

DISSERTATION

A SYSTEMATIC APPROACH TO TESTING UML DESIGNS

Submitted by

Trung T. Dinh-Trong

Department of Computer Science

In partial fulfillment of the requirements

for the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Spring 2007

UMI Number: 3266387

INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

UMI[®]

UMI Microform 3266387

Copyright 2007 by ProQuest Information and Learning Company.

All rights reserved. This microform edition is protected against unauthorized copying under Title 17, United States Code.

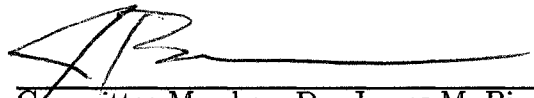
ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346


COLORADO STATE UNIVERSITY

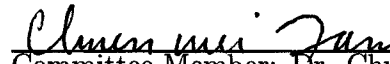
August 14, 2006

WE HEREBY RECOMMEND THAT THE DISSERTATION PREPARED UNDER OUR SUPERVISION BY TRUNG T. DINH-TRONG ENTITLED A SYSTEMATIC APPROACH TO TESTING UML DESIGNS BE ACCEPTED AS FULFILLING IN PART REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY.

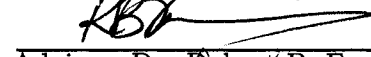
Committee on Graduate Work

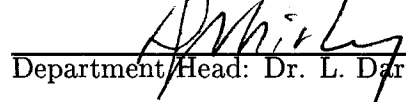

Committee Member: Dr. James M. Bieman


Committee Member: Dr. Yashwant K. Malaiya


Committee Member: Dr. Chuen-mei Fan


Co-Adviser: Dr. Sudipto Ghosh


Adviser: Dr. Robert B. France


Department Head: Dr. L. Darrell Whitley

ABSTRACT OF DISSERTATION

A SYSTEMATIC APPROACH TO TESTING UML DESIGNS

In Model Driven Engineering (MDE) approaches, developers create and refine design models from which substantial portions of implementations are generated. During refinement, undetected faults in an abstract model can traverse into the refined models, and eventually into code. Hence, finding and removing faults in design models is essential for MDE approaches to succeed.

This dissertation describes a testing approach to finding faults in design models created using the Unified Modeling Language (UML). Executable forms of UML design models are exercised using generated test inputs that provide coverage with respect to UML-based coverage criteria. The UML designs that are tested consist of class diagrams, sequence diagrams and activity diagrams.

The contribution of the dissertation includes (1) a test input generation technique, (2) an approach to execute design models describing sequential behavior with test inputs in order to detect faults, and (3) a set of pilot studies that are carried out to explore the fault detection capability of our testing approach.

The test input generation technique involves analyzing design models under test to produce test inputs that satisfy UML sequence diagram coverage criteria. We defined a directed graph structure, named Variable Assignment Graph (VAG), to generate test inputs. The VAG combines information from class and sequence diagrams. Paths

are selected from the VAG and constraints are identified to traverse the paths. The constraints are then solved with a constraint solver.

The model execution technique involves transforming each design under test into an executable form, which is exercised with the generated inputs. Failures are reported if the observed behavior differs from the expected behavior. We proposed an action language, named Java-like Action Language (JAL), that supports the UML action semantics. We developed a prototype tool, named *UMLAnT*, that performs test execution and animation of design models.

We performed pilot studies to evaluate the fault detection effectiveness of our approach. Mutation faults and commonly occurring faults in UML models created by students in our software engineering courses were seeded in three design models. Ninety percent of the seeded faults were detected using our approach.

Trung T. Dinh-Trong
Department of Computer Science
Colorado State University
Fort Collins, Colorado 80523
Spring 2007

ACKNOWLEDGEMENTS

I owe a special note of gratitude to my advisor, Dr. Robert France, and my co-advisor, Dr. Sudipto Ghosh, for their invaluable guidance and friendship. They patiently supported and encouraged me during every moment, good or bad. My appreciation also goes to my advisory committee members, Dr. James Bieman, Dr. Yashwant Malaiya, and Dr. Chuen-mei Fan, and my former advisory committee member, Dr. Daniel Turk, for taking the time to give me valuable feedback. I thank the entire Computer Science Department staff, especially Carol Calliham, Sharon Van Gorder, and our former accountant, Susan Short, for always smiling when they helped me with paperwork.

I extend many thanks to my friends and fellow graduate students. I appreciate Nilesh Kawane's problem solving skills and thank him for his sense of humor when we worked together on the prototype tool implementation. I appreciate the time and effort that Eunjee Song, Devon Simmonds, and Raghu Reddy spent helping me with my pilot studies by seeding faults into many design models.

Thanks are also due to Thanh Nguyen, Hong Pham, Son Nguyen, and Hang Nguyen, who helped me apply for admission to Colorado State University and cared for me when I came to the US. They are like family to me.

I thank my dearest wife, Linh, for her love, patience, companionship, and understanding. I thank my son, Toon, for being my source of joy. I am grateful to my sister, Trang, who always loves me and looks after me. I thank my parents, who

always gave priority to their children's education and constantly encouraged us to be inquisitive.

Finally, I would like to thank my sponsors for their support. The research was supported in part by the National Science Foundation and two IBM Eclipse Innovation Grants.

This dissertation is dedicated to my parents. My father was the first to show me a computer and explain to me the meaning of “information technology”. My mother sacrificed her career for her children’s well-being.

TABLE OF CONTENTS

1	Introduction	1
1.1	Problem	1
1.2	Overview of the solution and the research challenges	2
1.2.1	Scope of the research	4
1.3	Dissertation organization	5
2	Background	6
2.1	Software testing	6
2.2	The Unified Modeling Language	7
2.2.1	Class Diagram	9
2.2.2	Sequence diagram	10
2.2.3	Activity diagrams	11
3	Related Work	12
3.1	Program test input generation	12
3.1.1	UML-based test input generation	12
3.1.2	Path-oriented test case generation	14
3.2	Testing UML designs	16
3.3	Model execution	17
4	Approach	20
4.1	Generating Test Inputs	21
4.2	Generating the Executable Form	22
4.3	Generating the Testable Form	22

4.4	Executing Tests and Detecting Failures	23
4.5	Animating the execution	24
5	Generating Design Test Inputs	25
5.1	Generating the Variable Assignment Graph	29
5.2	Selecting Complete Paths	36
5.3	Generating Path Constraints	38
5.4	Solving Constraints	41
6	Java-Like Action Language	45
6.1	An overview of JAL statements	46
6.2	JAL control statements	47
6.3	JAL single statement and expression	48
6.4	Compound statements	51
7	Test Execution	53
7.1	Generating the <i>EDUT</i>	53
7.1.1	Transforming class diagrams into EDUT code	54
7.1.2	Generating the <i>TFactory</i> class	58
7.1.3	Generating <i>EDUT</i> method bodies from JAL specifications	58
7.2	Generating the <i>TDUT</i>	61
7.2.1	Generating code to check the initialization of variables	63
7.2.2	Generating code to check for existence of target objects	64
7.2.3	Generating code to validate pre- and post-conditions	64
7.2.4	Generating code for automation of test execution	67
7.3	Executing Tests	69
8	Tool Support	71
8.1	Model specification	72

8.2	Generation of the testable form	73
8.3	Test execution and failure reporting	77
8.4	Model animation	77
8.5	Testing <i>UMLAnT</i>	79
9	Pilot Studies	81
9.1	Test input generation	83
9.2	Test execution results	86
9.3	Discussion	88
10	Conclusions and Future Work	89
10.1	Summary of the contribution	89
10.2	Discussion	90
10.3	Future work	93
A	Java like Action Language Specification, Version 1.1	94
A.1	Introduction	94
A.2	Grammar	94
A.3	JAL syntax	96
A.3.1	Identifiers	96
A.3.2	Keywords	96
A.3.3	Primitive and Pre-defined Types	97
A.3.4	Condition statements	97
A.3.5	Loop statements	97
A.3.6	Atomic actions	98
A.3.6.1	Create object expression	98
A.3.6.2	Destroy object statement	99
A.3.6.3	Read link expressions	99
A.3.6.4	Create link statement	100

A.3.6.5	Delete link statement	101
A.3.6.6	Call operation expression	102
A.3.6.7	Return statement	103
A.3.6.8	Read attribute expression	103
A.3.6.9	Write attribute statement	104
A.3.6.10	Calculation expression	105
A.3.6.11	Accessing variables	106
A.3.7	Compound statement	106
B	UMLAnT User Guide	107
B.1	Creating a <i>DUT</i>	108
B.2	Generating <i>TDUT</i>	110
B.3	Writing test cases	111
B.4	Launching the test runner	112
B.5	Running test cases	113
B.6	Animating the execution	114
	References	115

LIST OF TABLES

5.1	Variables Defining the Start Configuration for the Path 0-1-2-3-5-10. . .	39
7.1	Rules to transform JAL creation expressions and destruction statements into Java	60
9.1	Sizes of systems under test.	81
9.2	Number of test cases generated from the models to satisfy the criteria. .	84
9.3	Fault detection data.	87

LIST OF FIGURES

4.1	Test Approach.	20
5.1	Test Input Generation Process.	26
5.2	UML Design Model for Product-Catalog Management.	28
5.3	A VAG Example.	29
5.4	Transformation Rule for Sequence Diagram Loop Structures.	36
5.5	The Constraints for the Path 0-1-2-3-5-10.	40
5.6	A Part of the Constraint for the Path 0-1-2-3-5-10 in Alloy Language. . .	42
5.7	Start Configuration for the Path 0-1-2-3-5-10.	44
6.1	The Product-Catalog System: addProduct JAL Specification	46
6.2	ReplyAction in UML 2.0	48
6.3	Example of Combination of Atomic Actions in UML 2.0	52
7.1	A template DUT class diagram	54
7.2	The <i>EDUT</i> TObject class generated from a class diagram.	55
7.3	The EDUT C1 class.	56
7.4	The EDUT C1Collection class.	57
7.5	A TFactory class.	59
7.6	EDUT generated from addProduct JAL Specification	61
7.7	Test Execution Packages.	62
7.8	The TDUT code generated from an attribute.	64
7.9	The TFactory class with code to interact with USE.	66

7.10	The <code>_set_A()</code> method that has code inserted to interact with USE.	67
7.11	EDUT generated from <code>addProduct</code> JAL Specification	68
7.12	Sample Test Case.	69
8.1	UMLAnT Architecture.	71
8.2	The Design Class Diagram of the <i>Model Management</i> Sub-System. . . .	72
8.3	UMLAnT Input Screen.	73
8.4	The Design Class Diagram of the <i>EDUT/TDUT Generator</i> Sub-System. . .	74
8.5	<i>UMLAnT</i> Classes that Play the Roles of <i>Elements</i> in the <i>Visitor</i> pattern. .	75
8.6	Sequence diagram for transforming a class into TDUT.	76
8.7	UMLAnT Animation Screen.	78
9.1	Relationship between Path Length and Constraint Size.	85
10.1	A modeling process that includes model testing.	93
A.1	Create Object Action Meta-Class Diagram [56].	98
A.2	Destroy Object Action Meta-Class Diagram [56].	99
A.3	Read Link Action Meta-Class Diagram [56].	99
A.4	Create Link Action Meta-Class Diagram [56].	100
A.5	Destroy Link Action Meta-Class Diagram [56].	101
A.6	Call Operation Action Meta-Class Diagram [56].	102
A.7	Reply Action.	103
A.8	Read Structural Feature Action Meta-Class Diagram [56].	104
A.9	Write Structural Feature Actions Meta-Class Diagram [56].	104
A.10	Value Specification Actions Meta-Class Diagram [56].	105
B.1	<i>DUT</i> Class Diagram of the Product Management System.	107
B.2	OCL Constraint for the “Demo” project.	109
B.3	JAL segment for <code>ProductCatalog::addCategory()</code>	109

B.4	JAL segment for <code>Category::setID()</code>	110
B.5	JAL segment for <code>Category::getID()</code>	110
B.6	JAL segment for <code>ProductCatalog::addCategory()</code>	110
B.7	JAL segment for <code>ProductCatalog::addProduct()</code>	110
B.8	JAL segment for <code>Product::setID()</code>	110
B.9	JAL segment for <code>ProductCatalog::findCategory()</code>	111
B.10	The Code for the “testOne” method.	112

Chapter 1

Introduction

1.1 Problem

Model Driven Engineering (MDE) approaches tackle the complexity of developing large software systems by raising the level of abstraction at which developers build software. In MDE approaches, developers focus on creating and evolving design models. Abstract logical models (e.g., Platform Independent Models) are systematically transformed to detailed design models (e.g., Platform Specific Models) [54]. Eventually, substantial portions of implementations are automatically generated from the models. If a design model contains faults that are not removed before transformation, those faults are passed to the generated code where they can be more expensive to remove. For MDE approaches to succeed, practical techniques for validating design models are needed.

MDE approaches require that models be precisely described using a modeling language. The Unified Modeling Language (UML) [56] is an OMG standard language for modeling object-oriented systems. Software developers can use the UML to describe designs at different levels of abstraction, from conceptual to detailed design [8]. UML design models consist of a variety of diagrams. Each diagram describes a view of the design. For example, a class diagram describes a structural view and sequence and activity diagrams describe behavioral views.

UML designs are typically evaluated using walkthroughs, inspections, and other types of design review techniques that are largely manual. Reviewers need to manually track and relate a large number of concepts across various diagrams. These manual tasks can quickly become tedious when the designs are complex, which is the case in many MDE projects.

1.2 Overview of the solution and the research challenges

This dissertation describes an alternative, systematic and automatable approach to validating design models. Executable forms of UML design models consisting of class, interaction, and activity models are exercised with test inputs. During test execution, the states of the system under test and the communication between objects are visualized using UML sequence and object diagrams.

This dissertation describes an approach to derive test inputs from design models, an approach to execute the tests on the design models, and an approach to animate the execution. The test input generation technique aims at deriving test inputs that satisfy a set of predefined test adequacy criteria. The test input generation requires both structural and behavioral information that is modeled in the designs under test. This information is scattered across different views of UML design models. For example, class models specify structural aspects of the modeled system, sequence models specify the interactions between objects, and operation pre- and post-conditions capture the effects of the operations. Hence, the test input generation requires analyzing different views of the design models. To ease the analyzing process, we developed a mechanism that integrates information from different views of a model into one view.

Executing a UML design model and animating the execution require that system behaviors be specified formally. In UML models, behaviors are specified using sequence diagrams, statecharts and activity diagrams. A sequence diagram only cap-

tures the interactions between objects. What happens inside an object (e.g., the modification of attribute values) is not described in a sequence diagrams. Statecharts and activity diagrams can provide more complete descriptions of behavior. However, both statechart and activity diagram views require a language to specify the actions that take place in states and activities. The UML 2.0 standard [56] includes the action semantics for this purpose but does not provide a surface notation for action languages. This dissertation provides an action language, called Java-like Action Language (JAL), which is based on the UML action semantics.

Test results are externally visible outputs generated during testing. Expected results, determined by oracles, are externally visible outputs that are generated by a correct system. If test results differ from expected results, then the test has detected a failure. The approach can also detect failures caused by inconsistencies across the UML diagrams in the model. For example, the behavior described by an activity diagram may produce a configuration that violates the constraints given in the class diagram. To detect such a failure, test execution is observed in terms of the actions performed by the system under test and the sequence of states that the execution passes through.

We also present a technique to provide support for visualizing the behavior of models during testing. The execution of a model is visualized using two types of views — object and sequence diagrams. The views get updated when developers step through the execution of operations. The animated object diagrams show the creation and deletion of objects and links, as well as the modification of attribute values. The animated sequence diagrams show the messages exchanged between objects during execution. Novice modelers and students learning the UML can use the animation technique to get visual feedback that can be used to help them identify problems with the modeled behaviors. Software developers in industry can use the approach for rigorously testing their models before the models are transformed into

code. Animation can also be used to help a developer understand design models that are created by other developers.

We implemented the test execution and visualization technique in a prototype tool called *UMLAnT*. It is an Eclipse plugin and works in conjunction with the Eclipse Modeling Framework (EMF).

We conducted pilot studies to investigate the effectiveness of our testing approach. The studies used a set of fault types obtained from an analysis of design models developed by students in senior and graduate level software engineering courses at Colorado State University, and from mutation analysis of UML designs [14]. The studies demonstrate the cost of testing in terms of the number of test inputs and the size of each input, and identify the fault types that are likely to be found using this testing technique.

1.2.1 Scope of the research

The work described in this dissertation focuses on validating UML design models. Testing code implementations is outside the scope of the dissertation.

The testing technique is used to detect semantic faults in well-formed UML design models. Well-formedness checks can be performed by existing UML drawing tools (e.g. Together [9] and Rational Rose [28]), and hence, is outside the scope of the dissertation.

In our approach, a UML design model under test consists of class diagrams in which each operation is associated with an activity diagram describing the operation's behavior, and sequence diagrams that describe the scenarios to be tested. Specifications for the operations that is called in the sequence diagrams must also be available to the testers. The specifications consist of pre- and post-conditions expressed in the OCL. These specifications are used to produce test inputs and to determine whether

executed operations satisfy their specifications. OCL is also used to describe class invariants.

The design models under test are assumed to describe sequential behaviors only. This assumption guarantees that the state of the system under test is always known when an execution of an atomic action is completed.

1.3 Dissertation organization

The rest of the dissertation is organized as follows:

- Chapter 2 presents background on software testing and UML, and defines the terminology that is used in the dissertation.
- Chapter 3 discusses related work in software testing.
- Chapter 4 provides an overview of our approach.
- Chapter 5 describes our technique for generating test inputs from UML design models.
- Chapter 6 describes JAL.
- Chapter 7 explains our technique for executing design models and detecting test failures.
- Chapter 8 presents the design of *UMLAnT*.
- Chapter 9 discusses the results of our pilot studies.
- Chapter 10 concludes the dissertation and outlines directions for future work.
- Appendix A describes the JAL grammar and syntax.
- Appendix B is a user's guide to *UMLAnT*.

Chapter 2

Background

The following sections describe the concepts and principles that are relevant to the dissertation. Section 2.1 discusses software testing terminology. Section 2.2 summarizes the description of UML diagrams that are used in our approach.

2.1 Software testing

According to Myers [45], testing is the process of executing a program with the intent of finding faults. In the testing approach described in this dissertation, testing is done by executing the design models. Adrison et al. [2] define a program under test as any object that can be executed. Using this definition, executable design models can be viewed as programs. We use the same program testing terminology, such as *test adequacy criteria*, *test inputs*, *test oracles*, *test cases*, *test drivers*, *test failures*, and *faults* in this dissertation.

A program under test is usually viewed as a representation of a function that maps input elements to output elements [2]. During testing, testers select input elements, determine the expected results, execute the program with the selected inputs, observe the actual results, and finally compare the actual results with the observed results. The actual results are produced by the program when test inputs are applied to it. The expected results specify the outputs that the program under test should produce

from test inputs. A result is an observable behavior of the program during execution. An oracle is a mechanism to produce expected results.

An automated oracle can make a pass/ fail evaluation. If the expected and actual results agree after the execution of each test input, the test is said to pass; otherwise it is a failure [6]. A failure indicates that there is a fault in the program. A fault is a missing or an incorrect component of the program. A single fault can result in various failures, and the same failure can be caused by different faults.

It is commonly known that testing a program with all possible inputs (exhaustive testing) is infeasible. During testing, only a subset, named test input set, of the set of all input elements, is chosen to execute the program. The test input set must be “large enough to span the domain” [2], yet small enough that the testing process can be completed within an acceptable timeframe. The selection of a test input set is usually guided by test adequacy criteria. Weyuker [61] defines a test adequacy criterion as a rule to determine when testing may terminate. For example, the *All statement coverage* criterion states that testing can terminate when all program statements are executed at least once during testing. A test adequacy criterion is used as a guideline to select a finite set of test inputs. During testing, new test inputs are generated until the selected test adequacy criterion is satisfied.

If a program under test is part of a larger system, executing the program during testing usually requires auxiliary software, such as drivers and stubs [2]. A driver sets up an appropriate environment and invokes functions of the program using the selected input sets. In our approach, test drivers are used to bring a system into a particular state before testing begins, and to evaluate the test results.

2.2 The Unified Modeling Language

UML 2.0 is specified using two complementary specifications: infrastructure [55] and superstructure [56]. The infrastructure defines the foundation language constructs

required for UML 2.0. The superstructure defines the user level constructs. As in the previous versions, UML 2.0 is defined by (1) a metamodel consisting of an abstract syntax, (2) a set of well-formedness rules and (3) an informally described semantics. The abstract syntax is defined by UML class diagrams supported by a natural language description. The well-formedness rules are expressed in the Object Constraint Language (OCL), a language for expressing side-effect free constraints [57]. The semantics are informally described using natural language.

UML defines various graphical diagrams including use cases, class diagrams, statecharts, activity diagrams, sequence diagrams, component diagrams, and deployment diagrams. To be tested, a model needs to be executable. An executable UML model needs at least a class model that describes the structural aspects of the system under test, and another model, such as a state model or an activity model, that describes the behavioral aspects of the system. Researchers have developed several state-based testing approaches [12, 48]. Statecharts are useful modeling tools in certain domains, such as embedded software development, but may not be suitable for use in other domains [37]. For example, it can be more convenient to model a library check-in/check-out system using class, sequence, and activity models instead of statecharts. Hence, there is a need for an approach to testing design models that contain diagrams other than statecharts.

We require that the models under test contain class diagrams that specify the structural aspects of the system under test. OCL is used to describe class invariants and operation pre- and post-conditions. Our approach aims at testing different scenarios, several of which may be specified by one sequence diagram. Hence in our approach, the models under test must contain sequence diagrams for the scenarios that will be tested. Sequence diagrams only capture the interactions between objects, and cannot be used to specify what happens inside each object (e.g., the modification

of attribute values). Our approach requires that the behavior of every operation in the class model be specified using an activity model.

2.2.1 Class Diagram

A class diagram captures information about classes and interfaces using attributes and operations, as well as the relationships between classes using associations and generalizations. A class characterizes a set of objects that share the same set of features, constraints, and semantics. An instance of a class is called an object. The class features include attributes and operations. The attributes of a class specify its data structure, and are represented by instances of properties that are owned by the class. Some of the attributes may represent the navigable ends of binary associations. The operations represent the common behavior of the class. A class diagram characterizes the set of valid object configurations.

A property has a name, which is unique among property names in the same class, and a type. A property relates an instance of the class to a value or a collection of values of the type of the property. A property can also have an optional initial value. Whenever an instance of a class is created, the properties are assigned with the corresponding initial values. A property can have a multiplicity constraint, specifying the bounds of the cardinality of the collection of values that can be associated with the property.

An operation specifies a method that every instance of the class can be requested to execute. It has a name, a return type, and an optional list of arguments. Each argument has a name and a type.

A binary association represents a relationship between two classifiers. It models how peer instances of classifiers relate to each other. Each end of an association connects to a classifier via a property. An end property of an association can belong to the set of owned attributes of the end class, indicating that the association is

navigable from the opposite ends. Otherwise, the association is not navigable from the opposite ends.

A generalization structure defines the relationship between a more general super-class and a more specific sub-class that is fully consistent with the super-class but has more properties. A sub-class defines a subset of the instances of the super-class. Any instance of the sub-class is also an instance of the super-class.

OCL can be used in the class diagram to specify class invariants, as well as pre- and post- conditions of operations. A pre-condition is a constraint on the state of the modeled system when the operation is invoked. A post-condition is the constraint on the state of the system when the operation is complete.

2.2.2 Sequence diagram

A sequence diagram models system behavior by specifying how objects interact to complete a particular task. An interaction is expressed by messages between lifelines. A lifeline is a participant in an interaction. A lifeline represents a class instance.

In this dissertation, a message can represent a method invocation, a reply message, and creation or deletion of a class instance. When a message representing a method invocation is sent, the corresponding operation is executed. When the execution of the invoked operation is complete, a reply message is sent from the called lifeline to the calling lifeline. Upon receiving the reply message, the calling lifeline will proceed.

A set of messages can be grouped into a **CombineFragment**. This dissertation restricts a **CombineFragment** to be either a **Loop** or an **Alternatives** fragment. A loop fragment specifies an iteration of messages. An alternatives fragment represents conditional interactions.

In our approach, each sequence diagram models the interaction between objects when a system operation is executed. A system operation is an operation of the

system that executes in response to an external input event generated by an actor to the system [40].

2.2.3 Activity diagrams

In this dissertation, activity diagrams are used to describe behavior needed to implement operation specifications. A formal language is needed to specify the actions that take place in activities described by activity diagrams. The UML 2.0 standard [56] includes the action semantics for this purpose, but does not provide a standard surface notation for action languages. This dissertation includes the description of a Java-like Action Language in Chapter 7.

An activity diagram models behavior by specifying the sequence of actions and the conditions for coordinating actions. An action is a fundamental unit of behavior specification. It takes a set of inputs and converts them into a set of outputs (both sets can be empty). Some actions also modify the state of the system in which the action executes. The following types of actions are included in the activity diagrams used in our approach: call operation actions, calculation actions, create and destroy object actions, create and destroy link actions, read and write link actions, and read and write variable actions.

Chapter 3

Related Work

In this chapter, we discuss related work in software testing. We summarize existing work on generating inputs for testing programs in Section 3.1. Work on validating design models is discussed in Section 3.2. Testing UML designs requires a mechanism to execute the design models. Section 3.3 discusses the status of various UML execution techniques.

3.1 Program test input generation

Test inputs for programs can be derived either from the specifications (black-box testing) or from the structure of the programs (white-box testing). Existing black-box testing techniques that generate test inputs from UML models are discussed in Section 3.1.1. Path-oriented test case generation, a widely used white box-testing technique, is reviewed in Section 3.1.2.

3.1.1 UML-based test input generation

Offutt and Abdurazik [48] describe how to generate test inputs from a restricted form of UML state-charts and defined four levels of test coverage: transition coverage, full predicate coverage, transition-pair coverage and complete sequence. These coverage levels require test sets to cover every transition, every clause of transition predicates, every pair of transitions and a complete sequence of transitions, respectively. The

authors also provide algorithms to generate test input sets that satisfy these coverage criteria. A limitation of the approach is that it supports only simple states, enable transitions and change events. Briand et al. [12] enhance the test generation approach to support call and signal events, as well as five types of actions: call, send, assignment, create, and destroy. In this approach, the transition guards, and pre- and post-conditions are expressed using the OCL. These OCL expressions are normalized and then analyzed to provide guidance to generate test inputs. Kim et al. [34] describe an approach to generate tests from UML state-charts that contain composite states. UML state-charts are transformed into extended finite state machines (EFSM). Two sets of coverage criteria were defined based on control flow and data flow on the generated EFSMs.

Continuing their work on generating tests from design specifications, Abdurazik and Offutt [1] describe a set of test requirements based on collaboration diagrams for both static and dynamic evaluation. Model artifacts that must be evaluated during static checking are classifier roles, collaborating pairs, messages and local variable definition-usage link pairs. These artifacts are described using the definition of link types where objects are created, defined, used, and destroyed. Abdurazik and Offutt also define a test criterion which requires that all messages in collaboration diagrams must be sent at least once. However, the authors do not discuss how to generate test inputs that satisfy the criterion.

Scheetz et al. [52] describe an approach to generate system test inputs from UML class diagrams. The class diagrams are restricted to contain only classes, associations and specification structures. Scheetz et al. first identify test objectives for every single class. A test objective describes a set of objects in terms of the states they can take on. Test objectives are derived from defining desired states of class instances after the test is executed. A state of an object is defined based on its attribute values and links to other objects. Test objectives for a complete class diagram can be aggregated from

the test objectives for each individual class specified in the class diagram. Finally, an AI planner is used to convert the test objectives into test input sets. The planner identifies a test input as a sequence of actions that bring the system from an initial state to a desired goal state.

Instantiating every possible number of instances of each class, and aggregating every possible test objective of every class in a class diagram may make the total number of class diagram level test objectives large. Scheetz et al. [52] did not describe how to select a desirable subset of test objectives for class diagrams.

Briand and Labiche [13] propose the TOTEM system test methodology. Test requirements are derived from UML analysis artifacts such as use cases, their corresponding sequence and collaboration diagrams, class diagrams and OCL expressions across these artifacts. Test cases, test oracles, and test drivers are then developed using these test requirements and more detailed design information. Currently, the authors focus on deriving test requirements from use cases and sequence diagrams only. The other tasks, such as generating test requirement from class diagrams and generating test cases, test oracle and test drivers are to be addressed in future work.

The approaches to generate test inputs from class and state models (see Briand et al. [12], Kim et al. [34], Offutt et al. [48], and Scheetz et al [52]) can be extended to complement our approach, which derives test inputs that satisfy interaction model based criteria. The approaches presented by Offutt et al. [1] and Briand et al. [13] describe test objectives that are derived from different design model artifacts. However, they do not describe how test inputs are generated from these objectives.

3.1.2 Path-oriented test case generation

Path-oriented test case generation involves selecting a set of execution paths and deriving program inputs that execute the paths. The execution paths are selected to satisfy a certain test adequacy criterion. Two types of methods have been proposed to

identify program inputs that cover a selected path: execution-oriented and symbolic execution based test data generation.

Execution-based test generation [17, 25, 38] involves analyzing the execution of programs with actual inputs and iteratively refining the input values until a desired path is traversed. The program is first executed with an arbitrary input, and the program execution flow is monitored. When an undesired branch is executed, function minimization search algorithms [17, 38] are used to find an input value so that the desired branch is traversed. An undesired branch is a branch that does not belong to the chosen path. Gupta et al. [25] use an iterative relaxation method to replace the minimization search algorithm. If the branch conditions on a path are non-linear functions, the relaxation method can find a desired input faster than the minimization algorithms.

Execution-based approaches require executing programs under test in order to find test inputs. These approaches cannot be applied to our work, which aims at deriving test inputs from the combination of class and interaction diagrams. Interaction models only specify the interaction between objects and do not have enough information to be executable. Class models specify behaviors declaratively using OCL pre- and post-conditions. To date, there is no approach to execute a model that only contains class and sequence diagrams.

Boyer et al. [10] proposed a technique to find inputs to cover a given path by executing that path using the symbolic execution technique [35, 36]. Programs are executed using symbolic values of variables instead of actual values. As a result, every branch predicate along the path is expressed in terms of the input symbols. Symbolic evaluation is used to generate a set of equalities and inequalities of the program input values, which must be satisfied for the path to be traversed. Several techniques, such as Benders [4] algorithm, Gomory [24] algorithm, Tsang's consistency algorithm [59], and Hentenryck's interval programming method [27] can be used to

solve the inequalities and find a desired input solution. Nguyen and Deville [58] extend the symbolic execution approach to deal with arrays and procedure calls.

Current symbolic execution based test generation approaches are only applicable for programs with primitive type variables. These techniques lack a mechanism to model objects and links, which are usually present in branch predicates of a UML design model. Hence, current symbolic execution approaches cannot be directly used to derive inputs in our approach. In our approach, we symbolically represent object configurations using sets and relations.

3.2 Testing UML designs

Andrews et al. [3] define two sets of UML design test adequacy criteria that are based on coverage of elements of class diagrams and collaboration diagrams. The first set is defined based on the structural aspects of the system, and consists of Association-End Multiplicity Criterion, Generalization Criterion and Class Attribute Criterion. These criteria were based on category-partition and boundary value analysis technique [49].

The second set of design test adequacy criteria is based on control flow, and is defined for collaboration diagrams [3]. A collaboration diagram based test adequacy criterion defines a set of collaboration diagram elements that need to be covered during testing:

- The Condition coverage (*Cond*) criterion: requires that every condition in the collaboration diagrams must be evaluated to both **TRUE** and **FALSE** at least once.
- The Each Message on Link (*EML*) criterion: requires that each message on a link connecting two objects in the collaboration diagram must be sent at least once.
- The All Message Paths (*AMP*) criterion: requires that every possible message path in the collaboration diagrams must be exercised. A message path is a sequence of messages that are sent when a collaboration diagram is executed.

Ghosh et al. [21] describe a testing approach that utilizes the above criteria to generate test inputs. The authors also demonstrate in a case study that test inputs tend to cover multiple coverage elements. The authors, however, do not provide a systematic approach to derive test inputs.

Pilskalns et al. [50] propose a graph-based approach to combine the information from structural and behavioral diagrams (class diagrams and sequence diagrams). In this approach, each sequence diagram is transformed into an Object-Method Directed Acyclic Graph (OMDAG). Each node in an OMDAG represents a method call or a return action, as well as the class of the object that initiates the call. The directed arcs represent control flow. OMDAGs can be used to generate the execution paths in the sequence diagram that satisfy different test adequacy criteria, such as *Cond*, *EML*, and *AMP*. The sequence of method calls corresponding to each test case is generated and recorded in a table called Object-Method Execution Table (OMET), which is used to track test execution. The authors propose a framework that generates test inputs and executes tests. However, the framework is described at a high level and the details need to be worked out before it can be validated.

Gogolla et al. [23] present an approach to validate UML class diagrams and OCL models using snapshots. A snapshot is an object diagram that represents system states at a certain time with objects, attribute values, and links. Test cases are used to demonstrate that snapshots can be constructed to obey constraints in the model. Also, invariants can be dynamically loaded and checked against the snapshots. Our approach utilizes the USE tool to (1) check if the runtime state of a system under test conforms to the specification, and (2) validate operation pre- and post-conditions.

3.3 Model execution

A number of UML design execution techniques exist. Riehle et al. [51] propose an architecture of a UML virtual machine. The UML virtual machine has UML as its

instruction set and the memory management facilities of an existing Java Virtual Machine as its memory model. The advantage of such an approach is that UML models can be directly executed without being transformed into any other format (such as a program in code generation approach). There are currently no publicly available virtual machines that cover the UML diagrams we are targeting in our work, and thus, we had to investigate other approaches to executing design models.

Mellor and Balcer [42] present an action language to make UML executable. They use domain-specific model compilers to execute the UML models. Their technique is part of a system development approach based on the Model Driven Architecture. There are also industrial tools that support model execution using action semantic languages. BridgePoint [43] uses BridgePoint Action Language (AL), Kabira [31] uses Kabira Action Semantics (AS), and iUML [33] uses Action Semantic Language (ASL). All of the above languages model software behavior using state machines. They, as well as our action language, JAL, are based on the same set of semantics, which is described in the UML 2.0 specification. However, our language has a syntax that is similar to Java. Thus, a developer who is familiar with Java will find it easy to learn JAL.

Another approach for executing UML designs is code generation, where a program (e.g., a Java program) is generated from a given UML design model. Assuming that the program represents exactly the information in the design model, executing the code is the same as executing the model. For execution purposes, both structural and behavioral aspects (modeled using interaction diagram, activity diagram or statechart) of the model need to be transformed into code. Harel and Gery [26] describe how to generate code from UML models that consist of class diagrams and state charts. Engels et al. [16] present a set of rules to transform UML class diagrams and collaboration diagrams into code. Extending this idea, Dinh-Trong [15] defines a set of rules to generate code from a UML models consisting of class, collaboration

and activity diagrams. Dinh-Trong [15] also extends the UML notation for the collaboration diagram, allowing one to model condition and iteration structures that are applied to a set of multiple messages.

The FUJABA tool [46] translates UML models into Java programs and vice versa. FUJABA represents UML models using class diagram notation and a new notation called the *Story Diagram*, which is a combination of the Statechart with the Collaboration Diagram. The Collaboration Diagram, however, is represented using a of non-UML notation, which is based on a rewrite-graph technique. There are also industrial tools that can generate code from both class diagrams and collaboration diagrams, such as Together [9].

To use any of the above approaches in our work would require extending the code generation mechanisms to support generation of the test infrastructure. We chose to extend the approach used by Dinh-Trong because we had access to code generation mechanisms.

Chapter 4

Approach

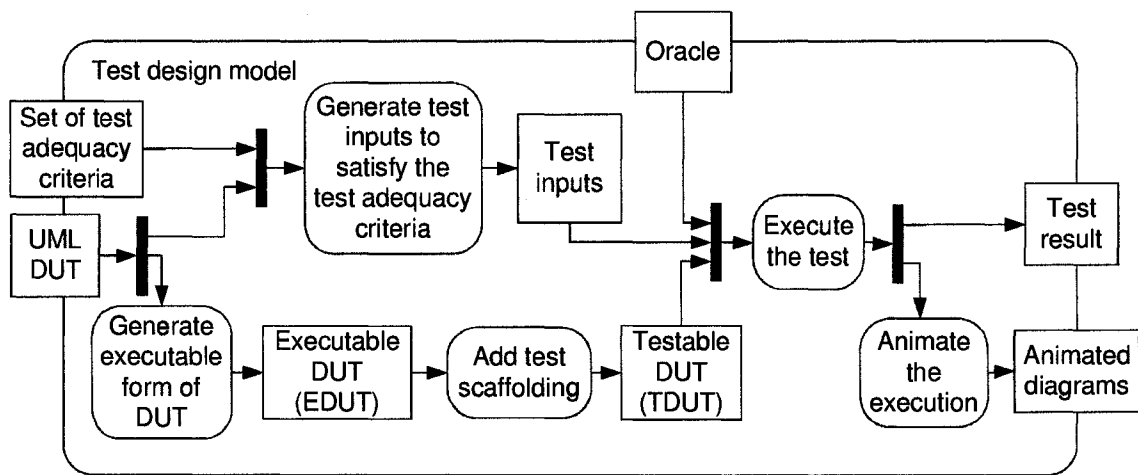


Figure 4.1: Test Approach.

Figure 4.1 summarizes the testing approach. Testing begins when a tester provides the UML design model under test, *DUT*, to the testing system and selects a set of test adequacy criteria that were described in Andrews et al. [3]. These criteria are used to create test objectives for test input generation and to assess test adequacy once testing is completed. The *DUT* is first transformed to an executable form, *EDUT*, and then into a testable form, *TDUT*, that also contains the test infrastructure. Testing is performed by executing the *TDUT* on the generated test inputs. An oracle defines

the expected behavior of the system. Test execution is visualized using animated object and sequence diagrams.

4.1 Generating Test Inputs

The test case generation technique aims at deriving test inputs that satisfy sequence diagram based test adequacy criteria. Originally, these criteria were defined in terms of collaboration diagram elements [3]. We restate the criteria for the more popular sequence diagrams. A sequence diagram based test adequacy criterion defines a set of sequence diagram elements that need to be covered during testing:

1. *All Message Coverage (Mesg)* criterion: Testing must cause each message in a sequence diagram to be sent at least once.
2. *Condition Coverage (Cond)* criterion: Testing must cause each condition in each decision to evaluate to both **TRUE** and **FALSE**.
3. *All Message Path Coverage (Path)* criterion: Testing must cause each possible message path in the sequence diagram to be traversed at least once.

A test input, t , is generated using information from the class model and one sequence diagram, sd . The test case t is used to test one of the scenarios that is specified by sd . A set of inputs generated from sd is used to test a variety of scenarios specified by sd .

A test input is a pair consisting of a start configuration, S , and an operation call event with a set of parameter values, P . Before a test is performed, the system must be brought to the start configuration, S . The start configuration contains the set of objects and links necessary to make the system operation call that starts test execution. The system operation takes P as its arguments.

The generation of test inputs is accomplished by processing a representation, called a Variable Assignment Graph (VAG), that describes the conditions under which paths

in a sequence diagram are executed. The information found in a VAG is obtained from the class diagram and the sequence diagram describing the scenarios that are the target of the tests.

4.2 Generating the Executable Form

The testing system transforms the *DUT* into an executable form, *EDUT*, which is a program that simulates the behaviors modeled in the *DUT*. The *EDUT* utilizes information from structural (class diagrams) and dynamic (activity diagrams) descriptions of the design to simulate modeled behavior. The *EDUT* contains two parts: a static structure representing the runtime configuration of the *DUT*, and a simulation engine. The static structure generated from class diagrams can create and maintain runtime configurations of the *DUT*. A configuration contains objects, their attribute values, and the links between them. The simulation engine is generated from activity models, which are represented using a Java-like Action Language, JAL, specifications. This engine decodes system events, triggering sequences of actions according to the information in the activity diagrams, and sends a sequence of signals to the *EDUT* static structure to update the configuration. The update involves adding and removing objects and links, as well as modifying attribute values.

4.3 Generating the Testable Form

The *TDUT* is obtained by adding test scaffolding to the *EDUT* to automate test execution and failure detection. To automate test execution, code is inserted into the *EDUT* to facilitate the creation of initial configurations and the application of test inputs to the *TDUT*. Failure detection is done by executing a set of checks for failure conditions. Code is inserted to performed the following checks:

1. Are the variables in conditions (such as transition guards in activity diagrams) initialized?

2. Are the parameters passed in operation calls initialized?
3. Does the target object of an operation call exist?
4. Does the pre-condition hold before operation execution?
5. Does the post-condition hold after operation execution?
6. Does the configuration produced by the execution of system events conform to constraints expressed in class diagrams? The set of constraints includes the association-end multiplicity constraints and any other constraints expressed in OCL. These constraints must hold after the execution of every system operation call.
7. Are the user-defined oracle constraints satisfied?

4.4 Executing Tests and Detecting Failures

Testing is performed by executing the *TDUT* with provided test inputs. During test execution, the effects of system behaviors modeled by activity diagrams are recorded and observed in terms of changes in the system state, where a system state is represented as an object configuration. As the test is executed, the runtime configuration is updated to reflect changes in the system state. The changes include creation and destruction of objects and links, as well as the modification of object attribute values.

During test execution, the *TDUT* detects failures by checking the failure conditions described in the previous section. The *TDUT* reports a failure if any of the above checks return a negative answer. Possible causes for test failures are given below:

1. Failing checks 1 – 3 indicates that there may be a fault in the activity diagram. Because of the fault, some variable is used before its value is defined.
2. Failing checks 4 or 5, indicates that the activity diagrams, the pre-conditions, and/or the post-conditions may be faulty.

3. Failing check number 6 indicates that the class diagrams or the activity diagrams may be faulty.
4. Failing check number 7 indicates that the design models under test do not capture the intended behaviors.

4.5 Animating the execution

For animation purposes, an execution observer is used. Whenever there is any change in the system configuration or some action is executed, the observer is notified. The observer then interacts with a graphical user interface, and the changes are displayed to the user with the help of animated object and sequence diagrams. During animation, an object diagram shows the current object configuration of the system. Sequence diagrams show the interaction between objects during test execution.

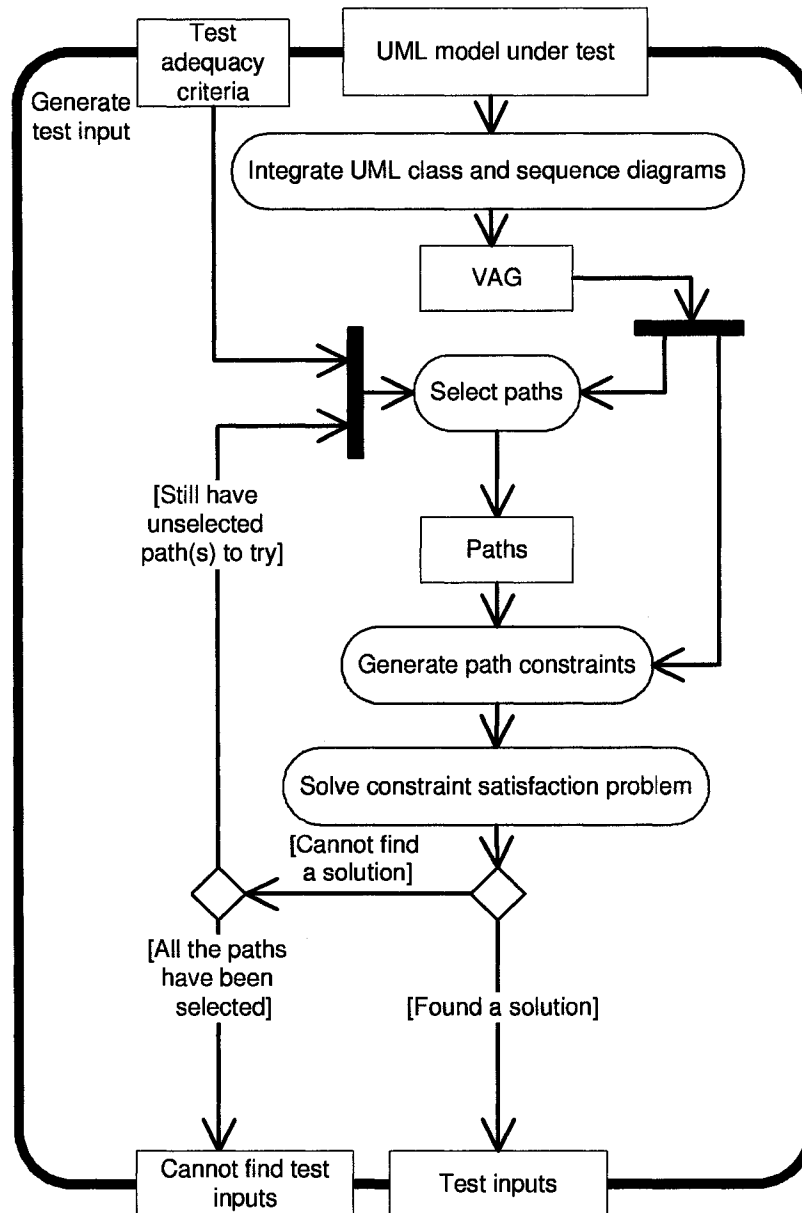
Chapter 5

Generating Design Test Inputs

The activity diagram in Figure 5.1 expands on the activity “Generate test inputs” in Figure 4.1. Information in the class model and a sequence diagram is used to derive a directed graph called the Variable Assignment Graph (*VAG*), which is used to analyze execution paths in the sequence diagram. In order to produce test inputs, a *VAG* needs to contain information on (1) the order in which messages are sent, (2) how variables are changed when sequence diagram messages are received and handled, and (3) what objects and links need to exist when each message is sent.

To obtain test inputs that satisfy a sequence diagram (SD) based criterion, we first select a set of paths in the *VAG* representing execution paths that cover the sequence diagram elements referred to by the criterion. For each selected path in a *VAG*, we determine the constraint on the test input that needs to be satisfied to execute the path. This constraint is the conjunction of (1) the class diagram invariant, (2) the pre-condition of the system operation that initiates the sequence diagram, and (3) the path constraint.

The class diagram invariant defines the set of valid configurations. In our approach, a system under test needs to be in a valid configuration both before and after the execution of any operation system. Hence, during test input generation, the class diagram invariant is taken into consideration to ensure that the generated start configuration is valid. We are not concerned about the validity of the final configuration



at this point because the final configuration is validated during test execution and it plays a role in determining if a test passed or failed.

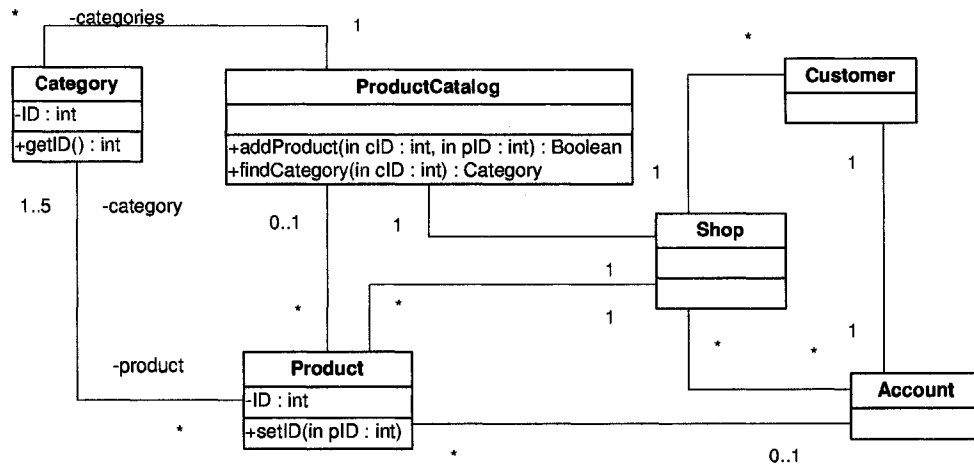
The UML specification [56] states that developers of an operation can assume that the pre-condition of the operation is satisfied before the operation is called.

Our approach ensures that the generated start configuration and parameters satisfy the pre-condition of the system operation that initiates the sequence diagram under test. On the other hand, the pre-conditions of the other operations that are called during the execution of this sequence diagram are not considered during the test input generation process. Instead, they are checked during test execution as part of failure detection.

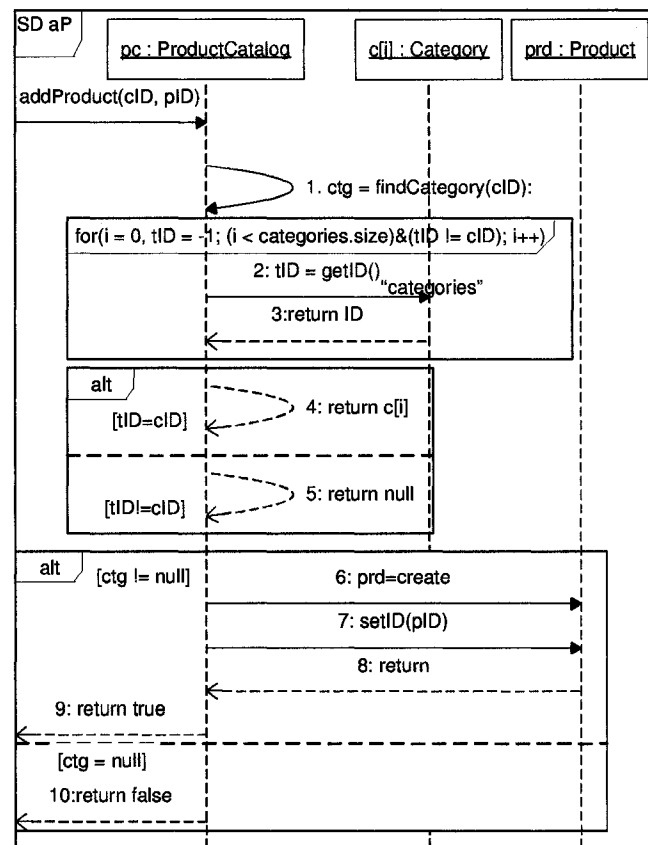
A path constraint is the condition on test inputs that is derived from the path using symbolic execution [10]. The path constraints, along with the class diagram invariant, and the pre-condition of the operation that initiates the sequence diagram under test, are solved using a constraint satisfaction solver. If the solver finds a solution, that solution is the generated test input. If the solver cannot find a solution, the path is discarded and a set of alternative paths is selected to satisfy the chosen criterion. It is possible that all the paths may have been selected but a set of inputs that satisfy the criterion cannot be found.

In practice, a sequence diagram, *sd*, of a system operation may refer to another sequence diagram, *rd*. In such cases, we require that *rd* be integrated with *sd* before the test inputs for *sd* are generated. Also, if *rd* is referred to by two sequence diagrams, *sd_i* and *sd_j*, and the test inputs generated from *sd_i* cover all messages (or conditions) in *rd*, then these elements of *rd* do not need to be covered again when test inputs are generated from *sd_j*.

The rest of this section discusses each activity in the test input generation process. We illustrate the activities using a UML design model for a product-catalog management feature (see Figure 5.2) of an online shopping system, OSHOP. The class diagram in Figure 5.2(a) shows that the *Products* are grouped into various *Categories*, which are managed using a *ProductCatalog*. A new *Product* can be added to the system by invoking the operation `ProductCatalog::addProduct(int cID, int pID)`. The *ProductCatalog* searches the set of *Categories* to find an instance that has



(a) Class Diagram



(b) Sequence Diagram

Figure 5.2: UML Design Model for Product-Catalog Management.

the attribute, *ID*, matching the parameter, *cID*. If such an instance is found, a new *Product* object is created. The attribute, *ID*, of the new object is assigned the value, *pID*. This scenario is specified in the sequence diagram in Figure 5.2(b).

5.1 Generating the Variable Assignment Graph

A number of existing graph-based path selection approaches have been developed to generate code test inputs that satisfy control flow based criteria [53], which are similar to the test adequacy criteria used in this paper. In order to utilize these existing path selection approaches, we transform UML sequence diagrams into directed graphs, VAGs. For example, Figure 5.3 shows the VAG that is generated from the sequence diagram that is shown in Figure 5.2.

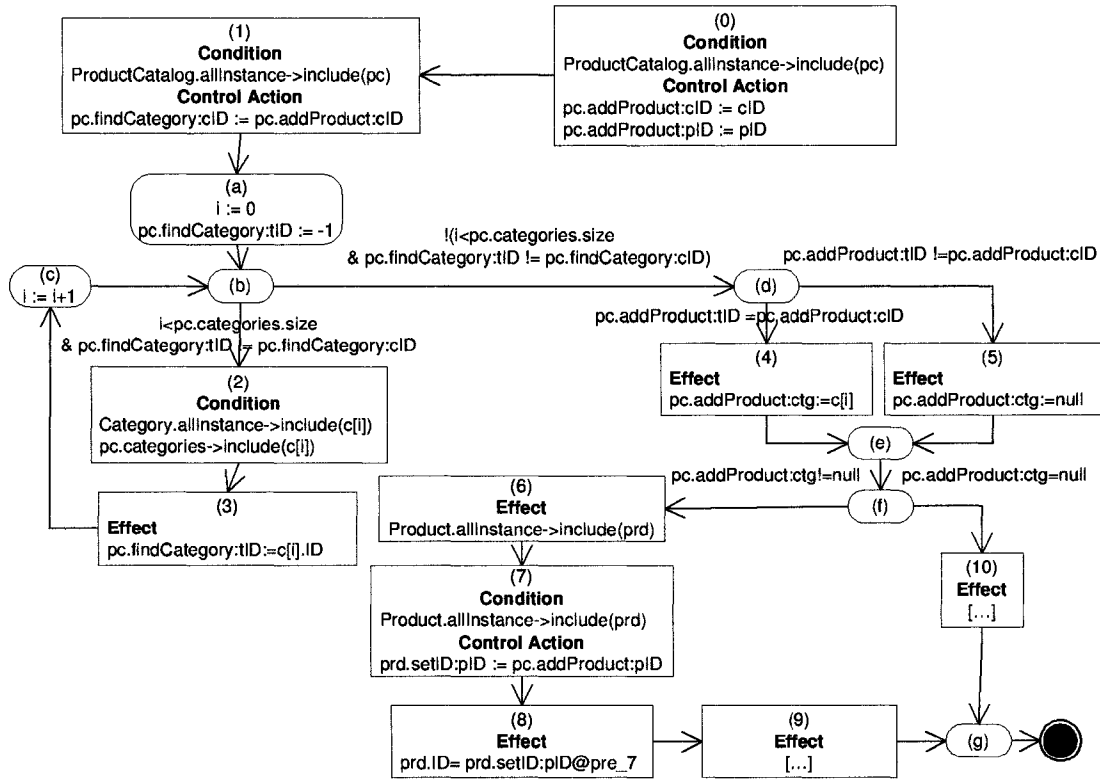


Figure 5.3: A VAG Example.

Just like code blocks are represented as nodes in program control flow graphs, each sequence diagram message is transformed into a VAG message node (e.g., nodes (0), ..., (10) in Figure 5.3 are generated from messages 0, ..., 10 in Figure 5.2). A message node records (1) how variables are changed when the corresponding message is received and handled, and (2) the objects and links that need to exist when the message is sent. A message node is composed of three parts: *Condition*, *Control action*, and *Effect*. The *Condition* part of a node records the links and objects that need to exist to enable the sending and receiving of the corresponding message. The *Control action* part records the assignment of the actual parameter values to the formal parameters. The *Effect* part records the changes to variable values after the execution of the action associated with the corresponding message. Any part of a node can be empty.

A VAG also has control nodes, which are used to represent (1) merging and branching of paths, (2) loop control, and (3) termination of execution. For example, in Figure 5.3, nodes (d) and (f) represent branching of paths, nodes (e) and (g) represent merging of paths, and nodes (b) and (c) represent loop control. A directed edge from a node, N_1 , to a node, N_2 , indicates that the changes recorded in node N_1 occur before the changes recorded in node N_2 . An edge can be associated with a predicate, which is the branching predicate in the sequence diagrams. In a UML sequence diagram, branching predicates include message conditions, conditions associated with alternative fragments, and conditions associated with loop fragments. The branch predicates in a VAG are shown by text enclosed in square brackets.

In sequence diagrams, different variables in different namespaces can have the same name. The namespaces of a variable can be determined based on the message in which the variable is used. For example, `cID` in message 1 in Figure 5.2 is used within the context of message 0, indicating that the name refers to the parameter `cID` of the operation `pc.addProduct()`. On the other hand, `cID` in the loop condi-

tion is used within the context of message 1, indicating that this name now refers to the parameter of the operation `pc.findCategory()`. When sequence diagram variables are transformed into VAG variables, they need to be fully qualified to avoid confusion. For example, the name `cID` in the sequence diagrams is transformed into `pc.addProduct:cID` in node (1) and `pc.findCategory:cID` in control node (a) in Figure 5.3.

We find test inputs by analyzing the relationship between the variables that are used in the sequence diagram. To do this, we need to know how and when variables are changed. We store the information about the changes of variables in the VAG nodes. Information about the changes in variable values is gathered from sequence diagrams and operation pre- and post-conditions:

1. When *call operation* and *create* messages are received, actual parameters are assigned into formal parameters. This assignment is stored in the *Control* part of the corresponding VAG message nodes. For example, the *Control action* part in node (1) records that the value of the actual parameter, `pc.addProduct:cID`, is assigned to the formal parameter `pc.findCategory:cID`. When the call operation message does not have any parameter, the *Control action* part is empty.
2. When *create* and *destroy* messages are received, the associated actions are executed, resulting in the creation and destruction of objects and links. When a *destroy object* action is executed, we assume that all the links associated with the destroyed object are also destroyed. The changes are stored in the *Effect* part of the corresponding VAG message nodes. For example, message node (6) in Figure 5.3 records that object of the class `Product`, `prd`, is created as a result of the execution of the *create* action associated with message 6 in Figure 5.2.
3. When a *return* message is received, the execution of the operation associated with the corresponding *call operation* message is finished, and the effect of the operation is completely realized. The effect of the operation, which is specified in

the operation post-condition, is recorded in the *Effect* part of the corresponding VAG message node. For example, message node (8) records the effect of the operation that is executed in response to the receipt of message 7. The effect is the assignment of the value of variable, `pID`, when message 7 is sent to the attribute, `prd.ID`.

Also, when a *return* message is received, the return value may be assigned into the appropriate variable as indicated in the *call operation* message. This effect is also recorded in the *Effect* part of the message node. For example, message node (6) records the assignment of the return value, `c[i]`, to the variable, `pc.findCategory:tID`.

4. The values of loop index variables are changed when execution iterates through loops. Information about the changes is stored in VAG control nodes. For example, the loop structure in Figure 7 is represented using control nodes (a), (b), and (c). Node (a) represents the initialization of the loop index variables, `i` and `tID`. Node (b) represents the evaluation of the loop condition, and node (c) represents the modification of index variable `i`.

The start configuration in a test input that exercises an execution path must contain a set of objects and links that enable all messages in the path to be sent. In our approach, we gather the constraint on the required objects and links that enable the sending of each message and store the constraint in the *Condition* part of the corresponding VAG message node. The constraint includes statements specifying the existence of the recipient of the message and the link between the sender and the receiver objects. For example, the *Condition* part of node (1) states that message 1 can be sent if object `pc` exists. The *Condition* part is empty if the message is a return message, because it is known that the recipient object and the link exist. For example, the *Condition* part of node (8) is empty, because message 8 is only sent after message 7 is sent, implying that both objects `pc` and `prd`, and the link between them

exist. The *Condition* part is also empty if the message is a create message (because the recipient object is created after the message is sent) or a call message associated with a static (class) operation. For example, see node (6), which corresponds to message 6.

We now describe the rules to generate a *VAG* from a class diagram and a sequence diagram.

1. The variables in a VAG are used to store object references, attributes values, association ends, operation parameters, and values of local variables. VAG variable names are fully qualified to avoid naming ambiguities as follows:

- VAG variables representing objects use the name of the corresponding objects in the sequence diagram. For example, `c[i]` in node (2) refers to object `c[i]` in the sequence diagram.
- Consider a variable, `v`, that is used within a method `m()` of a class `C`. If the name of `v` is the same as the name of an attribute of `C` (or a parameter of `m`), then `v` represents that attribute (or that operation parameter). For example, in message 3, `ID` represents attribute `c[i].ID`. If the name of `v`, is different from the names of all attributes and parameters, then `v` is a local variable.
- An attribute, or an association end, `a`, in an object, `o`, is referred to by a variable named `o.a` in the VAG. For example, `c[i].ID` in node (2) represents attribute `ID` of the object `c[i]`.
- A parameter, `p`, of a method, `m`, in an object, `o`, is referred to by a variable named `o.m:p` in the VAG. An example is the variable `pc.findCategory:cID` in node (1).
- A local variable, `v`, in a method, `m` of an object, `o`, is referred to by a variable named `o.m:v` in the VAG. An example is the variable `pc.findCategory:tID` in node (3).

2. Each message, M_i , in the sequence diagram that is sent from an object, o_s , to an object, o_r , is transformed into a VAG node, N_i . If M_i is sent via an instance of an association between o_s and o_r , the *Condition* part of N_i records that the link must exist. In this case, we require that M_i be annotated with the name of the corresponding association end at o_r . For example, message 2 is annotated to indicate that it is sent via an instance of an association that has the association end, `ProductCatalog::categories`. This assumption is needed because two objects can be linked using different association instances.

If M_i is a call or destroy message, the *Condition* part of N_i also records that o_r must exist. The contents of the other parts of N_i are produced as follows:

- If M_i is a create message, the *Effect* part of N_i records the initialization of the attributes of the created object and the definition of the variable that holds the object handle. The initial values of attributes are obtained from class diagrams.
- If M_i is a destroy message, the *Effect* part of N_i records the definition of the variable that holds the object handle, which is set to null.
- If M_i is a return message of a call message M_j , the assignment of the variable that holds the return value is stored in the *Effect* part. The assignment is denoted with the notation “:=” to distinguish it from “=” used in post conditions. Also, the *Effect* part of N_i records any other variable updates required by the post-condition of the called operation. If a variable, v , is expressed in M_j as $v@pre$, it will be denoted as $v@pre-j$ in the N_i to indicate that it refers to the value of v before M_j was sent. For example, variable `setID:pID` in node (8) is renamed to `setID:pID@pre-7`, because it refers to the value of `setID:pID` in node (7). We assume that there is no parameter that has the same name as an attribute in the associated class.

- If M_i is a call message, the *Control Action* part records the assignment of the actual parameters to the formal parameters. Actual parameters are specified in sequence diagrams while formal parameters are specified in class diagrams.
3. If a message, M_{i+1} , is sent right after the message, M_i , an edge from N_i to N_{i+1} is created.
 4. If a message M_i is followed by a condition structure that contains a set of predicates, (p_1, p_2, \dots, p_j) , and a set of corresponding messages, $(M_{i+1}, \dots, M_{i+j})$, (i.e., M_{i+k} is sent after M_i when p_k evaluates to true), then the following actions are performed:
 - Create a control node, C_i , and an edge from N_i to C_i .
 - Create j edges from the node C_i to N_{i+1}, \dots, N_{i+j} . Each edge (N_i, N_{i+k}) is associated with the predicate, p_k .
 - Create a control node C'_i and j edges from N_{i+1}, \dots, N_{i+j} to C'_i .
 - If the condition structure does not explicitly model an *else* predicate, create an edge from C_i to C'_i . This edge is associated with the *else* predicate and models the case when $p_1 \wedge p_2 \wedge \dots \wedge p_j = \text{False}$.
 5. If a message, M_i , is followed by a loop structure that matches the pattern shown in Figure 5.4(a), the loop is transformed into the structure shown in Figure 5.4(b).

The above transformation rules are based on two assumptions: (1) all objects in the sequence diagram have names, and (2) all return messages are shown. The first assumption is needed for the renaming of variables. In practice, an unnamed object can always be assigned an arbitrary name. The second assumption allows one to determine when the effect of an operation is completely realized.

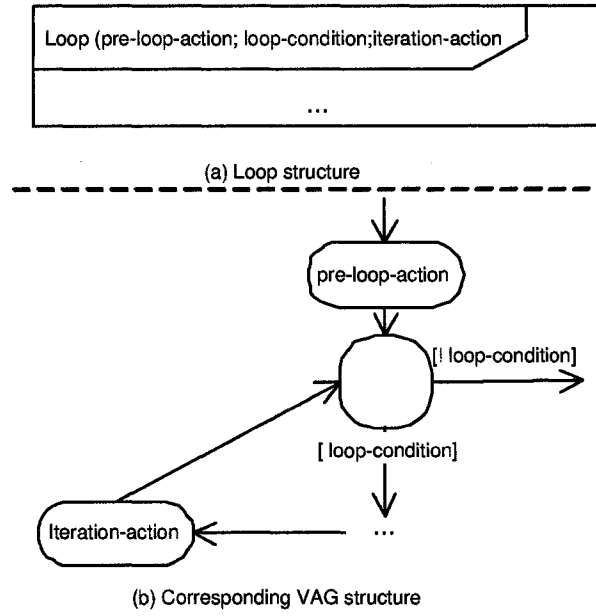


Figure 5.4: Transformation Rule for Sequence Diagram Loop Structures.

5.2 Selecting Complete Paths

The goal of the path selection process is to derive a set of complete execution paths in a *VAG* that satisfy a test adequacy criterion. A complete path in a *VAG* is a path from the node (0) to the termination node.

Suppose we have a set of complete paths, S , such that every node in the *VAG* belongs to a path in S . Since each message in a sequence diagram is transformed into a *VAG* message node (transformation rule number 2), a test set that covers S will force each message to be sent at least once. Hence, such a test set will satisfy the “*All Message Coverage*” criterion. Similarly, a test set that covers a set of complete paths that include all *VAG* edges satisfies the “*Condition coverage*” criterion because each decision branch in a sequence diagram is transformed into a *VAG* edge (transformation rules number 4 and 5). A test set that covers all *VAG* complete paths satisfies the “*All Message Paths*” criterion.

Finding the set of complete paths that covers all *VAG* nodes or edges is similar to finding paths that cover all statements or branches, respectively, in a program. Researchers have proposed various path selection techniques with different goals. Krause et al. [39], Miller et al. [44], Ntafos and Hakimi [47], and Wang et al. [60] aim at minimizing the number of generated paths. Krause et al. repeatedly choose the next most effective path that covers most of the remaining uncovered elements. Miller et al. use a heuristic procedure where programs are decomposed into decision-to-decision paths, which are then combined to form an optimal path. Ntafos et al. discuss a network-theory approach and demonstrated that the size of the minimum path can be determined by applying a minimum flow method or a maximum matching method. Wang et al. formulate the path selection problem as a zero-one integer programming problem and provided a generalized optimal model. Instead of attempting to generate a set of minimum paths, Bertolino et al. [5] utilize dominance and implication graphs to generate paths that are likely to be feasible. They assume that the smaller the number of predicates in a path, the more likely the path is to be feasible.

Any of the above path selection techniques can be used in our approach. A tester can use knowledge about the model to choose an appropriate path generation approach. For example, if the tester knows that most of the paths in a model are feasible, a technique aiming at a minimal set of complete paths is preferred. Otherwise, it is better to choose a path selection technique that generates complete paths that are more likely to be feasible.

When a model contains an unbounded loop, the “*All Message Paths*” criterion cannot be satisfied because the number of paths is infinite. In practice, testers may set a finite bound or apply Binder’s iteration coverage criterion [7] to relax the “*All Message Paths*” criterion. For the iteration coverage criterion, each loop is required to be executed zero, one, and a large number of times. In such cases, the technique

described by Pilskalns et al. [50] can be used to generate complete paths that satisfy the “*All Message Paths*” criterion.

5.3 Generating Path Constraints

Our path constraint generation technique is based on symbolic execution [10]. In existing symbolic execution techniques, the test inputs are either program parameters or variables that are defined using input statements. In our approach, inputs are (1) the parameters of the system operation that initiates the sequence diagram, and (2) the variables that are used to define the start configuration. These variables contain values representing class instances, attribute values, or association instances.

Variables that define a start configuration are those that are used without being defined (i.e., assigned a value) within the VAG. The values of these variables are set before the sequence diagram is executed because the start configuration must exist before the corresponding system operation is called. In other words, the values of these variables are set when the start configuration is built. Hence, in our approach, we consider a variable, v , as an input that defines a start configuration if (1) v represents class instances, attribute values, or association instances, and (2) v is used before being defined in the VAG.

For example, the inputs for the path *0-1-2-3-5-10* in the VAG shown in Figure 5.3 include the parameters *cID* and *pID*, as well as the variables listed in Table 5.1. The first column of the table lists the inputs; the second and the third columns list the edges, nodes, and node parts in which the variables are first used. These variables are all used before they are defined in the VAG, and thus, they are considered input variables.

To facilitate generation of path constraints, each selected path is transformed into an equivalent static single assignment (SSA) form [11]. In an SSA form, there is only one assignment for each variable. Hence, if two variables have the same name, they

Table 5.1: Variables Defining the Start Configuration for the Path 0-1-2-3-5-10.

Variable	First use	Compartment
<code>pc</code>	Node 1	<i>Condition</i>
<code>pc.categories.size</code>	Edge b-2	<i>Branch condition</i>
<code>pc.categories, c[i]</code>	Node 2	<i>Condition</i>
<code>c[i].ID</code>	Node 3	<i>Effect</i>

contain the same value. The transformation of a path into an SSA form can be done as follows:

1. Whenever a variable, v , is set to a new value, we state that v is redefined. In VAGs, all variables in the *Effect* part, except those that appear in the right hand side of an assignment operator (“:=”) and those denoted with *@pre* notations, are considered newly defined. Also, the variables appearing in the left hand side of the operator (“:=”) in the *Control Action* parts are also considered newly defined. For example, in Figure 5.3, variables `prd.pID` and `pc.addProduct:ctg` are redefined in nodes (8) and (4), respectively.
2. For every definition of a variable, v , generate a new SSA variable, v_j , where j is a unique number. Replace v with v_j in the definition. For example, the SSA form for the node (c) in the path 0-1-2-3-5-10 in Figure 5.3 is shown in line 12 in Figure 5.5:

$$i_1 = i_0 + 1$$

3. For every subsequent use of v before v is redefined, replace v with v_j . For example, the SSA form for the variable j in the edge that connects nodes (b) and (d) in the path 0-1-2-3-5-10 in Figure 5.3 is i_1 as shown in line 13 in Figure 5.5.
4. A variable $u@pre_n$ denotes a value of the variable u at node n of the VAG. Hence $u@pre_n$ is replaced by the SSA form of the variable u at node n . For example, `prd.setID:pID@pre_7` in node (8) refers to the value of `prd.setID:pID` in

node (7). In node (7) of the path *0-1-2-3-4-6-7-8-9*, the SSA form of this variable is `prd.setID:pID_0`. Thus, `prd.setID:pID@pre_7` is replaced by the SSA form, `prd.setID:pID_0`.

An important property of the SSA form is that all assignments can be treated as conditions (i.e., boolean expression). From the SSA form of a path, the path constraints can be constructed by forming the conjunction of the following conditions:

1. All the conditions in the *Condition* part.
2. All the branch predicates.
3. All variable definitions in the *Control Action* and *Effect* parts.

A part of the constraints for the path *0-1-2-3-5-10* is shown in Figure 5.5.

```

[1] ProductCatalog.allInstance -> include (pc)
[2] pc.addProduct:cID = cID
[3] pc.addProduct:pID = pID
[4] ProductCatalog.allInstance->include(pc)
[5] pc.findCategory:cID = pc.addProduct:cID
[6] i_0=0
[7] pc.findCategory:tID_0 = -1
[8] i_0<pc.categories.size & pc.findCategory:tID_0 !=
    pc.findCategory:cID
[9] Category.allInstance -> include (c[i_0])
[10] pc.categories -> include (c[i_0])
[11] pc.findCategory:tID_1 = c[i_0].ID
[12] i_1 = i_0 + 1
[13] !(i_1 < pc.categories.size & pc.findCategory:tID_1 !=
    pc.findCategory:cID)
[14] pc.addProduct:tID_1 != pc.addProduct:cID
[15] pc.addProduct.ctg = null
[16] pc.addProduct.ctg=null
[...]
```

Figure 5.5: The Constraints for the Path 0-1-2-3-5-10.

5.4 Solving Constraints

The path constraints produced in our test generation approach contain numerical symbols (e.g., `i_0`) as well as symbols that represent system configurations (e.g., `pc` and `pc.categories`). Hence, our constraint satisfaction problem cannot be solved by pure numerical constraint solvers (e.g., the e-box consistency based constraint solver [58]).

We use the constraint solver, Alloy [29], which allows the specification of constraints that contain integer, set and relation symbols. We use the Alloy constraint language to express the full set of constraints that contains the path constraints, the pre-condition of the system operation and the class diagram invariant. If Alloy finds a solution for the full set of constraints, this solution is transformed into a test input.

We extend the rules proposed by Massoni, Gheyi, and Borba [41] to transform UML class diagrams and OCL constraints into Alloy. Massoni et al. transformed a UML *class*, `c`, into an Alloy *signature*, `s`, which represent a set. The *properties* are transformed into Alloy *relations*. An instance, `o`, of `c` is transformed into an element, `e`, of the set represented by `s`.

Figure 5.6 shows a part of the Alloy script that is generated from the class diagram in Figure 5.2 and the OCL constraint shown in Figure 5.5. The start configuration must conform to the constraints specified in the class diagram and the system operation pre-condition. Intermediate configurations that are created during test execution do not need to conform to these constraints because we assume that the class diagram constraints only hold after the execution of a system operation call. We are not concerned about the validation of the final configuration at this point because the final configuration will be validated during test execution. Hence, in Alloy we only specify constraints on entities that play a role in the initial configurations (as shown in rule number 2 below). We now describe the rules to transform class diagrams into Alloy:

```

//List of classes
[1]sig ProductCatalog{}

[2]sig Category{}

[3]//The objects in the initial configuration

[4]//Class ProductCatalog
[5]  one sig ProductCatalog0 in ProductCatalog {
[6]    categories: set Category0
[7]  }

//Class Category
[8]  sig Category0 {
[9]    catalog: one ProductCatalog0,
[10]    products: set Product0,
[11]    ID: one Int
[12]  }
[...]
```

*/*Associations*/*

```

[13]  fact defineCatalogCategory{
[14]    all pc:ProductCatalog0|
      all c:Category0|(pc in c.catalog)
      <=>(c in pc.categories)
[15]  }
[...]
```

*/*Path constraint*/*

```

[16]  fact path_constraint{
[...]
```

[17] c_i_0 in pc.category

[...]

[18]}

Figure 5.6: A Part of the Constraint for the Path 0-1-2-3-5-10 in Alloy Language.

1. Each class C is transformed into a signature S . The Alloy signature S represents the set of all instances of UML class C that is created during testing. For example, lines 1 and 2 in Figure 5.6 specify the set of all objects of class `ProductCatalog` and `Category`, respectively.

2. For each UML class C , an Alloy signature, S_0 , is created to represent instances of class C in the initial configuration. For example, segments 4-7 and 8-12 in Figure 5.6 specify the instances of **ProductCatalog** and **Category** in an initial configuration. The rules to create S_0 are described in Massoni et al.:

- A UML *class*, C , is transformed into an Alloy *signature*, S_0 .
- A UML *attribute*, A , which is restricted to have the type *integer*, in a class, C , is transformed into an Alloy *relation*, R , which relates the corresponding signature S_0 to the set *Integer*. For example, line 11 in Figure 5.5 represents the attribute, **Category::ID**.
- Suppose two UML classes, C_1 and C_2 , are transformed into two Alloy signatures, S_{0_1} and S_{0_2} , respectively. A UML *association*, **AS** between C_1 and C_2 is transformed into two Alloy relations, R_1 and R_2 . R_1 and R_2 are defined in the signatures, S_{0_1} and S_{0_2} , respectively. Even though in general, R_1 and R_2 can refer to two different relationships, in this case they describe the same relation between S_{0_1} and S_{0_2} . For example, lines 6 and 9 represent the association between classes **ProductCatalog** and **Category**. We define an Alloy *fact* to indicate that R_1 and R_2 refer to the same relationship (e.g., lines 13-15 in Figure 5.5 specify that **ProductCatalog0.categories** and **Category0.catalog** refers to the same association):

```
fact defineAS{
    all s1:S_1 | all s2:S_2 |
    (s1 in s2.R_2)<=>(s2 in s1.R_1)
}
```

The path constraints, the class diagram invariants, and the pre-condition, all represented using OCL, are transformed into Alloy using the rules described in Massoni et al. For example, line 17 is an Alloy invariant that is generated from line 10 of the OCL constraint shown in Figure 5.5.

Generally, the domain of configurations that a constraint solver needs to search is infinite. However, Alloy can solve the constraint problems if we restrict the search range by setting a maximum total number of objects in the solution. In such cases, Alloy will either give us a solution or report that there is no valid solution within the range.

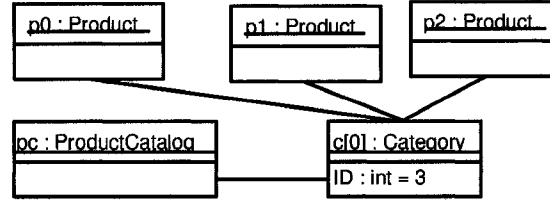


Figure 5.7: Start Configuration for the Path 0-1-2-3-5-10.

Alloy solutions are given in terms of sets and relations. We extract a start configuration from a solution as follows:

1. We first find all the sets, S_0 , that represent the initial configuration. For each instance, s , in a set S_0 , we create an instance, c , of class C .
2. If s has a relation, R , that represents an attribute, $C.A$, we set the attribute value, $c.A$, according to the value of $s.R$.
3. Suppose that there are two sets, S_1 and S_2 , that represent two classes, C_1 and C_2 . There are two elements, s_1 of set S_1 and s_2 of set S_2 , that represent two instances, c_1 of C_1 and c_2 of C_2 , respectively. If s_1 and s_2 relate to each other via two relations R_1 and R_2 , which represent the same association, AS , between two classes C_i and C_j , we create a link between c_i and c_j .

When we use Alloy to solve the constraints for the path 0-1-2-3-5-10 in Figure 5.3 with the maximum number of objects set to 5, we get a solution with $(cID, pID) = (0, 0)$, and a start configuration as shown in Figure 5.7.

Chapter 6

Java-Like Action Language

The Java-like Action Language (JAL) can be used to specify actions within the context of a UML activity diagram. In our approach, JAL is used to describe the sequence of actions performed by a class instance during the execution of an operation call. The following types of actions can be described using JAL: call operation actions, calculation actions, create and destroy object actions, create and destroy link actions, read and write link actions, read and write attribute actions, and read and write variable actions. Developers can use JAL to express an activity diagram in a textual format. The current version of JAL supports only synchronous operation invocations.

JAL provides access to the data described by class diagrams. Identifiers that are defined in class diagrams, such as class, operation, attribute names, and parameter names can be used in a JAL specification.

JAL follows the UML standard's recommendation for action languages and provides constructs for describing control structures. These are loops and conditions, which have the same syntactic structures as the 'if' and 'while' statements in Java, respectively.

Figure 6.1 shows an example of the JAL specification of the operation `ProductCatalog::addProduct(int cID, int pID)`. Class `ProductCatalog` belongs to the class diagram that is shown in Figure 5.2. When the operation is called, the target object of the call searches for a `Category` instance, `ctg`, with the attribute

value `ctg.ID` that matches the parameter, `cID`. If such an instance is found, an object, `prd`, of the type, `Product` is created. Object `prd` is then linked to `ctg` before the operation returns *TRUE*. If there is no `Category` instance that matches `cID`, the operation returns *FALSE*.

```
[1]  Category ctg;
[2]  ctg = this.findCategory(cID);
[3]  if(ctg != null){
[4]      Product prd;
[5]      prd = _create_object_Product();
[6]      prd.setID(pID);
[7]      _create_link_product_category(ctg, prd);
[8]      return true;
[9]  }
[10] else{
[11]     return false;
[12] }
```

Figure 6.1: The Product-Catalog System: `addProduct` JAL Specification

The following sections describe the syntax of JAL. For a complete specification of JAL, please refer to the Appendix A.

6.1 An overview of JAL statements

A JAL segment specifies the sequence of actions executed within an operation. It consists of a number of JAL statements. A JAL statement can be a simple statement (e.g., an operation call action), a loop, or condition statements. A simple statement can be an expression, a single statement, or a compound statement. A simple statement ends with a semicolon (“;”). An expression represents an action that returns a value. A simple statement represents an atomic action that does not return a value. A compound statement represents a combination of atomic actions.

JAL statements are composed of keywords, logical and arithmetic operators, and identifiers. Identifiers can be defined in class diagrams (e.g., names of classes, attributes, associations, operations, and operation parameters) or in JAL segments as local variables.

JAL identifiers must conform to the following rules:

- Identifiers are case sensitive.
- Identifiers may only contain the characters $[a - z]$, $[A - Z]$, and $[0 - 9]$.
- Identifiers must not start with a numeric character $[0-9]$.
- Identifiers must not be the same as the keywords. JAL has the following keywords: **if**, **when**, **return**, `_get_`, `_set_`, `_create_object_`, `_delete_object_`, `_create_link_`, `_delete_link_`, `_get_At`, `_get_Total`, `_add`, and `_remove`.

A JAL variable needs to be declared before use. A variable can have a pre-defined type or be an object handle. JAL currently supports the following pre-defined types: *integer*, *real*, *boolean*, *String*, and *collection*.

In the following sections, we describe condition and loop statements, simple statements and expressions, compound statements, and the use of collections in JAL.

6.2 JAL control statements

JAL control statements consist of condition statements and loop statements. Condition statements have the following syntax:

```
if ( boolean_expression) {  
    <SEQUENCE_OF_STATEMENTS_1>  
}  
[else {  
    <SEQUENCE_OF_STATEMENTS_2>  
}]
```

If `boolean_expression` evaluates to true, `<SEQUENCE_OF_STATEMENTS_1>` is executed, otherwise, `<SEQUENCE_OF_STATEMENTS_2>` is executed. The `else` branch can be omitted if `<SEQUENCE_OF_STATEMENTS_2>` is empty.

Loop statements in JAL have the following syntax:

```
while( boolean_expression ){
    <SEQUENCE_OF_STATEMENTS>
}
```

The body of the loop, `<SEQUENCE_OF_STATEMENTS>`, is executed when the loop guard, `boolean_expression`, is true.

6.3 JAL single statement and expression

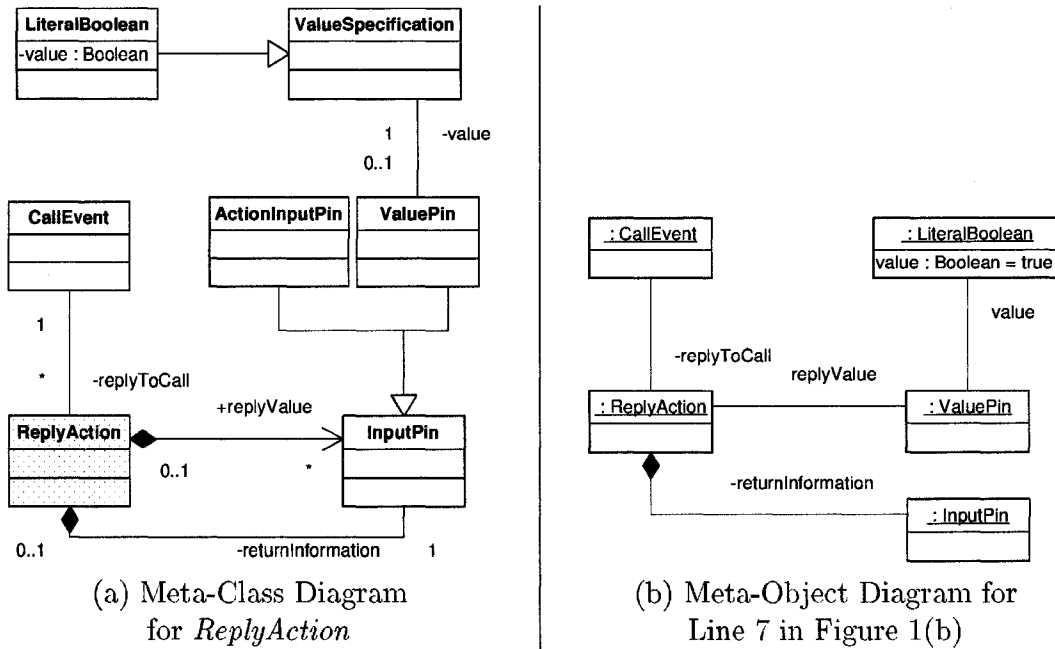


Figure 6.2: ReplyAction in UML 2.0

A UML atomic action is represented using a JAL single statement or expression. For example, line 8 in Figure 6.1 is a JAL return statement, which represents an

atomic *reply action*. A *reply action* returns a value to the caller of the previously accepted *call action*, and terminates the call. Figure 6.2(a) shows the *ReplyAction* and its associated meta-classes as specified in UML 2.0. *ReplyAction.replyValue* denotes a type of *InputPins* that contain returned values. *ReplyAction.replyToCall* denotes the trigger specifying the operation whose call is being replied to. *ReplyAction.returnInformation* is an *InputPin* that contains information that is used to determine the caller of the previously accepted *action call*, who receives the return values. In JAL, a *reply action* is represented as a return statement (e.g., `return retValue`).

The keyword, `return`, represents the return statement and `retValue` is an expression representing the value to be returned. We assume that a *reply action* never returns more than one value. Thus, we can easily transform a *return statement* into a Java return statement, which also returns one value.

The above example illustrates that the JAL *return* statement conforms to the semantics of the *reply action* as specified in the UML standard. A detailed description of JAL's conformance to the other actions is available in the JAL specification document (see appendix A).

JAL call operation, computation, read variable, and add variable value actions have the same syntax as Java method invocation, computation expression, variable expression and variable assignment statements, respectively. An expression is evaluated into a value and can be used as a parameter to another expression or a statement. A statement does not return a value and thus cannot be used as a parameter. JAL also supports the following primitive actions:

- The *CreateObjectAction* is represented by the JAL create object expression, `_create_object_<ClassName>()`.

The expression instantiates an object of the class with the name, `<ClassName>`, and returns the handle of the newly created object.

- The *DestroyObjectAction* is represented by the delete object statement, `_delete_object_(<objectHandle>)`.
The `<objectHandle>` is an expression that evaluates to the handle of an object, `o`. When the statement is executed, `o` gets destroyed along with all its links. However, the owned objects are left unchanged.
- The *CreateLinkAction* is represented by the create link statement, `_create_link_<AssociationName>(<objectHandle1>, <objectHandle2>)`.
The statement creates a link, which is an instance of the association with the name, `<AssociationName>`. This link connects two objects which are represented by two expressions, `<objectHandle1>` and `<objectHandle2>`.
- The *DestroyLinkAction* is represented by the delete link statement, `_delete_link_<AssociationName>(<objectHandle1>, <objectHandle2>)`.
The statement delete the link that connects two objects represented by the expressions `<objectHandle1>` and `<objectHandle2>`. The link is a instance of the association with the name, `<AssociationName>`.
- The *ReadLinkAction* is represented by an association navigation expression, `[ObjectHandle.]<AssociationEndName>`.
This expression evaluates into a read-only collection of objects that associate with the object, `ObjectHandle`, with the association-end named, `AssociationEndName`.
- The *ReadAttributeAction* is represented by the JAL read attribute expression, `[ObjectHandle.]_get_<AttributeName>()`.
- The *WriteAttributeAction* is represented by the JAL write attribute statement, `[ObjectHandle.]_set_<AttributeName>(<value>)`.

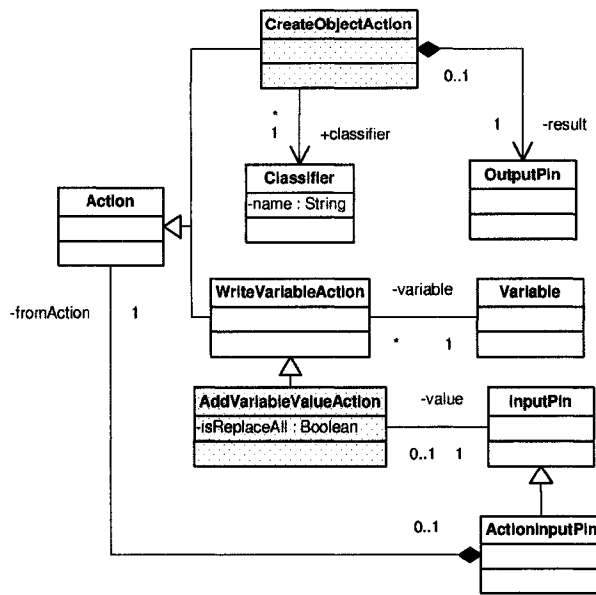
JAL supports ordered collections. A JAL ordered collection can contain only objects of the same type. The following ordered collection operations are supported:

- `<CollectionExpression>._getTotal()`: Get the number of items in the collection.
- `<CollectionExpression>._getAt(<index>)`: Get an item at the `<index>` position in the collection.
- `<CollectionExpression>._add(<Expression>)`: Add an item to the end of the collection.
- `<CollectionExpression>._remove(<index>)` Remove an item at the `<index>` position in the collection.

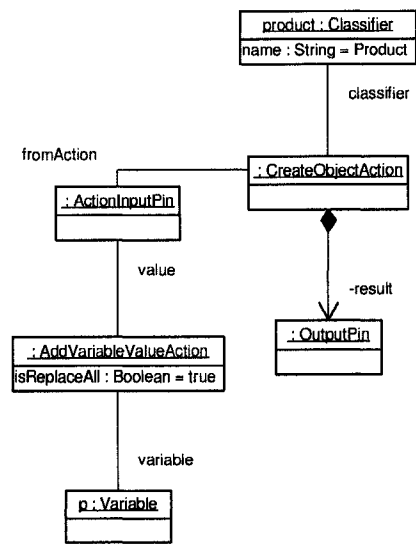
6.4 Compound statements

A JAL compound statement represents a combination of atomic actions and consists of an expression or a single statement that uses another expression as a parameter. For example, line 5 in Figure 6.1 shows a JAL compound statement that describes two atomic actions: a create object action and an add variable value action. Figure 6.3(a) shows the meta-classes that associate with *CreateObjectAction* and *AddVariableValue*. The *CreateObjectAction* instance associates with the instantiated *Classifier* and an *OutputPin* that holds the handle of the newly created object. The *AddVariableValue* instance associates with the updated variable and an *InputPin* that holds the new value. In UML 2.0, an *InputPin* can be an *ActionInputPin*, which can take the output from another action.

Figure 6.3(b) shows a meta-object diagram representing actions that are represented in line 5 of Figure 6.1. The meta-object diagram is an instance of the meta-class diagram shown in Figure 6.3(a). In Figure 6.3(b), the *CreateObjectAction* instantiates the class **Product**. The result of the *CreateObjectAction* is used in the *AddVariableValueAction*, which assigns the result to the variable, *prd*. In line 5 of Figure 6.1, the *CreateObjectAction* is represented using the expression `_create_object.Product()`. The assignment part (`prd = ...`) represents the *AddVariableValueAction*.



(a) Partial Meta-Class Diagram for Action Semantics



(b) Meta-Object Diagram for Line 3 in Figure 1(a)

Figure 6.3: Example of Combination of Atomic Actions in UML 2.0

Chapter 7

Test Execution

In this chapter, we describe a technique for executing a design under test, *DUT*, by transforming it into a Java program. First, we discuss the transformation of the *DUT* into an executable form, *EDUT*. We then describe how the *TDUT* is obtained from the *EDUT*. We show how the code is used to execute a test.

7.1 Generating the *EDUT*

Information from class and activity models is used to create Java programs that simulate the behavior specified in a model under test. UML class, attribute, and operation notations are transformed into Java class, attribute, and method declarations, respectively. For each class, *C*, in a *DUT*, a collection class, **CCollection**, is generated. An instance of **CCollection** maintains a collection of instances of *C*. The **CCollection** class is needed to represent association ends. The **CCollection** class has methods to add (or remove) instances of *C* to (or from) the collection. Association ends are transformed into Java attributes with collection class types.

A singleton class named **TFactory** is generated from the class diagram. This class has public methods to create and destroy instances of every class and association in the class diagrams. The **TFactory** class is used to handle JAL create and destroy links and objects actions. Activity models, represented using JAL, are transformed into Java method bodies.

7.1.1 Transforming class diagrams into EDUT code

We now describe the rules to transform class diagrams into EDUT code. The rules are illustrated using an example shown in Figures 7.1, 7.2, 7.3, and 7.4. Figure 7.1 shows a class diagram with two classes, **C1** and **C2**. **C1** has an attribute, **A**, and an operation, **Opt(p1:T1, p2:T2): ReturnType**. **C1** and **C2** are associated via an association, **C1C2**. Figures 7.2, 7.3, and 7.4 shows *EDUT* code generated from the class diagram.

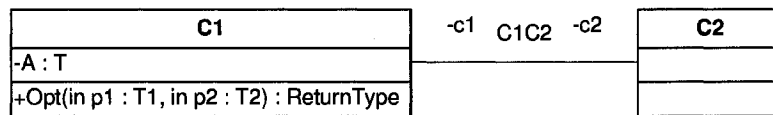


Figure 7.1: A template DUT class diagram

1. For each class model, we create an abstract class **TObject**, as shown in Figure 7.2. Every *DUT* class is transformed into a sub-class of **TObject**. **TObject** has a method, **generateUniqueName()**, which creates a unique *id* for each object in the system. **TObject** also has an abstract method **_destructor()** which is redefined in sub-classes of **TObject**.
2. A UML class, **C1**, is transformed into a Java class with the same name, as shown in line 1 in Figure 7.3. **C1** extends the **TObject** class.
3. A UML attribute, **A**, is transformed into a Java state variable as shown in line 3 in Figure 7.3. Also, for each UML attribute, **A**, Java methods **_set_A()** and **_get_A()** are created, as shown in lines 4-9 in Figure 7.3. These two methods provide an indirect mechanism to modify or to read the value of attribute **A**.
4. A UML operation, **C1::opt(p1:T1, p2:T2):ReturnType**, is transformed into a Java method declaration as shown in lines 21-23 in Figure 7.3. The body of the method is generated from the corresponding JAL specification shown in Figure 6.1.

```

public abstract class TObject{

    private String uniqueName;

    public TObject(){
        uniqueName = generateUniqueName();
    }

    public abstract void _destructor();

    protected static final TFactory _factory(){
        return TGlobal.getFactory();
    }

    public String getUniqueName(){
        return uniqueName;
    }

    public String toString(){
        return uniqueName;
    }

    private String generateUniqueName(){
        return TGlobal.getFactory().createUniqueName();
    }
}

```

Figure 7.2: The *EDUT* TObject class generated from a class diagram.

5. For each class, *C1*, in a *DUT*, a collection class, *C1Collection* is generated in the *EDUT* as shown in lines 11-24 in Figure 7.4. Every collection class, *C1Collection*, is a sub-class of a class named *TObjectCollection* (which is shown in lines 1-10 in Figure 7.4). An instance of *C1Collection* maintains an ordered set of instances of *C1*. The *C1Collection* class has methods, *_add(C1)* and *_remove(C1)*, that add and remove instances of *C1* to and from the collection. *C1Collection* also has the method, *_getAt(int i)*, that re-

```

[1] public class C1 extends TObject {
[2]     //Begin attribute List
[3]     private T A;
[4]     private T _get_A() {
[5]         return iD;
[6]     }
[7]     private void _set_A(T A) {
[8]         this.A = A;
[9]     }
[10]
[11]     ...
[12]     //Associations here
[13]     public C2Collection c2 = new C2Collection();

[14]     //Destructor here
[15]     public void _destruction(){
[16]         for (int i = 0; i < c2._getTotal(); i++) {
[17]             c2._getAt(i).c1.remove(this);
[18]         }
[19]         ...
[20]     }

[21]     //Operations here
[22]     public Return Type Opt(p1:T1, p2:T2, ..., pn:Tn){
[23]         ... //Method body is generated from JAL
[24]     }

```

Figure 7.3: The EDUT C1 class.

turns an instance of C1 at the index *i* in the collection. C1 inherits from TObjectCollection the method `_getTotal()`, which returns the total number of instances in the collection.

6. Suppose that a UML association, *C1C2*, associates two classes, C1 and C2. Suppose also that *C1C2* connects to C2 via an association end, *c2* (as shown is Figure 7.1. The association end is transformed into the Java state variable `C2Collection c2` in the Java class C1, as shown in line 13 in Figure 7.3.

```
[1] public abstract class TObjectCollection{
[2]     protected Vector set;
[3]
[4]     public TObjectCollection(){
[5]         set = new Vector();
[6]     }

[7]     public int _getTotal(){
[8]         return set.size();
[9]     }

[10] }

[11] public class C1Collection extends TObjectCollection{
[12]     public C1Collection( ){
[13]         super();
[14]     }

[15]     public C1 _getAt(int i){
[16]         return (Product) ( set.elementAt(i) );
[17]     }

[18]     public void add(C1 o){
[19]         set.add(o);
[20]     }

[21]     public void remove(C1 o){
[22]         set.remove(o);
[23]     }
[24] }
```

Figure 7.4: The EDUT C1Collection class.

7. For each UML class, C1, a destructor method is created, providing a mechanism to destroy instances of C1 (as shown in lines 14-18 in Figure 7.3). The destructor method searches through every object that refers to the deleted instance, and deletes the reference. For example, lines 16, 17, and 18 are generated so that when they are executed, all references from C2 instances to the deleted object of

C are removed. When all references to a Java object, *o*, of type **C1** are removed, the Java garbage collector automatically deletes *o*.

7.1.2 Generating the TFactory class

The **TFactory** class handles low level tasks such as maintaining references between objects (since Java does not have link or association concepts, a UML link is represented indirectly using Java references), checking association cardinalities, removing Java references when an object is destroyed, and processing Java exceptions.

Figure 7.5 shows the **TFactory** class that is generated from the class diagram in Figure 7.1, using the following rules:

1. For each *DUT* model, create a Java class **TFactory**.
2. For each UML class, **C1** in the *DUT*, create a method, `_create_object_C1()`, in the **TFactory** class, as shown in lines 3-6 in Figure 7.5.
3. For each UML association, **C1C2**, between two classes, **C1** and **C2**, create two methods, `_create_link_C1C2()` and `_delete_link_C1C2()`, as shown in lines 12-15 and 16-19 in Figure 7.5, respectively. These methods create and destroy an instance of the association, **C1C2**. The variable `firstEnd.c2` refers to the attribute **C1.c2** as shown in line 13 in Figure 7.5. Similarly, the variable `secondEnd.c1` refers to the attribute **C1.c2**.
4. Create a destructor method, `_delete_object()`, to destroy any instance of *EDUT* classes that represent *DUT* classes. For example, the destructor method shown in lines 21-26 in Figure 7.5 can delete instances of **C1** and **C2**, which are sub-classes of **TObject**.

7.1.3 Generating *EDUT* method bodies from JAL specifications

We now describe the rules to transform JAL specifications into Java method bodies. The rules are illustrated using Figure 7.6, which shows the *EDUT* method

```

[1] public class TFactory{
[2]     //Object Creators
[3]     public C1 _create_object_C1(){
[4]         C1 o = new C1();
[5]         return o;
[6]     }

[7]     public C2 _create_object_C2(){
[8]         C2 o = new C2();
[9]         return o;
[10]    }

[11]    //Link Creators and Destructors
[12]    public void _create_link_C1C2(C1 firstEnd, C2 secondEnd){
[13]        firstEnd.c2.add(secondEnd);
[14]        secondEnd.c1.add(firstEnd);
[15]    }

[16]    public void _delete_link_C1C2(C1 firstEnd, C2 secondEnd){
[17]        firstEnd.c2.remove(secondEnd);
[18]        secondEnd.c1.remove(firstEnd);
[19]    }

[20]    //Object Destructor
[21]    public void _delete_object( TObject o){
[25]        o._destructor();
[26]    }
[27] }

```

Figure 7.5: A TFactory class.

ProductCatalog:addProduct(int cID, int pID). The method body is generated from the JAL specification shown in Figure 6.1.

1. JAL *Call Operation*, *Add Variable Value*, and *Return* statements, as well as *Read Variable*, *Calculation* expressions are copied into *EDUT* method bodies, because their syntax is the same as Java method invocation, assignment, and return statements, and variable and calculation expressions, respectively:

- JAL *Call Operation*, *Write Variable*, and *Return* statements become Java method invocation, assignment, and return statements, respectively. For example, lines 2 and 8 in Figure 6.1 are copied into lines 3 and 9 in Figure 7.6.
 - JAL *Calculation* and *Read Variable* expressions become calculation and variable expressions in Java.
2. JAL *Write Attribute* statements and *Read Attribute* expressions, which access the value of an attribute, **A** are transformed into invocation of the methods `_set_A()` and `_get_A()` that are generated from the attribute, **A**. For example, line 6 in Figure 6.1 is copied into line 7 in Figure 7.6.
 3. *Read Link* expressions are transformed into invocations of methods, `_getTotal()` and `_getAt()`.
 4. *Create Object* and *Create Link* expressions as well as *Destroy Object* and *Destroy Link* statements are transformed into appropriate invocations of the methods in **TFactory**, as shown in Table 7.1. For example, lines 5 and 7 in Figure 6.1 become lines 6 and 8 in Figure 7.6.

Table 7.1: Rules to transform JAL creation expressions and destruction statements into Java

JAL	Java
<code>_create_object_classname()</code>	<code>_factory()._create_object_classname()</code>
<code>_delete_object(objExpression)</code>	<code>_factory()._delete_object(objExpression)</code>
<code>_create_link_AssociationId(obj1Id, obj2Id)</code>	<code>_factory()._create_link_AssociationId(obj1Id, obj2Id)</code>
<code>_delete_link_AssociationId(obj1Id, obj2Id)</code>	<code>_factory()._delete_link_AssociationId(obj1Id, obj2Id)</code>

5. JAL local variable declarations have the same syntax as Java variable declarations, hence JAL local variable declarations are copied directly into the Java code. For example, line 1 in Figure 6.1 is copied into line 2 in Figure 7.6.

6. Because JAL condition and iteration structures are the same as Java conditions (if ... then ... else ...) and while loop structures (while ...) respectively, they are copied into *EDUT*. For example, lines 3 and 10 in Figure 6.1 are copied into lines 4 and 11 in Figure 7.6.

```
[1] private boolean addProduct( int cID, int pID ){
[2]     Category ctg ;
[3]     ctg = this.findCategory(cID) ;
[4]     if (ctg!=null){
[5]         Product prd;
[6]         prd = _factory()._create_object_product();
[7]         prd.setID(pID);
[8]         this._factory()._create_link_product_category(ctg,prd) ;
[9]         return true;
[10]    }
[11]    else {
[12]        return false;
[13]    }
[14] }
```

Figure 7.6: EDUT generated from addProduct JAL Specification

7.2 Generating the *TDUT*

Figure 7.7 shows the packages that are required during test execution. The packages contain the generated *TDUT* package, and some associated packages and classes (e.g., **USE** and **Use Interface**). The **USE Interface** package provides a mechanism for *TDUT* to communicate with the **USE** tool, which is represented as the **USE** package in Figure 7.7.

The *TDUT* package shown in the figure is generated from the *DUT* whose class diagram is shown in Figure 7.1. The *TDUT* package contains a package called **Test**

specified in a class diagram. To facilitate its use, test scaffolding code is added to the *EDUT* to perform the following functions:

1. Inform USE about any changes in the state of the simulated system.
2. When there is an operation call action, invoke USE to check the pre-condition.
3. When there is a return action, invoke USE to check the post-condition.
4. At the end of the test, invoke USE to check the validity of the configuration.

The last check is performed using the JUnit assertion mechanism. JUnit is a framework to write repeatable test inputs for Java code testing [20]. The following subsections describe in detail how the *TDUT* is generated.

7.2.1 Generating code to check the initialization of variables

In Java programs, using the value of a local variable before it is initialized is considered an error, which is reported by Java compilers during compilation. In our approach, JAL local variables are transformed into Java local variables. Therefore, we can utilize Java compilers to check for the initialization of these variables. This type of design faults can thus be detected at compile time. However, Java state variables can be set in one method and be used in another, and the order in which each method is called can only be determined during runtime. Thus, Java compilers cannot check for the initialization of state variables, which are used to represent UML attributes in our approach. We need to insert code in the *EDUT* to check for the initialization of attributes.

Figure 7.8 shows the *TDUT* code that is generated for a UML attribute, A (shown in Figure 7.1). This code is the result of adding the following lines to the corresponding *EDUT* code, which is shown in lines 3-9 in Figure 7.3:

1. Line 3a defines a boolean flag that is initially set to false indicating that the attribute is not initialized.

```
[3]  private T A;
[3a] private boolean _flagA = false;
[4]  private T _get_A() {
[4a]      if (_flagID == false) {
[4b]          reportError("C1::A is not initialized.");
[4c]      }
[5]      return iD;
[6]  }
[7]  private void _set_A(T A) {
[8]      this.A = A;
[8a]      _flagA = true;
[9]  }
```

Figure 7.8: The TDUT code generated from an attribute.

2. Lines 4a, 4b, and 4c are added to the method `_get_A()`, which returns the value of `A`. These lines check the value of the flag. If it is false, the *TDUT* reports that the system under test attempts to use the value of `A` before initializing it.
3. Line 8a is added to the method `_set_A()`, which assigns a value to `A`. This line sets the flag to *TRUE* to mark that `A` has been initialized.

7.2.2 Generating code to check for existence of target objects

We use Java's "NullPointerException" to check for existence of target objects. When a Java program executes, if the target object of a method invocation does not exist, the Java Virtual Machine raises a "NullPointerException". Since method invocations are used to represent operation calls in our approach, we catch these exceptions and report it to testers when the target object of an operation call does not exist.

7.2.3 Generating code to validate pre- and post-conditions

In order to validate pre- and post-conditions and object configurations, USE maintains information about the current object configuration. We first insert code into the

EDUT to notify USE whenever there is a change in the object configuration. Changes in the configuration include creation and deletion of objects and links, as well as modifications to attribute values. The new code is inserted as follows:

1. Code is inserted into the **TFactory** class to notify USE about the creation and deletion of objects and links. Figure 7.9 shows the **TFactory** class in the *TDUT* that is generated from the class diagram shown in Figure 7.1. Segments of code in lines 4a-4c, 12a-12d, 16a-16d, and 21a-21c of Figure 7.9 are inserted to notify USE about the creation of an object, **C1**, the creation of a link, **C1C2**, the deletion of a link, **C1C2**, and the deletion of instances, respectively. Figure 7.9 shows the **TFactory** class after the code is inserted.
2. The method `_set_A()` (as shown in lines 7-9 in Figure 7.8) that is used to set values to the attribute **A** of a class, **C1**, is transformed into the code shown in Figure 7.10. Lines 7a, 7b, 7c, and 7d are added to notify USE about the new value of attribute **A**.

During execution, the *TDUT* requests USE to validate the pre- and post-conditions before and after an operation is called. To perform that task, code that represents operations in *EDUT* is transformed into *TDUT* as follow:

1. For each *DUT* operation, `Opt(...):returnType`, in a class, **C1**, create a method `C1::_Opt(...):returnType` in the *TDUT*.
2. Copy the body of *EDUT* method `C1::Opt(...):returnType` into the body of *TDUT* method `C1::_Opt(...):returnType`.
3. Insert code into the body of *TDUT* method `C1::_Opt(...):returnType` so that it first requests USE to validate the pre-condition, then invokes the method `C1::_Opt(...):returnType`, and finally requests USE to validate the post-condition.

```

[1] public class TFactory{
[2]     //Object Creators
[3]     public C1 _create_object_C1(){
[4]         C1 o = new C1();
[4a]         try{
[4b]             use.addObject("C1", o.getUniqueName());
[4c]         }catch(Exception e){System.err.println(e.getMessage());}
[5]         return o;
[6]     }
[7]     public C2 _create_object_C2(){
[8]         C2 o = new C2();
[8a]         try{
[8b]             use.addObject("C2", o.getUniqueName());
[8c]         }catch(Exception e){System.err.println(e.getMessage());}
[9]         return o;
[10]    }
[11]    //Link Creators and Destructors
[12]    public void _create_link_C1C2(C1 firstEnd, C2 secondEnd){
[12a]        try{
[12b]            use.addLink("C1C2", firstEnd.getUniqueName(),
[12c]                        secondEnd.getUniqueName());
[12d]        }catch(Exception e){System.err.println(e.getMessage());}
[13]        firstEnd.c2.add(secondEnd);
[14]        secondEnd.c1.add(firstEnd);
[15]    }
[16]    public void _delete_link_C1C2(C1 firstEnd, C2 secondEnd){
[16a]        try{
[16b]            use.deleteLink("C1C2", firstEnd.getUniqueName(),
[16c]                           secondEnd.getUniqueName());
[16d]        }catch(Exception e){System.err.println(e.getMessage());}
[17]        firstEnd.c2.remove(secondEnd);
[18]        secondEnd.c1.remove(firstEnd);
[19]    }
[20]    //Object Destructor
[21]    public void _delete_object( TObject o){
[21a]        try{
[21b]            use.destroyObject( o.getUniqueName());
[21c]        }catch(Exception e){/*System.out.println(e.getMessage());*/}
[25]        o._destructor();
[26]    }
[27] }

```

Figure 7.9: The TFactory class with code to interact with USE.

```

[7] private void _set_A(T A) {
[7a]     try {
[7b]         use.setAttributeValue(getUniqueName(), "A",
[7c]                                 String.valueOf(A));
[7d]     } catch(Exception e){System.err.println(e.getMessage());}
[8]     this.A = A;
[8a]     _flagA = true;
[9] }

```

Figure 7.10: The `_set_A()` method that has code inserted to interact with USE.

Following these steps, the *EDUT* shown in Figure 7.6 is transformed into the *TDUT* code shown in Figure 7.11. In Figure 7.11, the method `_addProduct()`, which is generated from the JAL specification, is called from `addProduct()` (see line 9). Lines 2-8 and 10-15 are inserted to check the pre- and post-conditions of the `addProduct` operation, respectively. The body of the method `_addProduct` (lines 19-30) in Figure 7.11 is copied from the body of method `addProduct` (lines 2-13) in Figure 7.6.

7.2.4 Generating code for automation of test execution

The **Test Driver** package, which is added to automate the test execution, is described in Kawane's Masters Thesis [32]. The package provides a mechanism so that test inputs can be applied to the *TDUT*. The package also provides a mechanism for *TDUT* to communicate with a GUI to report failures.

The package has an interface, *TestObserver*, that allows the *TDUT* to report failures. The package also has an abstract class, *ModelTestCase*, that represents the test cases. It extends the JUnit **TestCase** class to support a test environment for testing UML designs by providing a base class for the model test drivers, a graphical user interface for displaying progress and results of test execution, and an assertion function, `assertConformance()`, that delegates the validation of object configurations to

```

[1] public boolean addProduct( int cID, int pID ){
[2]     try{
[3]         use.optEnter("openter " + getUniqueName() + " addProduct(" +
[4]             String.valueOf(cID)+ "," + String.valueOf(pID) + ")" );
[5]     }
[6]     catch(Exception e){
[7]         System.out.println(e.getMessage());
[8]     }
[9]     boolean _ret = _addProduct(cID,pID);
[10]    try{
[11]        use.optExit( String.valueOf( _ret ));
[12]    }
[13]    catch(Exception e){
[14]        System.out.println(e.getMessage());
[15]    }
[16]    return _ret;
[17] } // EDUT code for addProduct() follows
[18] private boolean _addProduct( int cID, int pID ){
[19]     Category ctg ;
[20]     ctg = this.findCategory(cID) ;
[21]     if (ctg!=null){
[22]         Product prd;
[23]         prd = _factory()._create_object_product();
[24]         prd.setID(pID);
[25]         this._factory()._create_link_product_category(ctg,prd) ;
[26]         return true;
[27]     }
[28]     else {
[29]         return false;
[30]     }
[31] }

```

Figure 7.11: EDUT generated from addProduct JAL Specification

the USE tool. *ModelTestCase* also contains the set of assertion functions provided by the JUnit framework, such as `assertTrue` or `assertFalse`.

7.3 Executing Tests

```
Class TestDriverImpl extends ModelTestCase{
    void testOne (){
        //Create start configuration
        ProductCatalog pc = factory._createProductCatalog();
        Category c_0 = factory._createCategory();
        Product p_0 = factory._createProduct();
        Product p_1 = factory._createProduct();
        Product p_2 = factory._createProduct();
        factory._create_link_productcatalog_category(pc, c_0);
        factory._create_link_product_category(p_0, c_0);
        factory._create_link_product_category(p_1, c_0);
        factory._create_link_product_category(p_2, c_0);
        c_0.setID(3);
        // Send test signal
        pc.addProduct(0, 0);
        // check conformance
        AssertConformance();
    }
}
```

Figure 7.12: Sample Test Case.

To specify a set of test cases, a tester creates a concrete class (e.g., *TestDriverImpl*) that is a subclass of *ModelTestCase*. For each test case, the tester specifies a test method with the prefix *test* (e.g., `testOne()`). The method body has three parts: a prefix to create the start configuration, a sequence of system operation calls, and assertion statements. The prefix contains a series of **TFactory** method invocations to instantiate objects and links between them. The prefix may also contain a few method invocations to set the object attributes. The sequence of system operation calls in the method body of a test method represents the sequence of system events in the corresponding test case. The assertion statements correspond to the invocation of assertion functions (e.g., `assertConformance()`).

Figure 7.12 shows the example of a test method, `testOne()`, that a tester provides as a test input. The prefix part of the method creates a start configuration (shown in Figure 5.7) with an instance of *ProductCatalog*, an instance of *Category*, three instances of *Product*, and the appropriate links between them. The prefix is followed by the operation call, `pc.addProduct(0, 0)` that initiates the testing of the path, *0-1-2-3-5-10*.

When a tester executes the test cases inside the class, *TestCaseImpl*, *UMLAnT* automatically invokes every method that has the prefix *test*. The failures are detected by USE or *UMLAnT* and reported through the interface, *TestObserver*.

Chapter 8

Tool Support

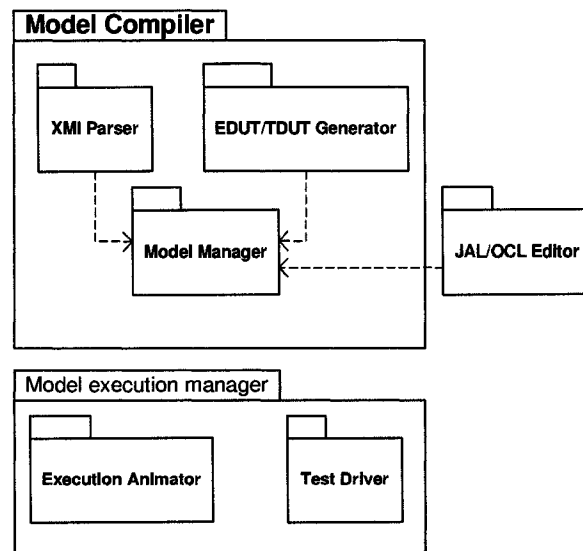


Figure 8.1: UMLAnT Architecture.

We developed a prototype tool called the “UML Animator and Tester” (*UMLAnT*) that automates our test execution and animation approach. The tool is an Eclipse plugin. Figure 8.1 shows the architecture of *UMLAnT*. The subsystems are as follows:

1. The *JAL/OCL Editor* is used to specify the model under test (*DUT*).
2. The *XMI Parser* parses the class diagrams saved in the XMI format.
3. The *EDUT/TDUT Generator* generates the testable form of the *DUT*.

4. The *Model Manager* maintains instances of the UML metamodel (i.e. the models under test).
5. The *Test Driver* executes tests and reports failures.
6. The *Execution Animator* helps visualize the execution.

8.1 Model specification

The Eclipse Modeling Framework (EMF) and Omondo EclipseUML plugins are used to draw and specify the *DUT*. An XMI Parser is used to parse the models and generate an instance of the UML metamodel inside the Model Manager.

Figure 8.2 shows a subset of the design class diagram of the Model Manager subsystem (shown in Figure 8.1). This class diagram contains UML meta-classes that represents elements in a UML class diagram. The “**ClassInfo**” class represents the UML meta-class, “*Class*”. A **ClassInfo** can have zero or many **Operations**, **Attributes**, and **Associations**. The current version of *UMLAnT* supports only binary associations, and thus each **Association** has exactly two **AssociationEnd**.

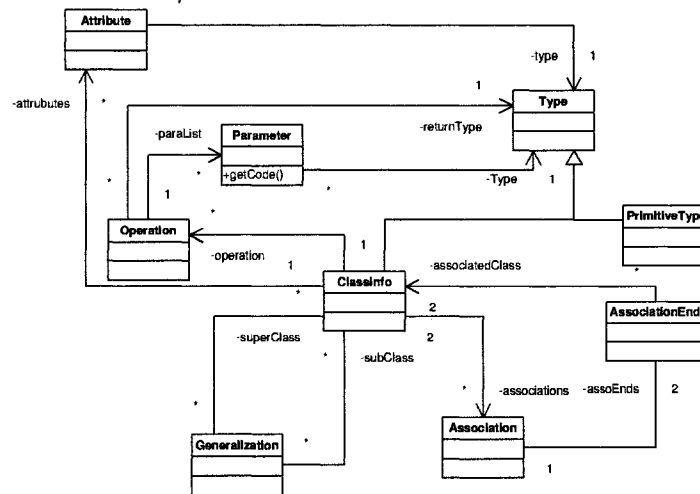


Figure 8.2: The Design Class Diagram of the *Model Management* Sub-System.

Figure 8.3 shows the model input screen of *UMLAnT* with an example specification of the operation `ProductCatalog::addCategory`, presented by the editor in the bottom right of the figure.

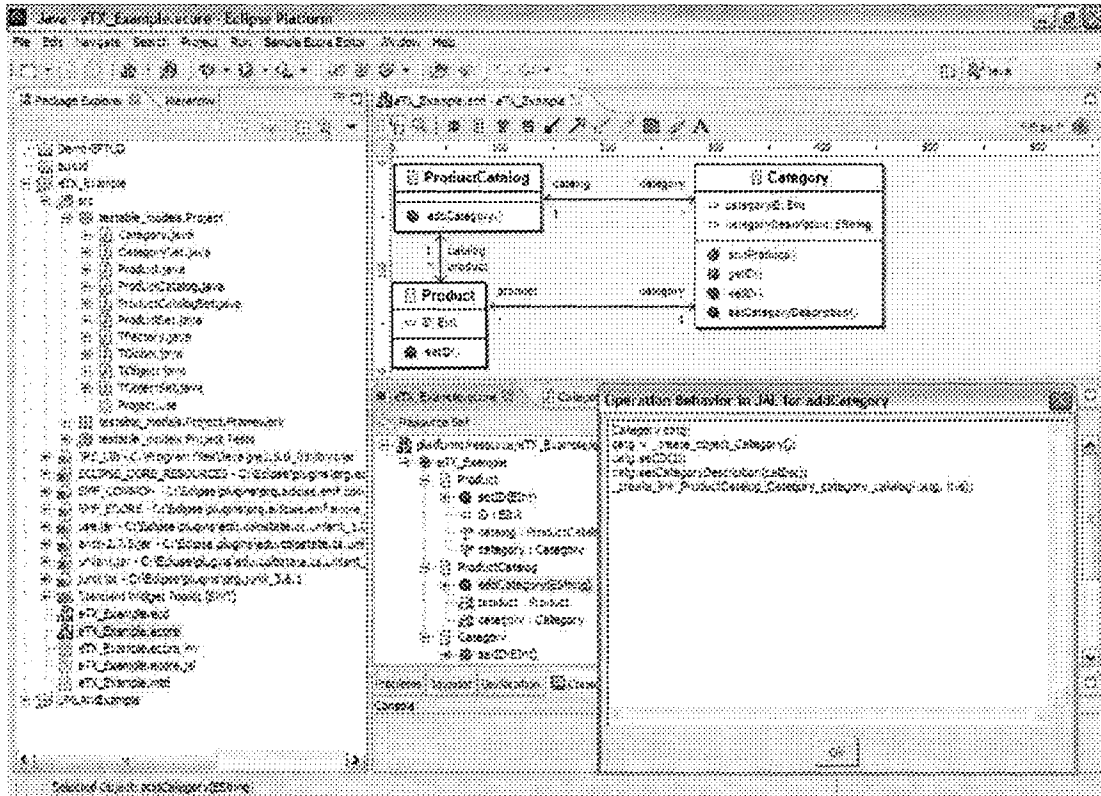


Figure 8.3: UMLAnT Input Screen.

8.2 Generation of the testable form

The *EDUT/TDUT Generator* uses an implementation of the Visitor design pattern [19] to extract information about the *DUT* from the *Model Manager*. The *EDUT/TDUT Generator* then applies the rules described in Chapter 7 to transform the *DUT* into the *EDUT* and adds test scaffolding to generate the *TDUT*. The generated *TDUT* utilizes the *Model Execution Manager* to execute tests and detect failures.

Figure 8.4 shows a set of classes of the *TDUT/EDUT Generator* subsystem. The **CollectionClassGenerator** class transforms each UML class into a *Collection* class. The **FactoryGenerator** creates a **TFactory** class for each design model under test. The **InfrastructureGenerator** creates **TObject** and **TObjectSet** classes. The **TObject** and **TObjectSet** classes are the same for every model under test.

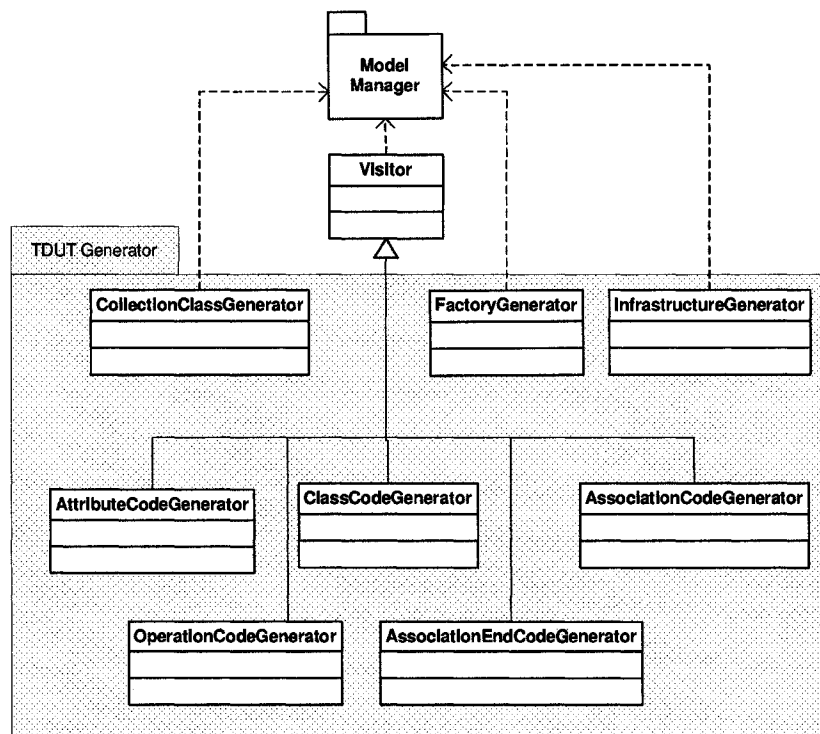


Figure 8.4: The Design Class Diagram of the *EDUT/TDUT Generator* Sub-System.

The **ClassCodeGenerator**, **AttributeCodeGenerator**, **AssociationCodeGenerator**, **OperationCodeGenerator**, and **AssociationEndCodeGenerator** classes are used to transform elements of a UML class into Java. A UML model is traversed several times when the *TDUT* and the *USE* formats of the model are generated. Information in each class in the *Model Manager* package is manipulated differently each time different forms are generated. Hence, we implemented these transformations using the *Visitor* pattern [19].

All classes shown in Figure 8.2 play the roles of *Concrete Elements* in the *Visitor* pattern. They are sub-classes of the *Abstract Element* class, **Element**, shown in Figure 8.5. In Figure 8.4, **Visitor** plays the role of the *Abstract Visitor*, and its sub-classes play the roles of *Concrete Visitors*. According to the visitor pattern, the transformation of an entire class diagram can be implemented in a single *Concrete Visitor* class. However, in order to increase the cohesion in the system, we implemented the transformation of classes, attributes, operations, associations, and association ends using separate classes. Thus, **ClassCodeGenerator**, **AttributeCodeGenerator**, **AssociationCodeGenerator**, **OperationCodeGenerator**, and **AssociationEndCodeGenerator** transform classes, attributes, associations, operations, and association ends, respectively, into code.

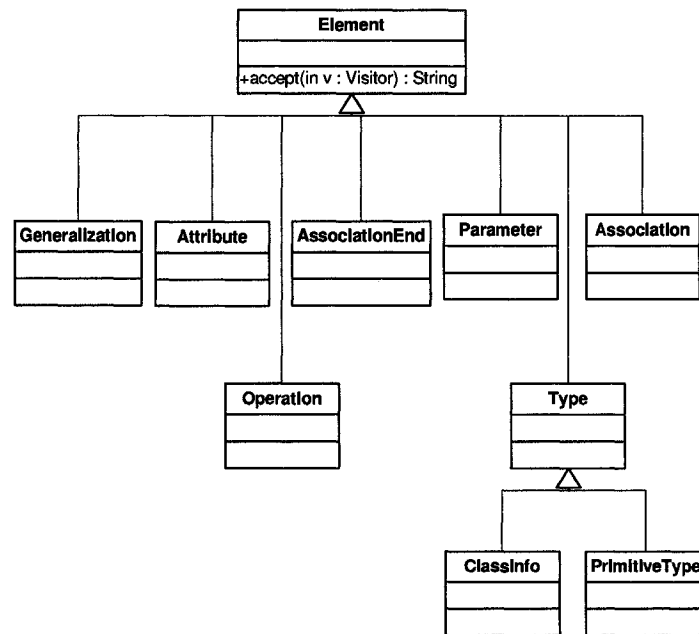


Figure 8.5: *UMLAnT* Classes that Play the Roles of *Elements* in the *Visitor* pattern.

Figure 8.6 shows the sequence diagram that describes the transformation of a UML class into *TDUT*. The **accept()** and **visit()** call operation messages are sent as specified in the *Visitor* pattern. As shown is this sequence diagram, when

a class is visited, all of its attributes, associations, and operations are also visited. **ClassCodeGenerator** utilizes the code generated by **AttributeCodeGenerator**, **OperationCodeGenerator**, and **AssociationCodeGenerator** to generate code for a class. The **AssociationCodeGenerator** utilizes the code generated by **AssociationEndCodeGenerator** to generate code from association. To avoid making the sequence diagram cluttered, the interaction between **AssociationCodeGenerator** and **AssociationEndCodeGenerator** is not shown.

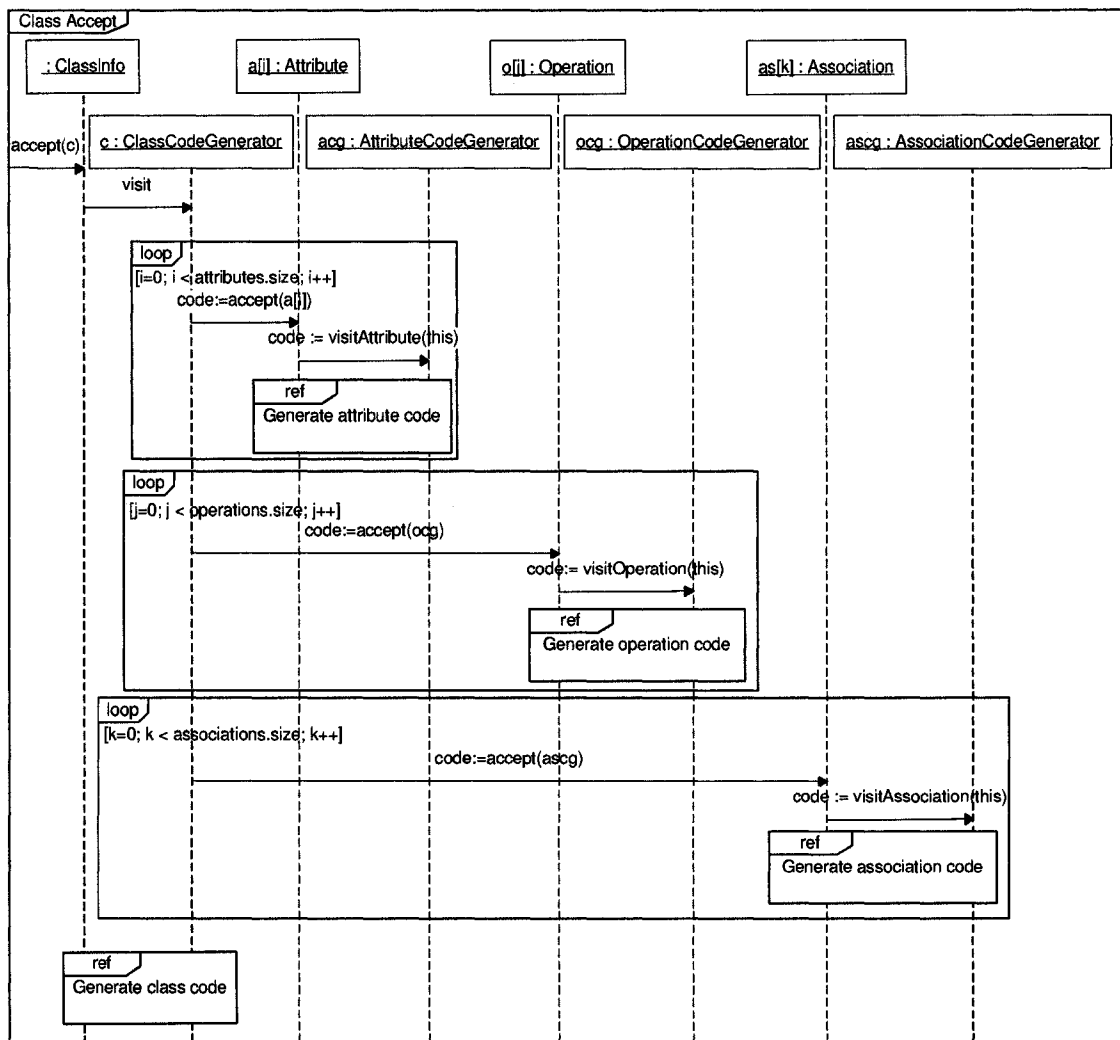


Figure 8.6: Sequence diagram for transforming a class into TDUT.

8.3 Test execution and failure reporting

Test inputs that are generated by the Alloy constraint solver are currently manually entered by testers into the tool in the form of JUnit tests, as described in the previous chapter. When a tester executes the test cases inside the class, *TestCaseImpl*, *UMLAnT* automatically invokes every method that has the prefix, *test*. The failures are detected by *USE* or *UMLAnT* and reported through the interface, **TestObserver**.

UMLAnT provides *USE* with pre- and post-conditions specified in OCL and requests *USE* to validate them for every operation before and after its execution, respectively. Also, after the execution of every system event in the test input, *UMLAnT* signals *USE* to check the object configuration against the class diagram constraints. Any failure detected by *USE* or *UMLAnT* is reported using the interface, **TestObserver**.

UMLAnT reports the following failures: “*OCL invariant checking failed*”, “*Post-condition failed*”, “*Message sent to null*”, “*Collection out of bounds*”, “*Conformance checking failed*”, and “*Oracle condition failed*”.

8.4 Model animation

During test execution, the result of every action performed by each object is recorded in a log file. The results recorded include (1) any changes to the configurations, and (2) messages that are exchanged between objects. The changes to the configuration include creation and deletion of objects and links and modification to attribute values. Messages that are exchanged between objects includes create and destroy object messages and method invocations.

When the execution terminates, the *Execution Animator* reads the log file and updates the sequence and object diagram views. Whenever the action involves changing an attribute value, or creation or deletion of a class or association instance, the object diagram view is updated. Whenever the action involves sending a message, or cre-

ation or deletion of an object, the sequence diagram view is changed. Figure 8.7 shows an example of sequence diagrams (parts (a) and (b)) and object diagrams (parts (c) and (d)) created by *UMLAnT* during the animation of test execution. Figure 8.7(d) shows the creation of the link between the instances, *tObj_1:ProductCatalog* and *tObj_2:Category* in Figure 8.7(c).

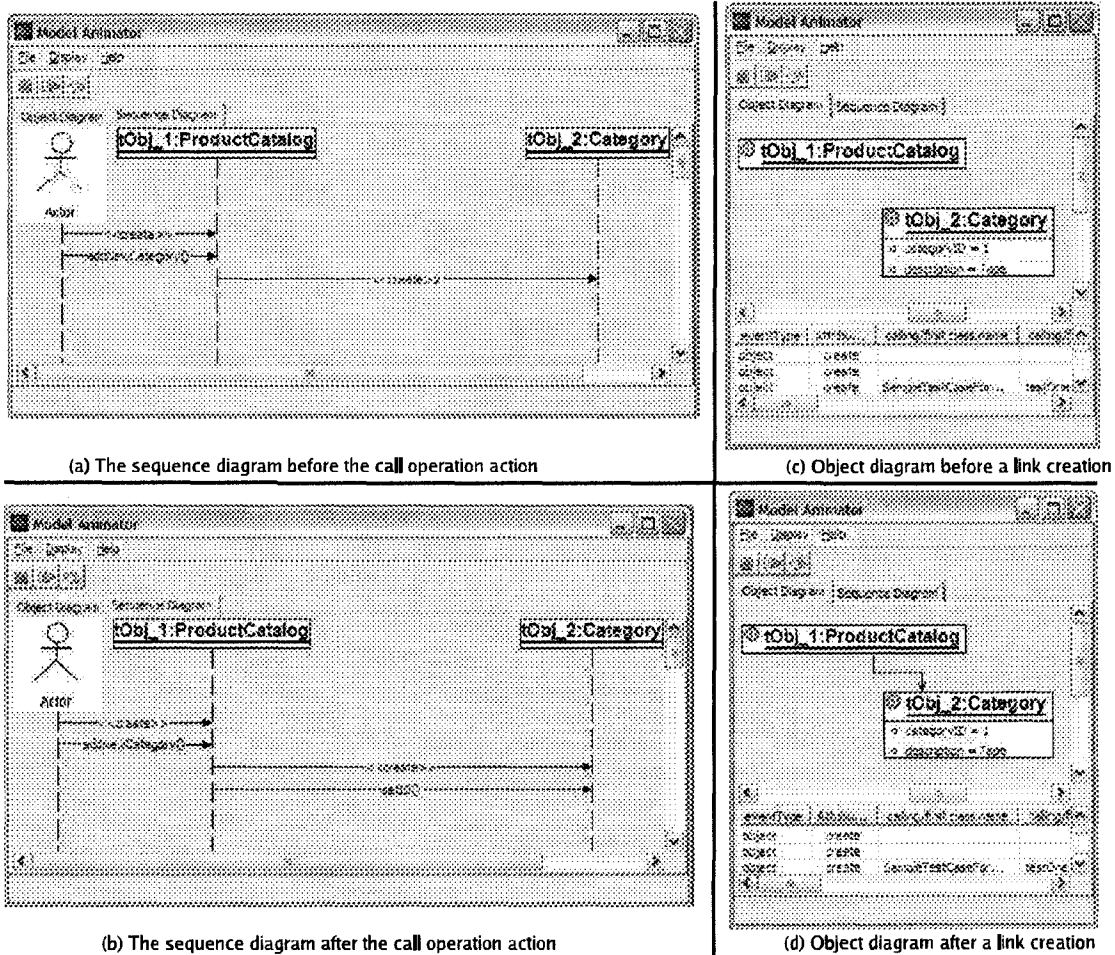


Figure 8.7: UMLAnT Animation Screen.

We can use the animation feature of *UMLAnT* to detect the existence of infinite loops. If we suspect such a problem when the execution of the faulty model does not terminate after a long time, we can terminate the execution manually and review the

generated run-time sequence diagram. We can inspect the sequence of actions that were repeatedly executed until we terminated the test execution.

8.5 Testing *UMLAnT*

We tested *UMLAnT* to evaluate the correctness of the transformation from *DUT* to Java. We used *UMLAnT* to generate executable Java code from a set of UML models that contain class diagrams and JAL segments. We carefully reviewed the input models before the transformation to ensure that they were syntactically correct and that the behavior described in the JAL segments was the intended one. We then checked if the generated programs exhibited correct behavior to assess the correctness of the transformation.

The input models were created so that they contained every type of construct in the input metamodel. In our case, the input metamodel is the subset of UML metamodel that describes concepts in class diagram views and action semantic views. Class diagrams used in our test cases contain classes, attributes, primitive types, operations with parameters, operations without parameters, generalizations, binary associations with one-to-one, one-to-many, and many-to-many cardinalities at two ends. JAL specifications in our test cases contain all primitive actions and control structures that are supported by our transformation approach. Furthermore, our input models also cover several combinations of the metamodel constructs. The input models cover attributes with all different primitive types: integer, floating, boolean, and string. The parameter types cover all primitive types and also class types. The input models also cover operations without any return types and operations with all the primitive types and class types.

To create an oracle, we leveraged the fact that the outputs of our transformation are executable. Therefore, we validated the output Java programs by testing them. The test inputs for the Java programs were derived manually from their design

specifications (i.e., the *DUT*). We made sure that the test inputs covered the design elements in the *DUT* using the criteria described in chapter 4. We then executed the programs with the generated inputs. In some cases, the generated programs could not be compiled, indicating that there were errors in the output. In some other cases, the programs compiled, but some of the test cases we ran against the generated Java programs failed, indicating that there were errors in the programs. Given that the input models were known to be correct, an error in the output meant that there was an error in the transformation. Using this technique we were able to detect and remove a number of errors in *UMLAnT*. This increased our confidence in the prototype implementation.

Chapter 9

Pilot Studies

We performed three pilot studies to (1) demonstrate the fault detection capability of the test inputs generated using our approach, and (2) explore the performance of our approach. We applied our design testing approach to an online shopping system (OSHOP), a UML model composition system (COMP), and a UML to VAG transformation subsystem (UML2VAG). The design models for three systems were created by teams of software engineering graduate students.

Table 9.1 shows the characteristics of the models under test. For each model, the columns show the name of the system, the number of classes in the class models, the number of sequence diagrams, the number of messages in each sequence diagram, and the number of activity diagrams, respectively. Each sequence diagram describes multiple scenarios and contains alternative and loop structures, and thus there is more than one possible execution path (e.g., see Figure 5.2(b)).

Table 9.1: Sizes of systems under test.

System	Number of classes	Number of sequence diagrams (SD)	Number of messages per SD	Number of activity diagrams
OSHOP	6	3	10, 11, 12	12
COMP	5	1	41	14
UML2VAG	17	2	28, 23	28

A set of fault types was compiled by identifying common faults that designers normally introduce while modeling behavior. The list of fault types was created based in part on studies of design models developed by students in our courses, and mutation analysis of UML designs [14]. Faults based on these types were seeded into the models. The following list shows the fault types and describes how faults of each type were seeded:

1. Missing actions (MA): Remove an action from an activity diagram.
2. Faulty order of actions (FOA): Change the location of an action in an activity diagram.
3. Faulty actions (FA): Replace an action by a new action in an activity diagram.
4. Restrict the scope of condition structures (SCS): Change the scope of a condition structure so that a few actions are incorrectly taken out of the structure.
5. Broaden the scope of condition structures (BCS): Change the scope of a condition structure so that a few actions are incorrectly included in the structure.
6. Faulty parameters (FP): Change the value of a parameter.
7. Missing branch of an alternative structure (MBA): Delete a branch of an alternative structure.
8. Faulty condition (FC): Modify a condition of an alternative or iterative structure.
9. Faulty OCL predicate (FOP): Modify an OCL statement, for example, replace an operator by another. The OCL statement can be an invariant, a pre-condition or a post-condition.
10. Faulty association end multiplicities (FAM): Modify the multiplicity value at an association end.
11. Wrong inheritance tree (WIT): Modify the inheritance tree so that one class becomes a sub-class of its sibling. In other words, replace a broad inheritance tree by a deep inheritance tree.

12. Wrong reference to Parent Class (WRP): Replace an association to a class by an association to a parent class.

Faults from each of the above types were seeded into the models one by one. Fault types 1–8 were used to seed faults into sequence diagrams and the rest were used to seed faults into class diagrams. Some faults resulted in the creation of equivalent designs and some others produced models that did not conform to the UML syntax. These models were discarded. We were left with 57 faulty models, which were given to a tester who did not know what faults were seeded.

9.1 Test input generation

The tester generated test inputs for each model using the approach described in this paper. The paths were always selected to cover the maximum number of uncovered elements. Also, the paths are selected so that the loops are iterated at most once. For each model, the tester generated three sets of test cases to satisfy the “*All Message Coverage (Mesg)*”, “*Condition Coverage (Cond)*”, and “*All Message Path Coverage (Path)*” criteria. The maximum number of instances of each class was initially set to 5 when the Alloy constraint solver was used.

Table 9.2 shows the number of paths required to cover each criterion for the models under test. When we generated input sets that covered the *Cond* criterion, the Alloy solver did not find solutions for 3 paths. After manually reviewing the path constraints, we concluded that 2 paths were infeasible; both required a boolean variable to be *TRUE* and *FALSE* at the same time. The third path was indeed feasible: when we increased the number of object for each class in Alloy into 6, Alloy provided a solution for the path. The infeasible paths were discarded and two other paths were selected that were feasible and also covered the remaining elements.

When generating test inputs that cover the *Path* criterion, we found 42 more infeasible paths. Among them, 12 paths belonged to OSHOP, 31 to COMP, and 1

Table 9.2: Number of test cases generated from the models to satisfy the criteria.

System	Mesg	Cond	Path
OSHO	3	6	8
COMP	3	5	8
UML2VAG	3	4	6

to UML2VAG. In our study, manually recognizing infeasible paths was trivial when the same boolean expressions were required to be *TRUE* and *FALSE* at the same time. In other cases, recognizing an infeasible path required more complex logical reasoning. For example, we recognized an infeasible path when it required all the following conditions to be true:

```

i = 0
i < m1.size
j = 0
!(j < m1.size)

```

Figure 9.1 represents the relationship between the length of a path and the size of its constraint in our study. Each constraint was written in conjunctive normal form. The length of a path is measured by the number of messages along the path. The size of a constraint is measured by the number of OCL conjuncts in that constraint. The figure shows that the size of a path constraint increased almost linearly with respect to the number of messages along the path in our case study. In general, the size of a path constraint depends on the number of messages in the path, the size of the post-condition of each operation that is called when the path is executed, and the size of each branching predicate along the path.

The number of test inputs generated using our approach does not depend on the size of the class diagrams. Rather, the number of test inputs generated from a sequence diagram increases when the total number of paths in the sequence diagram

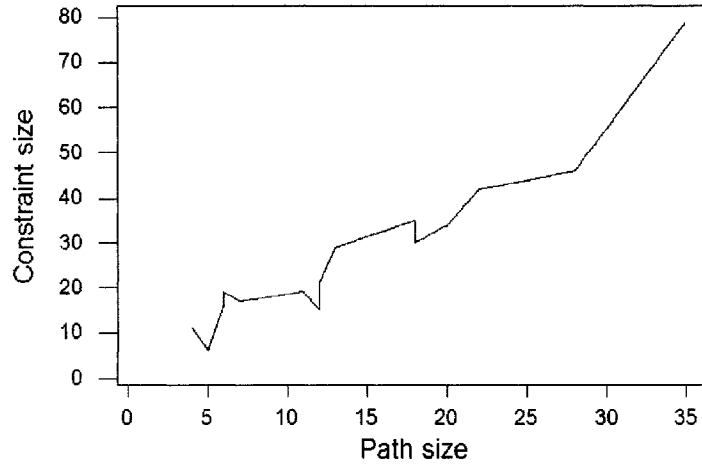


Figure 9.1: Relationship between Path Length and Constraint Size.

increases. This is due to the fact that the number of test inputs depends on the path selection step. During path selection, we only use the information regarding the ordering between the nodes and edges of a *VAG* without using the information stored inside each node. Moreover, the ordering of the nodes and edges in a *VAG* is obtained from the sequence diagram, not the class diagram.

The size of the start configuration for a test input, however, tends to depend on the both class and sequence diagrams under test. For each chosen test path in a sequence diagram, the number of sequence diagram participants that either send or receive one or more messages belonging to the path defines the number of objects that will be executed during testing. For example, when the path 0-1-2-3-5-10 in Figure 5.3 is executed, there are only two objects, *pc* and *c[i]* that actually participate in the execution. However, the start configuration usually needs to contain some additional objects to make the configuration conform to the class diagram. In the example in

Figure 5.3, the class diagram specifies that any instance of *ProductCatalog* must link to one instance of *Shop*. Thus, the configuration must contain an instance of *Shop*.

9.2 Test execution results

In our study, we considered the original model as the oracle. We first ran the generated test cases on the original model. At the end of each test execution, we obtained the final object configuration from *UMLAnT*. We specified a set of oracle conditions based the final configurations.

Table 9.3 summarizes the number of faults detected by each criterion. The first column of the table denotes the type of the seeded fault. The second column shows the number of faults that are seeded in the models under test for each fault type. The last three columns represent the number of faults for each fault type that are detected by tests that satisfy the *Mesg*, *Cond*, and *Path* criteria, respectively.

Out of 57 seeded faults, 51 (89.6%) were detected by test inputs satisfying the *Path* criterion. Among the 41 faults that were seeded into the activity diagrams, 38 (92.7%) were detected by these test inputs. There were six faults that were not detected by any test inputs used in the study.

Among the faults seeded in the class diagrams, three were not detected by test cases satisfying *Path*, four were not detected by test cases satisfying *Cond*, and five were not detected by test cases satisfying *Mesg* criteria. These faults occurred in cases where the faulty class diagram characterized a smaller set of configurations than the original class diagram. Since we seeded only one fault in each model, the behavioral diagrams (in this case, the activity and sequence diagrams) must be unchanged. The behavior described by these diagrams may require configurations that are not characterized by the faulty class diagram. The faults would be detected by *UMLAnT* if the test execution produced a configuration that did not conform to the faulty class diagram. Since the criteria used in our approach did not target class diagram

Table 9.3: Fault detection data.

Type of fault	Number of faults seeded	Number detected by <i>Mesg</i>	Number detected by <i>Cond</i>	Number detected by <i>Path</i>
MA	7	6	7	7
FOA	5	5	5	5
SCS	4	3	4	4
BCS	2	0	0	1
FA	7	6	6	6
FP	7	6	6	6
MBA	3	0	3	3
FC	6	3	5	6
INV	5	5	5	5
FAM	6	2	3	4
WIT	4	4	4	4
WRP	1	0	0	0
Total	57 (100%)	40 (70%)	48 (84.2%)	51 (89.6%)
Activity diagram faults	41 (100%)	29 (70.7%)	36 (87.8%)	38 (92.7%)
Class diagram faults	16 (100%)	11 (68.8%)	12 (75%)	13 (81.3%)

elements and thus did not require the tests to produce several different configurations, it is likely that a test input generated based on sequence diagram criteria may miss such a fault.

All three faults seeded in the activity diagrams that were not detected by any criterion were inside loop structures. These faults cause test failure only when the loops are executed at least twice. However, in our test selection approach, loops are executed only once.

Two of the faults were seeded into condition structures, causing the value of some variables to be set incorrectly only when some rare paths are executed. However, the test sets that satisfy *Mesg* and *Cond* criteria did not cause these paths to be executed and hence, did not find the faults. As expected, the test inputs that satisfy the *Path* criterion discovered these faults.

Seven faults in activity diagrams can only be detected by test cases that cause some conditions to evaluate to *FALSE*. In our study, the test sets generated based on the *Mesg* criterion always made these conditions evaluate to *TRUE*, since the *FALSE* branches are empty. Hence, test inputs generated using only *Mesg* criterion cannot detect these faults.

9.3 Discussion

The pilot study is a small scale study. The size of the models, the number of inserted faults, and the number of the original models used in the study were small. However, the studies indicate that the test inputs generated using our approach may be effective at finding a number of fault types and thus, motivate further studies on a larger scale.

Test inputs that are generated based on the sequence diagram based criteria can be used to reveal faults in behavioral diagrams more effectively than faults in structural diagrams. To target the faults in structural diagrams, one needs to force testing to cover several different set of configurations.

The techniques used to select test paths may affect test effectiveness. For example, our study shows that test sets that execute loops only once may be insufficient to find certain types of faults. To target such faults, one needs a test criterion that forces the test inputs to execute loops several times. For example, one can use Binder's iteration coverage criterion [7], which requires each loop to be executed zero, one, and a large number of times.

Chapter 10

Conclusions and Future Work

This chapter first summarizes the contribution of the dissertation. Open issues related to the contributions are then discussed. The chapter ends with directions for future work.

10.1 Summary of the contribution

This dissertation presents an approach to testing UML designs. UML designs are transformed into testable forms that include code for performing test execution and animation. We described a list of conditions that are checked during testing and a set of failure types that are detected by our approach. The approach also supports the animation of test execution. The ability to animate models can help one better understand modeled behavior. Novice and experienced developers can both benefit from the visualization of modeled behavior provided by model animators. Model animation gives quick visual feedback to novice modelers and thus, helps them identify improper use of modeling constructs. Experienced modelers can use model animation to understand designs created by other developers better and faster.

We introduced the JAL action language, which represents action semantics specified in UML 2.0 standard. The JAL syntax is similar to Java, hence developers who are familiar with Java can easily learn JAL.

This dissertation presents a systematic approach for generating test inputs from UML design models. We propose the use of a VAG to generate path constraints. VAG is a directed graph, hence existing graph-based approaches to selecting execution paths (such as [5, 39, 44, 47, 60]) can be applied to VAG. Moreover, the VAG symbolically represents object configuration constraints as sets and variables, and thus, can be used to generate test inputs for object oriented systems. Using the VAG makes it easier to generate path constraints, because the tester now needs only one representation instead of having to work with several different diagram views. While symbolic execution has been traditionally used on procedural programs where behaviors are described imperatively, we have applied it to UML models, where behaviors are specified declaratively in the operation pre- and post-conditions.

The dissertation presents a prototype tool, *UMLAnT*, an Eclipse plugin that automates the test execution and animation approach. *UMLAnT* is integrated with some widely used software development technologies, tools and languages, such as Eclipse, EMF, JUnit, UML, and Java, thereby enhancing its applicability.

The dissertation describes three pilot studies, where our testing approach is applied to test design models. Our approach detected about 90% of the faults seeded during the studies. The results support the view that testing design models can help detect design flaws, though more studies are needed to evaluate the approach.

10.2 Discussion

An issue to consider for any constraint-based testing approach is the applicability of the approach to solve path constraints for large models. The results of our pilot studies suggest that the size of a path constraint increases linearly with the size of the path. Besides, our studies also suggest that the time it takes the Alloy constraint solver to solve a path constraint does not depend significantly on the path length.

We noted that the bigger the size of a class diagram, the longer it takes Alloy to solve a constraint. However, if using the constraint based approach is costly, testers can combine it with cheaper approaches such as random testing. For example, one can use random approach to generate inputs that cover most of the conditions (or messages) in sequence diagrams, and use the constraint-based approach only to cover the remaining conditions (or messages).

Our test input generation approach is automatable. Rules exist for transforming UML diagrams into VAGs, transforming constraints into Alloy, and generating path constraints. The path selection can be fully automated if infeasible paths do not exist in the models. In the presence of infeasible paths, none of the existing path selection techniques can be fully automated, since the problem of determining whether there is an input that exercises a path is undecidable [5]. A number of researchers (e.g., [39]) suggested a semi-automated solution where paths that are automatically generated are presented to testers. The testers then determine whether a path is feasible or not. Given a finite range of the number of objects, the Alloy constraint solver can be used to determine if there is an input that exercises a path. In our approach, we can set a limit on the number of objects in Alloy to a reasonably large number and solve a path constraint. If Alloy reports that there is no solution within that limit, the tester can manually analyze the path and decide if the path should be rejected. Alternatively, a higher limit on the number of objects can be used.

A significant concern regarding UML action semantics is that the current semantics do not significantly raise the level of abstraction above that provided by programming languages [18, 37]. Our experience with using JAL to specify stand-alone non-distributed systems indicate that while JAL provides some abstraction over Java, it is not much higher. For example, the JAL segment in Figure 6.1 is similar to the generated Java code shown in Figure 7.6. The only significant additional Java code in this example is the code in the **TFactory** class, which is about 70 lines long.

With platform-independent models (PIM) [54] of distributed systems, however, using JAL instead of programming languages can produce models at a higher level of abstraction. The model in Figure 5.2 may be considered as a PIM of a distributed system, where *ProductCatalog* and *Product* are located in different machines communicating using the Java RMI [62] mechanism. At the PIM level, a modeler can still use the JAL segment in Figure 6.1 to specify the behavior of the operation, *ProductCatalog::addProduct()*. The generated platform-specific code, however, would be much more detailed than the Java code shown in Figure 7.6. For example, code will be needed to handle RMI exceptions and creation of objects at remote locations.

In Chapter 2 we state that our approach can be applied to completely specified UML design models, in which every operation is associated with an activity diagram (specifying using JAL) and all operation pre- and post- conditions are fully specified using OCL. Such a requirement may at first seem to add too much overhead to the software development processes; OCL and action languages are rarely used in current development approaches. However, the fact that we have developed an automatable technique that utilizes OCL invariants and pre- and post-conditions to find faults will motivate developers to specify these items, especially for safety critical software systems. Moreover, in MDE approaches, operations that are specified using action languages may be automatically transformed into code.

Sometimes the modeled system is too big and it is not feasible or desirable for developers to completely specify all the pre- and post-conditions and JAL specifications. Our approach may still be used to test the most critical scenarios described in a design model. Figure 10.1 describes a development process for such situations. First, developers create the design class diagram, which is the UML diagram that is most widely used. Developers use their domain knowledge to identify the system's most critical scenarios that must be tested. The developers use sequence diagrams to specify these scenarios. Next, developers provide pre- and post- conditions and

JAL specifications for each operation that is called in the critical sequence diagrams. These scenarios are tested using our approach.

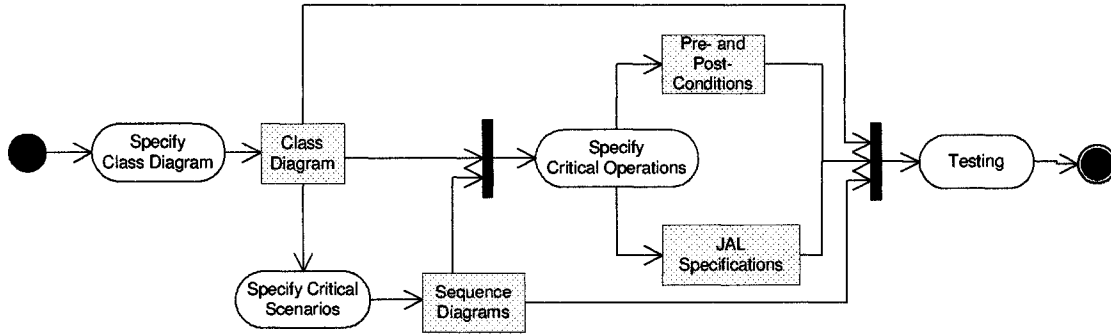


Figure 10.1: A modeling process that includes model testing.

10.3 Future work

Currently, JAL supports only synchronous action semantics. Future work can extend JAL to support asynchronous semantics. Techniques are needed to transform activity diagrams from graphical format to JAL format and vice versa.

We only use class and interaction diagrams to generate test input sets. Future work in generating test inputs includes using information from activity and use case models. Also, the approach can be extended to generate test inputs that satisfy class diagram based test adequacy criteria described in Andrews et al. [3].

Although our inputs are intended to test design models, they can be transformed into inputs that may be used to test the implementations as well. This will help validate the code against the models. The transformation of model test inputs to code test inputs can be investigated in future work.

In the current version of *UMLAnT*, animations are performed after test execution is complete. Future work in developing tools can include enhancements so that the animation is performed during test execution. *UMLAnT* can also be extended to support the transformation of UML diagrams into VAGs and VAGs into test inputs.

Appendix A

Java like Action Language Specification, Version 1.1

A.1 Introduction

JAL is an action language that supports the action semantics described in the UML 2.0 specification. JAL can be used to specify actions within the context of a UML activity diagram. JAL supports specification of a sequence of actions performed by a class instance during the execution of an operation call. It provides access to the data described in class diagrams and the data supplied by signals that initiate the specified sequence of actions. The current version of JAL only supports synchronous operation invocation.

A.2 Grammar

The original version of the grammar can be found in Kawane's Masters Thesis [32].

We modified the grammar so that it uses the standard BNF operators:

1. Terminal symbols are represented in bold face.
2. X^* represents zero or more repetitions of X .
3. $X \mid Y$ represents a choice between X or Y .
4. $[X]$ represents an optional use of X . However, in the JAL grammar below, the characters '[' and ']' also appear as terminal symbols and are used to declare

sets. To differentiate such occurrences, we use quotes around them (“[” and “]”).

The JAL grammar is given below:

1. `jal_segment := jal_statement*`
2. `jal_statement := jal_expression; | jal_delete_statement
| jal_creating_link_statement | jal_delete_link_statement | statement`
3. `jal_expression := jal_read_attribute_expression
| jal_modify_attribute_expression
| jal_creating_expression | jal_read_association_expression
| jal_count_association_expression | expression`
4. `variable_declaration := type variable_declarator ;`
5. `variable_declarator := identifier | identifier “[” “]”`
6. `jal_read_attribute_expression := _get_identifier ()`
7. `jal_modify_attribute_expression := _set_identifier(jal_expression)`
8. `jal_creating_expression := _create_object_identifier`
9. `jal_delete_statement := _delete_object(identifier) ;`
10. `jal_creating_link_statement := _create_link_identifier (identifier ,
identifier) ;`
11. `jal_delete_link_statement := _delete_link_identifier (identifier ,
identifier) ;`
12. `jal_read_association_expression := identifier._get_At (identifier)`
13. `jal_count_association_expression := identifier._get_Total ()`
14. `if_statement := if (expression) {jal_segment} [else {jal_segment}]`
15. `while_statement := while (expression) {jal_segment}`

The non-terminal symbols, `type`, `identifier`, `arg_list`, `expression`, and `statement`, have the same interpretations as those for non-terminal symbols used in the Java Language Specification [30].

A.3 JAL syntax

A JAL segment specifies the sequence of actions executed within an operation. It consists of a number of JAL statements. A JAL statement can be a simple statement (e.g., an operation call action), a loop, or condition statements. A simple statement can be an expression, a single statement, or a compound statement. A simple statement ends with a semicolon (“;”). An expression represents an action that returns a value. A simple statement represents an atomic action that does not return a value. A compound statement represents a combination of atomic actions.

A.3.1 Identifiers

JAL statements are composed of keywords, logical and arithmetic operators, and identifiers. Identifiers can be defined in class diagrams (e.g., names of classes, attributes, associations, operations, and operation parameters) or in JAL segments as local variables. JAL identifiers must conform to the following rules:

- Identifiers are case sensitive.
- Identifiers may only contain the characters $[a - z]$, $[A - Z]$, and $[0 - 9]$.
- Identifiers must not start with a numeric character $[0-9]$.
- Identifiers must not be the same as the keywords.

A variable can have a primitive type or be an object handle. A JAL variable needs to be declared before being used.

A.3.2 Keywords

JAL has the keywords `if`, `when`, `return`, `_get_`, `_set_`, `_create_object_`, `_delete_object_`, `_create_link_`, `_delete_link_`, `_get_At`, `_get_Total`, `_add`, and `_remove`.

A.3.3 Primitive and Pre-defined Types

JAL supports the primitive types *int*, *float*, *String*, and *boolean*. JAL also defines the *Collection* type that only contains objects of the same type.

A.3.4 Condition statements

Condition statements in the JAL have the following syntax:

```
if ( boolean_expression ) {  
    <SEQUENCE_OF_STATEMENTS_1>  
}  
[else {  
    <SEQUENCE_OF_STATEMENTS_2>  
}]
```

If `boolean_expression` evaluates to true, `<SEQUENCE_OF_STATEMENTS_1>` is executed, otherwise, `<SEQUENCE_OF_STATEMENTS_2>` is executed. The `else` branch can be omitted if `<SEQUENCE_OF_STATEMENTS_2>` is empty.

A.3.5 Loop statements

Loop statements in the JAL have the following syntax:

```
while( boolean_expression ){  
    <SEQUENCE_OF_STATEMENTS>  
}
```

The body of the loop, `<SEQUENCE_OF_STATEMENTS>`, is executed when the loop guard, `boolean_expression`, is true.

A.3.6 Atomic actions

A UML atomic action is represented using a JAL single statement or expression. JAL supports the following atomic actions: *CreateObjectAction*, *DestroyObjectAction*, *ReadLinkAction*, *CreateLinkAction*, *DestroyLinkAction*, *CallOperationAction*, *ReplyAction*, *ReadStructuralFeatureAction*, *WriteStructuralAction*, *ValueSpecificationAction*, *ReadVariableAction*, and *WriteVariableAction*. These atomic actions are represented by create object expression, destroy object statement, read object expression, create link statement, delete link statement, call operation expression, return statement, read attribute expression, write attribute statement, calculation expression, read variable expression, and write variable statements.

A.3.6.1 Create object expression

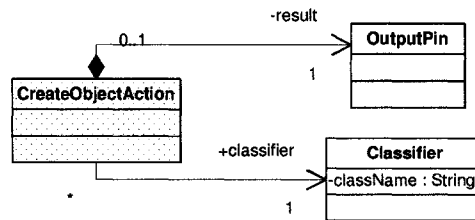


Figure A.1: Create Object Action Meta-Class Diagram [56].

The *CreateObjectAction* shown in Figure A.1 is represented by the JAL create object expression, `_create_object_<className>()`.

- `_create_object_` is the keyword representing the create object expression.
- `<className>` is the name of the class that is instantiated. It is denoted by the attribute `Classifier.className` in Figure A.1.

The create object expression evaluates into the reference of the newly created object. When a new object is created, all its attributes are undefined, unless default values for the attributes are given in the class diagrams.

A.3.6.2 Destroy object statement

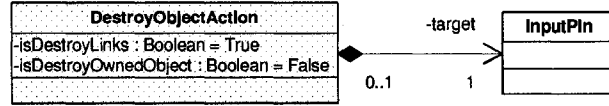


Figure A.2: Destroy Object Action Meta-Class Diagram [56].

The *DestroyObjectAction* shown in Figure A.2 is represented by the delete object statement, `_delete_object_(<objectHandle>)`.

- `_delete_object_` is the keyword representing the delete object statement.
- `<objectHandle>` is an expression representing the value associated with the `target` `InputPin` of the *DestroyObjectAction*. `<objectHandle>` evaluates to the object that is destroyed.

When an object is destroyed, all the links are also destroyed, but all the owned objects are left unchanged.

A.3.6.3 Read link expressions

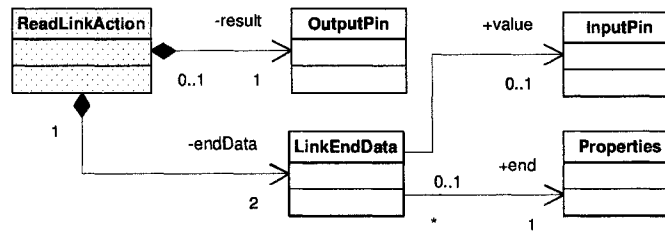


Figure A.3: Read Link Action Meta-Class Diagram [56].

The *ReadLinkAction* shown in Figure A.3 is represented by an association navigation expression, `[ObjectHandle.]<AssociationEndName>`.

- `ObjectHandle` is an expression representing the value associated with the `InputPin` object, `endData.value`, of the *ReadLinkAction*. This expression evaluates to an object at one end of a link.

- `<AssociationEndName>` is the identifier representing the name of the `Properties` object, `endData.end`, of the `ReadLinkAction`. This `endData.end` object represents the association end at the other end of the link.

This expression evaluates into a read-only collection of objects that associates with the object, `ObjectHandle`, with the association-end named, `AssociationEndName`. In JAL, two operations can be applied to this collection:

- `_get_Total()`: Returns the number of objects in the collection.
- `_get_At(index)`: Returns an object at the `<index>` position in the collection.

A.3.6.4 Create link statement

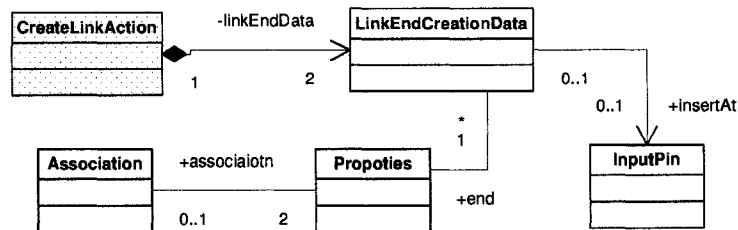


Figure A.4: Create Link Action Meta-Class Diagram [56].

The *CreateLinkAction* shown in Figure A.4 is represented by the create link statement, `_create_link_<AssociationName>(<objectHandle1>, <objectHandle2>)`.

- `_create_link_` is the keyword representing the `CreateLinkAction`.
- `<AssociationName>` is the identifier representing the name of the `Association` object in the one-element bag `linkEndData.end.association`, of the `CreateLinkAction`. This association is instantiated during the execution of the action.
- `<objectHandle1>` and `<objectHandle2>` are expressions representing the values associated with the `InputPin` objects in the bag `linkEndData.insertAt` of the `CreateLinkAction`. These expressions evaluate to the objects at the two

ends of the newly created link. These objects must be instances of classes at the two ends of the instantiated association.

The create link statement creates a link that is an instance of the association with the name, `<AssociationName>`. This link connects two objects represented by the expressions, `<objectHandle1>` and `<objectHandle2>`.

A.3.6.5 Delete link statement

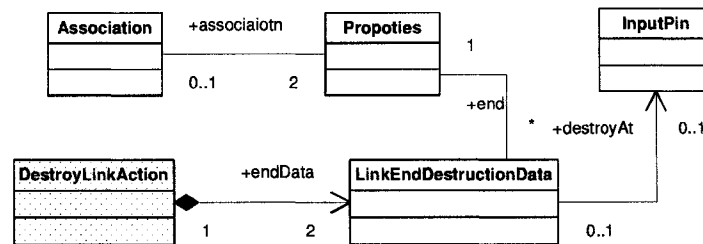


Figure A.5: Destroy Link Action Meta-Class Diagram [56].

The *DestroyLinkAction* shown in Figure A.5 is represented by the delete link statement, `_delete_link_<AssociationName>(<objectHandle1>, <objectHandle2>)`.

- `_delete_link_` is the keyword representing the *DestroyLinkAction*.
- `<AssociationName>` is the identifier representing the name of the **Association** object in the one element bag, `endData.end.association`, of the *DestroyLinkAction*. This association is of the type of the link that is deleted.
- `<objectHandle1>` and `<objectHandle2>` are expressions representing the values associated with the **InputPin** objects in the bag `endData.destroyAt` of the *DestroyLinkAction*. These expressions evaluate to the objects at the two ends of the deleted link. These objects must be instances of classes at the two ends of the association of the deleted link.

The statement deletes the link that connects two objects represented by the expressions `<objectHandle1>` and `<objectHandle2>`. The link is an instance of the association with the name, `<AssociationName>`.

A.3.6.6 Call operation expression

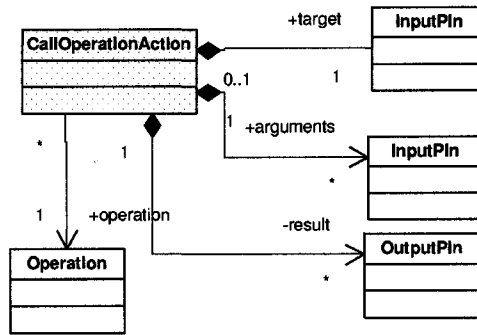


Figure A.6: Call Operation Action Meta-Class Diagram [56].

The *CallOperationAction* shown in Figure A.6 is represented by the call operation expression, `[<ObjectHandle>].<OperationName>([<Parameter1>, <Parameter2>, ...])`.

- `<ObjectHandle>` is an expression representing the value associated with the `InputPin` object, `target`, of the `CallOperationAction`. This expression evaluates into the target object of the operation call.
- `<OperationName>` is the identifier representing the name of the `Operation` object, `operation`, of the `CallOperationAction`. The operation object represents the called operation.
- `<Parameter1>`, `<Parameter2>`, ... is a comma separated list of expressions. Each of the expression in the list represents a value associated with an `InputPin` object in the set, `arguments`, of the `CallOperationAction`. This set represents the arguments of the operation call.

This expression evaluates into the value that is returned by the called operation. The return value is represented by the `OutputPin` object, `result`, of the `CallOperationAction`.

A.3.6.7 Return statement

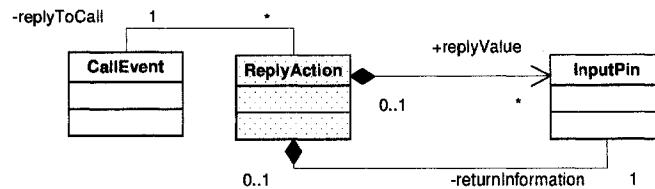


Figure A.7: Reply Action.

The *ReplyAction* shown in Figure A.7 is represented by the return statement, `return <ReturnValue>`.

- `return` is the keyword representing the *ReplyAction*.
- `<ReturnValue>` is an expression representing the value associated with the `InputPin` object, `returnValue`, of the *ReplyAction*.

The *ReplyAction* terminates the execution of the current operation call, and returns the `returnValue` to the operation that called the current operation. In the UML specification, an operation call can return multiple values. In JAL, it is assumed that each operation call can return at most one value.

A.3.6.8 Read attribute expression

The *ReadStructuralFeatureAction* shown in Figure A.8 is represented by the read attribute expression, `[objectHandle]._get_<AttributeName>()`.

- `_get_` is the keyword representing the *ReadStructuralFeatureAction*.

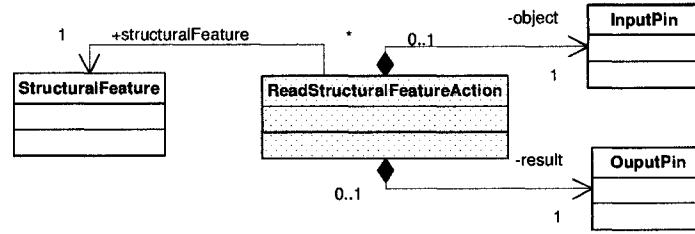


Figure A.8: Read Structural Feature Action Meta-Class Diagram [56].

- `<AttributeName>` is the identifier representing the name of the `StructuralFeature` object, `structuralFeature`, of the `ReadStructuralFeatureAction`. This object represents the accessed attribute.
- `<objectHandle>` is an expression representing the value associated with the `InputPin`, `object`, of the action. The expression evaluates to the object that contains the accessed attribute.

This expression returns the attribute value, `<AttributeName>`, of the object, `<objectHandle1>`.

A.3.6.9 Write attribute statement

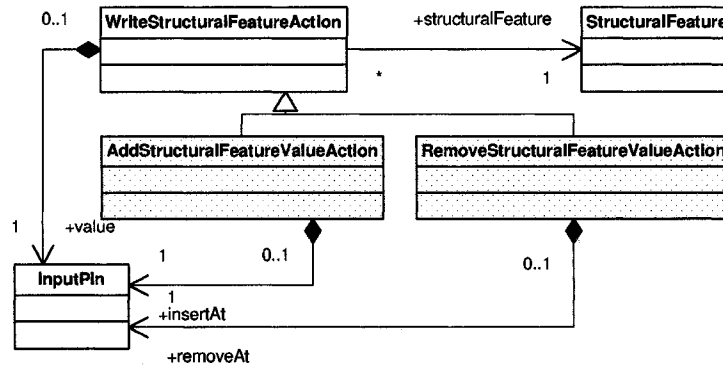


Figure A.9: Write Structural Feature Actions Meta-Class Diagram [56].

The `WriteStructuralFeatureAction` shown in Figure A.9 is represented by the write attribute statement, `[ObjectHandle]._set_<AttributeName>(<value>)`

- `_set_` is the keyword representing the *WriteStructuralFeatureAction*.
- `<AttributeName>` is the identifier representing the name of the *StructuralFeature* object, `structuralFeature`, of the *WriteStructuralFeatureAction*. This object represents the accessed attribute.
- `<objectHandle>` is an expressions representing the value associates with the *InputPin* object, `insertAt`, of the *WriteStructuralFeatureAction*. This expression evaluates to the object that contains the accessed attribute.
- `<value>` is an expression that evaluates to the new value that is assigned to the attribute, `<AttributeName>`.

This statement removes the old value of the attribute, `<AttributeName>`, of the object, `<objectHandle1>`. It then adds the new value, `<value>`, to the attribute.

A.3.6.10 Calculation expression

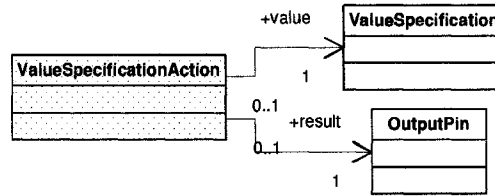


Figure A.10: Value Specification Actions Meta-Class Diagram [56].

The *ValueSpecificationAction* shown in Figure A.10 is represented by a calculation expression. In JAL, the calculation expressions for Boolean, integer, float, and string expressions are the same as those in Java.

The following collection operations are supported:

- `<CollectionExpression>._get_Total()`: Get the number of items in the collection.
- `<CollectionExpression>._get_At(<index>)`: Get an item at the `<index>` position in the collection.

- `<CollectionExpression>..add(<Expression>)`: Add an item to the end of the collection.
- `<CollectionExpression>..remove(<index>)` Remove an item at the `<index>` position in the collection.

A.3.6.11 Accessing variables

The syntax for accessing a variable in JAL is the same as in Java.

A.3.7 Compound statement

In JAL, a compound statement represents a combination of atomic actions. A JAL compound statement consists of an expression or a single statement that uses another expression as a parameter.

Appendix B

UMLAnT User Guide

This appendix describes how to use the *UMLAnT* tool. The steps are illustrated using a small example of a simple “Product Management” system. A partial class diagram of the system is shown in Figure B.1. The system has one **ProductCatalog**, which is used to managed zero or many **Products**. Each **Product** is categorized into exactly one **Category**.

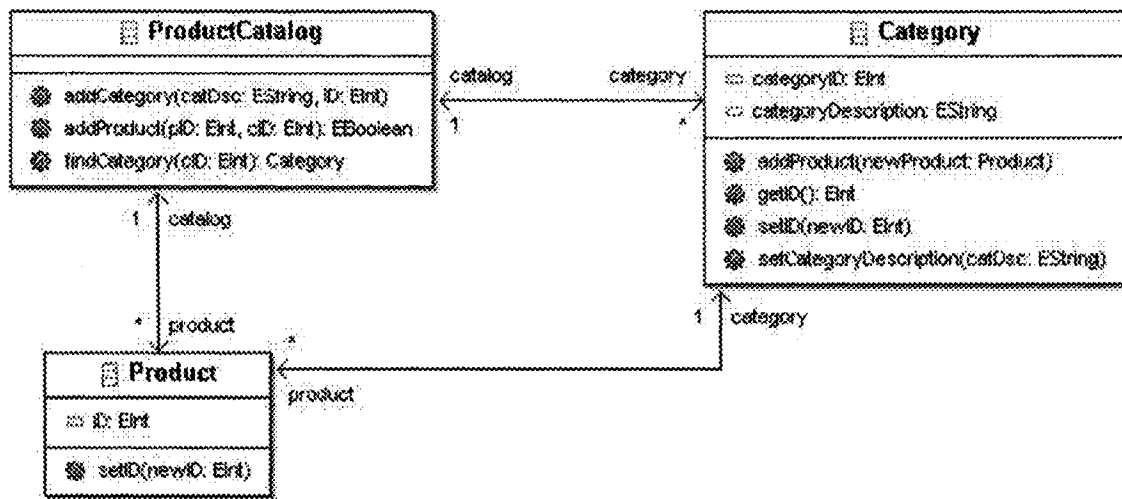


Figure B.1: *DUT* Class Diagram of the Product Management System.

The steps are listed below and explained in subsequent sections:

1. Specify the design under test (*DUT*) in Eclipse using the Omondo EclipseUML graphical editor and *UMLAnT* plug-in editor.

2. Use *UMLAnT*'s TDUT generator to obtain the testable form (*TDUT*) of the model.
3. Specify JUnit like test inputs.
4. Execute tests and observe test results.
5. Use *UMLAnT*'s animator to animate the test execution.

B.1 Creating a *DUT*

1. Create a new Eclipse project:
 - Select **File** → **New** → **Project**. Choose **“EMF Project”**.
 - Set the name for the project, e.g., **“Demo”**. Click **“Next”**. Chose **“Create an empty Project”**, click **“Finish”**.
2. Create an *EMF* class diagram for the *DUT*:
 - On the **Package Explorer** view, right click on the newly created Project (**“Demo”** in our example) and then choose **“New → Other”**.
 - Select **“EMF Class Diagram”** and then click **“Next”**.
 - In the **“File name”**, type the name of the new *EMF* file. Usually, the name of this file is the same as the project name, which is **“Demo.ecd”** in the example used here.
 - In **“Package”** text box, type the name of the package, for example, **“Demo”**, and then click **“Finish”**.
 - Open the newly created *EMF* file (which is **“Demo.ecd”** in our example) with the graphical editor. Using the editor, create the class diagram for the *DUT*. For example, create the class diagram as shown in Figure B.1.
 - Save the class diagram by press **“Ctrl-S”**.
3. Open the **“.ecore”** file in *EMF*'s tree-based sample *Ecore editor* (e.g., the **“Demo.ecore”** file):

- Choose the file in the package explorer and right click.
- Choose **Open As** → **Sample Ecore Model Editor** from the context menu to open the file in the tree editor.

UMLAnT provides additional capabilities to the editor to let the user specify constraints and operation behaviors.

4. Right click anywhere within the tree view and choose **UMLANT** → **Specify Invariants** to input system OCL constraints. For example, enter the OCL constraint as shown in Figure B.2.

```
context ProductCatalog inv:
  -- There should not be 2 categories with the same categoryID
not self.category->exists(c1, c2:Category|(c1.categoryID =
c2.categoryID) and (c1<>c2))
```

Figure B.2: OCL Constraint for the “Demo” project.

5. Collapse all the nodes of the tree to see attributes, operations and associations. Right click on each operation and choose **UMLANT** → **Specify Behavior in JAL** to input JAL specification. Enter the JAL segments shown in Figures B.3, B.4, B.5, B.6, B.7, B.8, and B.9.

This completes the specification phase of the system.

```
Category catg;
catg = _create_object_Category();
catg.setID(ID);
catg.setCategoryDescription(catDsc);
_create_link_ProductCatalog_Category_category_catalog(catg,
this);
```

Figure B.3: JAL segment for `ProductCatalog::addCategory()`.

```
_set_categoryID(newID);
```

Figure B.4: JAL segment for `Category::setID()`.

```
return _get_categoryID();
```

Figure B.5: JAL segment for `Category::getID()`.

```
Category catg;  
catg = _create_object_Category();  
catg.setID(ID);  
catg.setCategoryDescription(catDsc);  
_create_link_ProductCatalog_Category_category_catalog(catg,  
this);
```

Figure B.6: JAL segment for `ProductCatalog::addCategory()`.

```
Category ctg=this.findCategory(cID);  
if(ctg!=null){  
    Product p = _create_object_Product();  
    p.setID(pID);  
    _create_link_Product_ProductCatalog_catalog_product(this, p);  
    // _create_link_Product_Category_category_product(ctg, p);  
    return true;  
}  
return false;
```

Figure B.7: JAL segment for `ProductCatalog::addProduct()`.

```
_set_iD(newID);
```

Figure B.8: JAL segment for `Product::setID()`.

B.2 Generating *TDUT*

A testable form of the system is generated from the model specification. In the left window of package explorer, choose the “.ecore” file(e.g., “Demo.ecore” and then

```

int total=category._getTotal();
int i=0;
while(i<total){
    Category c=category._getAt(i);
    int id=c.getID();
    if(id==cID){
        return c;
    }
    i=i+1;
}
return null;

```

Figure B.9: JAL segment for `ProductCatalog::findCategory()`.

right click. Choose **UMLANT** → **Generate Testing Package**. This generates the following files under the “src” directory:

1. A testable form of the model in the Java package, `testable_models.Project`.
2. The *USE* specification `Project.use` in the `testable_models.Project` package.

Chapter 8 contains more information on the integration of *UMLAnT* and *USE*.

3. A JUnit-like framework for running tests in the package, `testable_models.Project.Framework`.
4. A `SampleExample` test driver in the package, `testable_models.Project.Tests`.

B.3 Writing test cases

Writing test cases for models is similar to writing JUnit¹ test cases for testing code. One difference while writing test cases for models is that the test case class extends the `testable_models.Project.Framework.ModelTestCase` class, instead of the `junit.framework.TestCase` class. The `ModelTestCase` class integrates the

¹<http://www.junit.org>

model execution engine and the *USE* subsystem. The class also provides a tester with an additional assert method, `assertConformance()`, which checks the conformance of a particular run-time object configuration against the model specification. A sample test case, `testable_models.Project.Tests.SampleTestCaseForModel`, is generated as part of `testable_models.Project.Tests` package. As an example, we edit the “testOne” method of the `SampleTestCaseForModel` class and enter the code as shown in Figure B.10.

```
public void testOne(){
    //Create start configuration
    ProductCatalog pc = factory._create_object_ProductCatalog(this);
    pc.addCategory("Book", 1);

    //send test signal(s)
    pc.addProduct(2, 1);

    //Use assertXXXX as oracle to check the results
    this.assertConformance();
}
```

Figure B.10: The Code for the “testOne” method.

B.4 Launching the test runner

The class `testable_models.Project.Framework.ModelTestsRunner` generated by *UMLAnT* loads and runs test cases written by the tester. in the package `testable_models.Project.Tests`. To run the tests, the *CLASSPATH* must contain these jar files: `junit.jar`, `UMLAnT.jar`, `antlt-2.7.5.jar` and `USE.jar`. The *CLASSPATH* needs to contain the “SWT” library. We can add these .jar files and the “SWT” libraries to the classpath as follows:

1. On the **Package Explorer** view, right click on the current project (e.g., “Demo”, choose “**Properties**”.

2. Select **“Java Build Path”**.
3. Select **“Libraries”** tab.
4. Click **“Add Library”**, and then select **“Standard Widget Toolkit (SWT)”**. Click **“Next”**, and then **“Finish”**.
5. Click **“Add External JARs”** and select **“... plugins\edu.colostate.edu.umlant_1.0.0\use.jar”**.
6. Repeat step 5 to add the following .jar files:
 - **“... \plugins\edu.colostate.edu.umlant_1.0.0\antlt-2.7.5.jar”**
 - **“... \plugins\edu.colostate.edu.umlant_1.0.0\umlant.jar”**
 - **“... \plugins\org.junit_3.8.1\junit.jar”**
7. Click **“OK”**.

B.5 Running test cases

We can execute the *ModelTestRunner* as follows:

1. On the menu bar, select **“Run → Run...”**
2. Click **“Browse”** to select the current Project (e.g., **“Demo”**. The main class automatically gets set to **“ModelTestRunner”**.
3. Choose the **“(x)=Arguments”** tab. On **“VM arguments”**, enter:


```
-Djava.library.path =  
“‘[...]\plugins\org.eclipse.swt.win32_3.0.2\os\win32\x86’”
```
4. Click **“Run”**

Once the test runner is launched, it lists all the test case drivers written by the tester. Testers can choose and run a test driver one at a time. The list of drivers is shown in the left pane and the results are displayed on the right pane. In our example, there is one test driver, **“SampleTestCaseForModel”**. Select the test driver and click on **“Run”**.

The tests are executed and the test runner reports failures, if any. For our example, *UMLAnT* reports one failure, which occurs because of a fault in the JAL statement for the “`ProductCatalog::addProduct()`” operation in Figure B.7. To fix the fault, we can uncomment the currently commented line in the JAL specification. To close the test runner, the tester must click on “**Exit**”.

B.6 Animating the execution

On the **Package Explorer** view, the tester will see a “.mtd” file (e.g., “`Demo.mtd`”). If the file is not visible, the view needs to be refreshed. This is done by clicking on the current project (e.g., “`Demo`”) and then choosing “**Refresh**”). The “.mtd” file logs all the actions that were executed during the test. Right clicking on the file and choosing “**UMLAnT Animated Debugger → Run Debugger**” launches the animator..

The animator window contains two tabs: “**Object Diagram**” and “**Sequence Diagram**”, which are the two views used to animate the test execution. Three buttons under the menu bar are used to control the animation. The right-most button allows the tester to step through the actions; the middle button allows the tester to run to the end of the animation; the left-most button is currently inactive. Since the animation is performed quickly in the current version of the tool, if a tester chooses to run to the end of the animation, we can only see the end result once the animation is complete. In a future version, when the tester chooses to run to the end of the animation, the tool will pause briefly after animating each action. During each pause, the left-most button can be used to stop the animation.

REFERENCES

- [1] A. Abdurazik and J. Offutt. Using UML collaboration diagrams for static checking and test generation. In *Proceedings of the 3rd International Conference on the UML*, pages 383–395, York, UK, October 2000.
- [2] W. Adrion, M. Branstad, and J. Cherniavsky. Validation, verification, and testing of computer software. *ACM Computing Survey*, 14(2):159–192, December 1982.
- [3] 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, April-June 2003.
- [4] J. F. Benders. Partitioning procedures for solving mixed-variables programming problems. *Numerische Mathematik* 4, pages 238–252, 1962.
- [5] A. Bertolino and M. Marre. Automatic generation of path covers based on the control flow analysis of computer programs. *IEEE Transactions on Software Engineering*, 20(12):885–899, December 1994.
- [6] R. V. Binder. *Testing Object-Oriented Systems. Models, Patterns, and Tools*. Addison Wesley, USA, 2004.
- [7] R. V. Binder. *Testing Object-Oriented Systems Models, Patterns, and Tools*. Object Technology Series. Addison Wesley, Reading, Massachusetts, October 1999.
- [8] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.
- [9] Borland Software Corporation. Together 6.0. <http://borland.com/together/>, 2003.
- [10] R. S. Boyer, B. Elspas, and K. N. Levitt. Select—a formal system for testing and debugging programs by symbolic execution. In *Proceedings of the International Conference on Reliable Software*, pages 234–245, April 1975.

- [11] M. M. Brandis and H. Mossenbock. Single-pass generation of static single-assignment form for structured languages. *ACM Transactions on Programming Languages and Systems*, 16(6):1684–11698, November 1994.
- [12] L. Briand, J. Cui, and Y. Labichi. Towards automated support for deriving test data from UML statecharts. In *Proceedings of the 6th International Conference on the UML*, pages 265–279, San Francisco, CA, USA, October 2003.
- [13] L. Briand and Y. Labiche. A UML-based approach to system testing. *Software and Systems Modeling*, 1(1):10–42, Sept 2002.
- [14] T. Dinh-Trong, S. Ghosh, R. France, B. Baudry, and F. Fleury. A Taxonomy of Faults for UML Designs. In *2nd MoDeVa workshop - in conjunction with MoDELS*, October 2005.
- [15] T. T. Dinh-Trong. Rules For Generating Code From UML Collaboration Diagrams and Activity Diagrams. Master’s thesis, Colorado State University, Fort Collins, Colorado, 2003.
- [16] G. Engels, R. Hucking, S. Sauer, and A. Wagner. UML collaboration diagrams and their transformations to Java. In *Proceedings of the 2nd International Conference on the UML*, pages 416–429, Fort Collins, CO, USA, October 1999.
- [17] R. Ferguson and B. Korel. The chaining approach for software test data generation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(1):63–86, January 1996.
- [18] R. France, S. Ghosh, T. Dinh-Trong, and A. Solberg. Model-Driven Development Using UML 2.0: Promises and Pitfalls. *Computer*, 39(2), February 2006.
- [19] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.
- [20] E. Gamma and K. Beck. Junit. <http://www.junit.org/>, 2001.
- [21] S. Ghosh, R. B. France, C. Braganza, N. Kawane, A. Andrews, and O. Pilskalns. Test adequacy assessment for UML design model testing. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 332–343, Denver, CO, 2003.
- [22] M. Gogolla, J. Bohling, and M. Richters. Validation of UML and OCL models by automatic snapshot generation. In *Proceedings of the 6th International Conference on the UML*, pages 265–279, San Francisco, CA, USA, October 2003.
- [23] M. Gogolla, J. Bohling, and M. Richters. Validating UML and OCL Models by Automatic Snapshot Generation. *Software and System Modeling*, 4(4):386–398, Nov 2005.

- [24] R. E. Gomory. An algorithm for integer solutions to linear programs. *Recent Advances in Mathematical Programming*, 1963.
- [25] N. Gupta, A. P. Mathur, and M. L. Soffa. Automated test data generation using an iterative relaxation method. In *Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 231–244, Lake Buena Vista, FL, USA, November 1998.
- [26] D. Harel and E. Gery. Executable Object Modeling with Statecharts. *IEEE Computer*, 30(7):31–42, 1997.
- [27] P. V. Hentenryck, L. Micheal, and Y. Deville. *Numerica. A modeling language for global optimization*. The MIT Press, Cambridge, Massachusetts, London, 1997.
- [28] IBM. Rational Rose. <http://www-306.ibm.com/software/awdtools/developer/rosexde/>, 2004.
- [29] D. Jackson, I. Schechter, and I. Shlyakhter. Alcoa: The alloy constraint analyzer. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE '00)*, pages 730–733, Limerick, Ireland, June 2000.
- [30] B. Joy, G. Steele, J. Gosling, and G. Bracha. *Java(TM) Language Specification*. The Java Series. Addison-Wesley Professional, Reading, Massachusetts, June 2000.
- [31] Kabira Technology. Kabira. <http://www.kabira.com/>, 2006.
- [32] N. Kawane. EPTUD: An Eclipse plugin for testing UML design models. Master's thesis, Colorado State University, Fort Collins, Colorado, 2005.
- [33] Kennedy Carter. iUML. <http://www.kc.com/>, 2006.
- [34] Y. Kim, H. Hong, D. Bae, and S. Cha. Test cases generation from UML state diagrams. *IEE Proceedings - Software*, 146(4):187–192, 1999.
- [35] J. C. King. A new approach to program testing. In *Proceedings of the international conference on Reliable software*, pages 228–233, Los Angeles, LA, USA, 1975.
- [36] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.
- [37] A. Kleppe, J. Warmer, and W. Bast. *MDA Explained - The Model Driven Architecture: Practice and Promise*. Addison-Wesley, 2005.
- [38] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, August 1990.

- [39] K. W. Krause, R. W. Smith, and M. A. Goodwin. Optimal software test planning through automated network analysis. In *IEEE Proceedings of the 1973 Symposium on Computer Software Reliability*, pages 18–22, New York, 1973.
- [40] C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process Second Edition*. Prentice-Hall, 2002.
- [41] T. Massoni, R. Gheyi, and P. Borba. A UML class diagram analyzer. Technical report, Information Center, Federal University of Pernambuco, Brazil, September 2004.
- [42] S. Mellor and M. Balcer. *Executable UML: A Foundation for Model Driven Architecture*. Addison Wesley Professional, 2002.
- [43] Mentor Graphics, Accelerated Technology division. Bridgepoint. <http://www.mentor.com/>, 2006.
- [44] E. F. Miller, M. R. Paige., J P. Benson, and W. R. Wisheart. Structural techniques of program validation. In *Digest COMPCON74*, pages 161–164, 1974.
- [45] G. J. Myers. *The Art of Software Testing*. John Wiley and Sons, New York, NY, 1979.
- [46] U. A. Nickel, J. Niere, R. P. Wadsack, and A. Zundorf. Roundtrip Engineering with FUJABA. In *Proceedings of the 2nd Workshop on Software-Engineering*, Bad Honnef, Germany, August 2000.
- [47] S. C. Ntafos and S. Louis Hakimi. On path cover problems in digraphs and applications to program testing. *IEEE Transactions of Software Engineering*, SE-5:520–529, Sept. 1979.
- [48] J. Offutt and A. Abdurazik. Generating tests from UML specifications. In *Proceedings of the 2nd International Conference on the UML*, pages 416–429, Fort Collins, CO, USA, October 1999.
- [49] T. Ostrand and M. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676–686, June 1988.
- [50] O. Pilskalns, A. Andrews, S. Ghosh, and R. B. France. Rigorous testing by merging structural and behavioral uml representations. In *Proceedings of the 6th International Conference on the Unified Modeling Language*, pages 234–248, San Francisco, CA, USA, October 2003.
- [51] D. Riehle, S. Fraleigh, D. Bucka-Lassen, and N. Omorogbe. The Architecture of a UML Virtual Machine. In *Proceedings of the 2001 Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '01)*, pages 327–341. ACM Press, 2001.

- [52] M. Scheetz, A. von Mayrhauser, R. France, E. Dahlman, and A. E. Howe. Generating test cases from an OO model with an AI planning system. In *Proceedings of the 10th International Symposium on Software Reliability Engineering*, pages 250–259, Boca Raton, FL, USA, January 1999.
- [53] K. Tai. On program testing criteria. In *Proceedings of IEEE Computer Society's 3rd International Computer Software and Applications Conference*, pages 494–499, November 1979.
- [54] The Object Management Group. MDA Guide. Version 1.0.1, OMG, omg/03-06-01, 2003.
- [55] The Object Management Group. Unified Modeling Language: Infrastructure. Version 2.0, OMG, formal/05-07-05, 2005.
- [56] The Object Management Group. Unified Modeling Language: Superstructure. Version 2.0, OMG, formal/05-07-04, 2005.
- [57] The Object Management Group. Object Constraint Language - OMG Available Specification. Version 2.0, OMG, formal/06-05-01, 2006.
- [58] N. T. Sy and Y. Deville. Consistency techniques for interprocedural test data generation. In *Proceedings of the 9th European Software Engineering Conference held jointly with 10th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 108–117, Helsinki, Finland, September 2003.
- [59] E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.
- [60] H. S. Wang, S. R. Hsu, and J. C. Lin. A generalized optimal path selection model for structural program testing. *The Journal of Systems and Software*, 10:55–63, 1989.
- [61] E. Weyuker. Axiomatizing software test data adequacy. *IEEE Transactions on Software Engineering*, 12(11):1128–1138, June 1986.
- [62] W. Grosso. *Java RMI*. O'Reilly, Sebastopol, CA, October 2002.