Rightarrow B.m2 \Rightarrow$

  $B.S1 \Rightarrow C.m1 \Rightarrow$

  $C.S1 \Rightarrow ret(C.m1) \Rightarrow$

  $B.S1 \Rightarrow ret(B.m2) \Rightarrow$

  $X.S1 \Rightarrow X.m1 \Rightarrow$

  $X.S1 \Rightarrow B.m3 \Rightarrow$

  $B.S1 \Rightarrow C.m2 \Rightarrow$

  $C.S1 \Rightarrow ret(C.m2) \Rightarrow$

  $B.S1 \Rightarrow C.m3 \Rightarrow$

  $C.S1 \Rightarrow ret(C.m3) \Rightarrow$

  $B.S1 \Rightarrow \mathbf{X.m2} \Rightarrow$

  $X.S1$

- Use Case 3, Path 2 - New comment, failed

  $UC1\_* \Rightarrow e \Rightarrow$

  $X.S1 \Rightarrow B.m2 \Rightarrow$

  $B.S1 \Rightarrow C.m1 \Rightarrow$

  $C.S1 \Rightarrow ret(C.m1) \Rightarrow$

  $B.S1 \Rightarrow ret(B.m2) \Rightarrow$

  $X.S1 \Rightarrow X.m1 \Rightarrow$

  $X.S1 \Rightarrow B.m3 \Rightarrow$

  $B.S1 \Rightarrow C.m2 \Rightarrow$

  $C.S1 \Rightarrow ret(C.m2) \Rightarrow$

  $B.S1 \Rightarrow C.m3 \Rightarrow$

  $C.S1 \Rightarrow ret(C.m3) \Rightarrow$

  $B.S1 \Rightarrow \mathbf{X.m3} \Rightarrow$

  $X.S1$

3.1.10 Detected Faults

This section will briefly recap on each possible fault within the fault model described within this paper and summarize our own fault discoveries within the blog project.

Static Faults

1. Generalization violation

   - Detection: Check every method call against its class during the construction of the DIG. If a method does not belong to class then 1) the method call is wrong or 2) a generalization relationship is missing.

Calling Object	Message	Parameters	Called Object
X User/Browser	A.m2		A PostsController
A PostsController	P.m1		P Post
P Post	ret(P.m1)	@post	A PostsController
A PostsController	ret(A.m2)		X User/Browser
X User/Browser	X.m1		X User/Browser
X User/Browser	A.m3		A PostsController
A PostsController	P.m2	Hash : params	P Post
P Post	ret(P.m2)	@post	A PostsController
A PostsController	P.m3		P Post
P Post	ret(P.m3)	boolean	A PostsController
A PostsController	**X.m2**		X User/Browser

Table 3.8: Blog - UC1, Path1: Instance-Trace Table.

Calling Object	Message	Parameters	Called Object
X User/Browser	A.m2		A PostsController
A PostsController	P.m1		P Post
P Post	ret(P.m1)	@post	A PostsController
A PostsController	ret(A.m2)		X User/Browser
X User/Browser	X.m1		X User/Browser
X User/Browser	A.m3		A PostsController
A PostsController	P.m2	Hash : params	P Post
P Post	ret(P.m2)	@post	A PostsController
A PostsController	P.m3		P Post
P Post	ret(P.m3)	boolean	A PostsController
A PostsController	**X.m3**		X User/Browser

Table 3.9: Blog - UC1, Path2: Instance-Trace Table.

Calling Object	Message	Parameters	Called Object
X User/Browser	A.m4		A PostsController
A PostsController	P.m4	Hash : params	P Post
P Post	ret(P.m4)	@post	A PostsController
A PostsController	ret(A.m4)		X User/Browser
X User/Browser	X.m4		X User/Browser
X User/Browser	A.m5		A PostsController
A PostsController	P.m4	Hash : params	P Post
P Post	ret(P.m4)	@post	A PostsController
A PostsController	P.m5	Hash : params	P Post
P Post	ret(P.m5)	boolean	A PostsController
A PostsController	**X.m2**		X User/Browser

Table 3.10: Blog - UC2, Path1: Instance-Trace Table.

Calling Object	Message	Parameters	Called Object
X User/Browser	A.m4		A PostsController
A PostsController	P.m4	Hash : params	P Post
P Post	ret(P.m4)	@post	A PostsController
A PostsController	ret(A.m4)		X User/Browser
X User/Browser	X.m4		X User/Browser
X User/Browser	A.m5		A PostsController
A PostsController	P.m4	Hash : params	P Post
P Post	ret(P.m4)	@post	A PostsController
A PostsController	P.m5	Hash : params	P Post
P Post	ret(P.m5)	boolean	A PostsController
A PostsController	**X.m5**		X User/Browser

Table 3.11: Blog - UC2, Path2: Instance-Trace Table.

Calling Object	Message	Parameters	Called Object
X User/Browser	B.m2		B CommentsController
B CommentsController	C.m1		C Comment
C Comment	ret(C.m1)	@comment	B CommentsController
B CommentsController	ret(B.m2)		X User/Browser
X User/Browser	X.m1		X User/Browser
X User/Browser	B.m3		B CommentsController
B CommentsController	C.m2	Hash : params	C Comment
C Comment	ret(C.m2)	@comment	B CommentsController
B CommentsController	C.m3		Comment
C Comment	ret(C.m3)	boolean	B CommentsController
B CommentsController	**X.m2**		X User/Browser

Table 3.12: Blog - UC3, Path1: Instance-Trace Table.

Calling Object	Message	Parameters	Called Object
X User/Browser	B.m2		B CommentsController
B CommentsController	C.m1		C Comment
C Comment	ret(C.m1)	@comment	B CommentsController
B CommentsController	ret(B.m2)		X User/Browser
X User/Browser	X.m1		X User/Browser
X User/Browser	B.m3		B CommentsController
B CommentsController	C.m2	Hash : params	C Comment
C Comment	ret(C.m2)	@comment	B CommentsController
B CommentsController	C.m3		Comment
C Comment	ret(C.m3)	boolean	B CommentsController
B CommentsController	**X.m3**		X User/Browser

Table 3.13: Blog - UC3, Path2: Instance-Trace Table.

- **Fault #1**: The method signatures m4 and m5 for all models did not match between the class and state diagrams. The arguments were void in the state diagram when they should have been a hash of attributes, as defined in the class diagrams.

2. Association violation

    - Detection: For every edge created in the DIG, make sure that method calls between two different classes has an association relationship in the class diagram.
    - No association violations were detected for our case study. This was most likely due to the well defined domain of our canonical blog project. Associations between users, posts, and comments are straightforward and easily translated into UML.

3. Navigation violation

    - Detection: Method calls between two different classes can only be performed one way if the association relationship is directed.
    - Directed associations are present between controllers and models, but we did not encounter any association violations during the entire process. Once again, this is most likely due to the limited nature of associations between our classes and the simplicity of the domain.

4. Invalid calling state or called state

    - Detection: While embedding the state diagrams into the system behavioral graph:

1) an object in a particular state is calling a method from another class while the state diagram does not allow the call in that state, or

2) a method from an object in a particular state is being called while the state diagram of the object does not allow the method to be called while the object is in that state.

- **Fault #2**: The state diagrams for all models only defined method m4 to be called while in state S1 (new record), when in fact the method should be able to be called while in state S1 and state S2 (existing record).

5. Missing behavior for a call behavior in activity/sequence diagram

   - Detection: During DIG construction, a call behavior appears in one diagram while it has not been defined in another diagram.
   - **Fault #3**: In the class diagram, all models had the `+new(attributes : Hash)` method defined, but this method was not present in the state diagrams.
   - **Fault #4**: The User/Browser entity in our sequence diagram made use of m5 method (redirect to /posts/:id/edit URL), but this method was not defined in any other diagrams.

Dynamic Faults

1. Multiplicity violations

   - Detection: After the instance/trace table construction, whenever an object is created or destroyed, for every relationship that the object's class type has in the class diagram, one needs to check the total number of the objects of that class per each relationship with objects of other classes.

- The multiplicities of our objects within the DIG were all within the specified bounds in our class diagram and hybrid instance-trace tables.

2. OCL constraint violations

   - Detection: Again, after instance/trace table construction, check the OCL constraints for each object, method, or attribute before and after each edge traversal in DIG.
   - As previously mentioned, we seeded our case study with three constraint violations. All three of our seeded violations were detected by our approach. Below are the details for each:

     1) **Fault #5**: While attempting to create a new post with an invalid Post.user_id attribute, both paths for UC1 detected this violation. This violation occurs when the PostsController attempts to call the Post.m2 method with the invalid attribute.

     2) **Fault #6**: While attempting to edit an existing post with an empty Post.title attribute, both paths for UC2 detected this violation. This violation occurs when the PostsController attempts to call the Post.m5 method with an empty title attribute.

     3) **Fault #7**: While attempting to create a new comment for a Post that doesn't exist (invalid Comment.post_id), both paths for UC3 detected this violation. This violation occurs when the CommentsController attempts to call the Comment.m2 method with the invalid post_id attribute.

3. Untraversable or non-existent path

    - Detection: Dependent upon the test adequacy criterion which dictates our desired path coverage through the DIG.
    - For our criterion, we went with full conditional coverage. Meaning, we selected our unique paths so that each conditional statement was fully covered through the DIG. We did not discover any invalid paths during this process.

3.1.11 Other Issues

While working our way through our process, a few other issues arose which were not related to the fault model but worthy of noting nonetheless. Additional behavior was defined in a few of the diagrams that was unneeded for the scope of this project. The focus was to design the minimal functionality needed to address the use cases and therefore there is no need to keep the functionality around in the designs if it is never used.

Each controller defined an unused show method in both the class and state diagrams. Additionally, the Comment model defined an unused `update_attributes` method definition in the class and state diagrams. This method would have been used if one of our use cases covered the editing of comments.

Chapter 4

Conclusions and Future Works

4.1 Analysis

As demonstrated, the wide range of support for various UML diagrams in our approach make it both flexible and unique. The entire process is based upon mature technologies and techniques, such as UML and directed graph theory. By focusing on fault detection in the design phase, we are able to discover issues early in the development lifecycle and ensure they do not make into into the development phase. At the same time, our approach is somewhat limiting due to the fact that we can only find certain types of faults as listed at the conclusion of case study. The entire process is tedious and time consuming. Some of the tasks can be done in parallel, but the most time consuming tasks cannot. The diagrams become huge and cumbersome rather quickly, and the complexity of the process increases dramatically as the size of the project increases. However, the process can be automated.

The majority of the downsides of our approach mentioned can be addressed through automation of the process. Our approach has been well defined and presented in detail, and can therefore be converted into an automated application in a fairly straightforward manner.

One automated approach would be to construct the UML diagrams by hand in an application similar to Visual Paradigm UML Modeler, and then export the diagrams into a parsable format such as XML. Once the diagrams are in their XML representation, an application could parse the XML, build all relevant directed graphs and tables based on this data, and finally present a report detailing the issues and faults discovered.

This sort of automation would eliminate all the manual directed graph construction, as well as path generation from the DIG, and the manual building of the hybrid instance-trace table. The process would no longer be time consuming and complexity of the DIG construction would not be an issue.

Overall, our approached has improved the current state of testing design documents by providing a flexible and effective solution based on mature technologies such as UML and graph theory. This dissertation has laid a strong design testing foundation to build upon with future works.

Bibliography

[1] A. Andrews, R. France, S. Ghosh, and G. Craig. Test adequacy criteria for UML design models. *Journal of Software Testing, Verification, and Reliability*, 13(2):95–127, October 2000.

[2] R. Binder. *Testing Object-Oriented systems models*. Addison-Wesley, 1999.

[3] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide- 2nd Edition*. Addison-Wesley, 2005.

[4] L.C. Briand and Y. Labiche. A UML-based approach to system testing. In *UML '01: Proceedings of the 4'th International Conference on UML*, pages 194–208, London, UK, 2001. Springer-Verlag.

[5] J. Scott Johnson S. Powers B. Trott M. Trott C. Doctorow, R. Dornfest. *Essential Blogging: Selecting and Using Weblog Tools*. O'Reilly Media, 2002.

[6] J. Ebert. A sensetive transitive closure algorithm. *Information Processing Letters*, 12:255–258, 1981.

[7] J. Eve and R. Kurki-Suonio. On computing the transitive closure of a relation. *Acta Informatica*, 8:303–314, 1977.

[8] H. Rohnert P. Sommerlad M. Stal F. Buschmann, R. Meunier. *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*. Wiley, 1996.

[9] M. Gogolla, J. Bohling, and M. Richters. Validation of UML and OCL models by automatic snapshot generation. In *UML '03: Proceedings of the 6'th International Conference on UML*, pages 265–279, 2003.

[10] Y.E. Ioannidis and R. Ramakrishnan. Efficient transitive closure algorithms. In *Proceedings of the 14'th VLDB Conference*, pages 335–346, Los Angeles, California, 1988.

[11] Y.E. Ioannidis, R. Ramakrishnan, and L. Winger. Transitive closure algorithms based on graph traversal. *ACM Transactions on Database Systems*, 18(3):512–576, 1993.

[12] S.J. Mellor and M.J. Balcer. *Executable UML: A Foundation for Model Driven Architecture*. Addison-Wesley Professional, 2002.

[13] I. Munro. Efficient determination of the transitive closure of a directed graph. *Information Processing Letters*, 1(2):56–58, 1971.

[14] O. Pilskalns, A. Andrews, A. Knight, S. Ghosh, and R. France. UML design testing. *Journal of Information Science and Technology*, 19(8):192–212, August 2007.

[15] O. Pilskalns, A.A. Andrews, S. Ghosh, and R.B. France. Rigorous testing by merging structural and behavioral UML representations. In *UML '03: Proceedings of the 6'th International Conference on UML*, pages 234–248, 2003.

[16] P. Purdom. A transitive closure algorithm. *BIT*, 10:76–94, 1970.

[17] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 2005.

[18] L. Schmitz. An improved transitive closure algorithm. *Computing*, 30:359–371, 1983.

[19] T.D. Trong, N. Kawane, S. Ghosh, R. France, and A. Andrews. A tool-supported approach to testing UML design models. In *ICECCS '05: Proceedings of the 10'th IEEE International Conference on Engineering of Complex Computer Systems*, pages 519–528, 2005.