Test Scenario Generation and Management

A THESIS SUBMITTED
FOR THE AWARD OF THE DEGREE OF

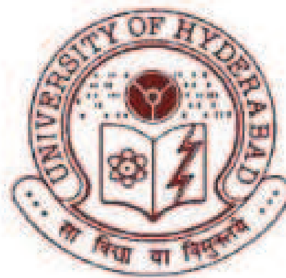
Doctor of Philosophy

in

Computer Science

by

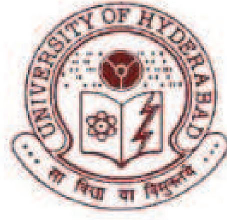
Ms. Sapna P.G.



Department of Computer and Information Sciences
School of Mathematics & Computer/Information Sciences
University of Hyderabad
Hyderabad - 500 046, INDIA

AUGUST 2010

Department of Computer & Information Sciences
School of Mathematics and Computer & Information Sciences
University of Hyderabad, Hyderabad – 500046
INDIA



CERTIFICATE

This is to certify that the thesis work entitled **'Test Scenario Generation and Management'** being submitted by **Ms. Sapna P.G.** (Regd. No. **06MCPC10**) in partial fulfillment of the requirement for the award of degree of **Doctor of Philosophy in Computer Science** of the University of Hyderabad, is a record of bonafide work carried out by her under my supervision.

The matter embodied in this report has not been submitted for the award of any other degree/diploma of any other Institute/University.

Prof. Hrushikesha Mohanty
(Supervisor)
Department of Computer & Information Sciences
University of Hyderabad, Hyderabad – 500046, INDIA

Prof. Arun Agarwal
(Head)
Department of Computer &
Information Sciences (CIS)
University of Hyderabad
Hyderabad – 500046, INDIA

Prof. Amarnath
(Dean)
Schools of Maths. & Computer/
Information Sciences (MCIS),
University of Hyderabad
Hyderabad – 500046, INDIA

DECLARATION

I hereby declare that the work embodied in this thesis has been carried out by me under the supervision of Prof. Hrushikesha Mohanty in the Department of Computer and Information Sciences, University of Hyderabad, Hyderabad, India, as per the Ph.D. ordinances of the University. This thesis has not been submitted for the award of a research degree of any other university. Information derived from the published and unpublished work of others has been acknowledged and a list of references is given. Where work described in this thesis have been published, references are cited for the same in the body of the work and a list of such publications given.

Sapna P.G.
(Reg. No. 06MCPC10)

Whatever I am offered in devotion with a pure heart - a leaf, a flower, fruit, or water - I accept with joy.

- the Bhagavad Gita.

To

Achan & Arun

ABSTRACT

Software testing is performed to determine whether or not a system satisfies the requirements of customer. Due to the increasing size of software systems, manual testing becomes untenable. Hence, there is a need to automate techniques for software testing. Specification based testing intends to confirm implementation of specifications. This work is concerned with user requirements collected during requirement engineering of an intended system.

The Object Management Group defines the Unified Modelling Language(UML) as a general-purpose visual modeling language that is used to specify, visualize, construct and document artifacts of a software system. UML captures information about the static structure as well as dynamic behaviour of a system. The static structure defines objects as well as the relationship between objects that are part of the system implementation usually represented in use case, class and component diagram. Dynamic behaviour of the system is specified by the activity, sequence and state diagram.

The semi-formal nature of UML has both advantages and disadvantages: the advantage primarily lies in its ease of use as well as understandability by various stakeholders of the system. Also, different diagrams can be used to model varying aspects of the system. The same leads to difficulties in the form of maintaing completeness and consistency within and between UML diagrams. Specification based testing using UML needs consistent and complete UML diagrams. Again for testing, scenarios respresenting working of intended system are extracted and studied for the purpose. The study includes generation, prioritization and selection of scenarios from UML specification.

Diagram consistency checking follows a horizontal transformation approach and well formed rules are applied. Particularly, diagrams(through their artifacts) are stored in RDBMS and consistency checking rules are formulated as triggers, assertions, etc to alert user incase inconsistency is encountered.

Each use case representing functional system requirement is illustrated by ac-

tivity diagrams; and scenarios are extracted by traversing these diagrams. Domain specific relations among activities are considered to reduce the number of scenarios that may turn large due to multiplicative factor contributed due to concurrent activities. A tool is developed to support scenario extraction process.

Use cases and scenarios are prioritized to obtain an ordering for execution of test cases. Use case priority is obtained both from the customer as well as computed from structural complexity of a use case diagram. Also, the structural primitives of the activity diagrams are used to obtain the priority of a scenario. The priority of a scenario is computed by a weighted sum of both use case priority and scenario priority.

An effective ordering of test scenarios for execution helps in early detection of defects. However, given constraints of cost and time, it may not be possible to execute the test suite completely. Instead, a subset of test scenarios may be selected for testing. The fourth contribution is selection techniques to help select a subset of scenarios based on the similarity that exists between scenarios. The first technique uses Levenshtein distance as a measure to calculate dissimilarity between scenarios. A second technique looks at finding common subscenarios, their length and relative position to calculate similarity. A third technique uses Agglomerative Hierarchical Clustering(AHC) to cluster similar scenarios for selection of a representative scenario.

Given a large number of use cases and scenarios for even a medium sized system, there is a need for effective management of use cases and scenarios to facilitate testing. The thesis proposes an ontology to aid scenario management. Scenario management is demonstrated using Protege, an open source ontology editor. A tool also has been developed for scenario generation and prioritization. The usability of the proposed concepts are demonstrated through two case studies and results are analyzed to study the impact of different kinds of prioritization and selection techniques.

Acknowledgments

I take this opportunity to thank all those who have helped me in the successful completion of work leading to this thesis and in making my stay at the University of Hyderabad, memorable.

First, I am ever indebted to this university for giving me an opportunity to undertake this exciting and challenging journey. With an ideal mix of science and arts, aiming for excellence and developing a holistic individual, this university has helped me grow as a human being with its beautiful environment, perfect ambience and remarkable people. I will always cherish the time I spent here at the university.

My deepest gratitude is to my supervisor, Prof. Hrushikesha Mohanty for his guidance and encouragement throughout the period of work. Learning and working with him has been an enriching experience both in terms of computer science as well as the world as a whole. The discussions and meetings taught me the nuances of research. He has been there for inspiration and ideas, and has always challenged me to think on my own and given me the space to grow. Without his time and dedication I could not have completed this work.

I thank Prof. Arun Agrawal, Head, Department of Computer & Information Sciences, University of Hyderabad, for his support and wishes. I sincerely thank Prof. T. Amaranath, Dean, School of MCIS, for his valuable advices in completing this thesis.

I appreciate deeply the contributions of the doctoral committee members, Prof. Bapi Raju, Dr. Durga Bhavani and Dr. Siba K. Udgata. Their detailed comments, interesting questions and insights have been of great value to me.

I have had an interesting visit to the Fondazione Bruno Kessler (previously IRST), Trento, Italy during the PhD programme. I would like to thank Prof. Paolo Tonella and Dr. Angelo Susi for their inspiration and collaboration. Prof. Paolo Tonella's detailed knowledge on software testing topics made working with him an enlightening experience. I also thank members of the Software Engineering Group for the discussions on issues related to testing.

I would also like to thank all the faculty members of the department who provided the right ambience for constructive research and enriched my life by being part of this journey. I thank all the staff of the Department of CIS for their constant support, members of the Artificial Intelligence Lab and the staff of Indira Gandhi Memorial Library for providing me help and facilities at the most crucial of times. Special thanks goes to Ms. B. Chandrakala, Ms. B. Padmamma and Mr. Tony A. Hill who have been patient with my innumerable requests for facilities.

Thanks is due to colleagues and friends with whom I have worked as part of this journey: Krishna Reddy, Cu Duy Nguyen, Sandhya Banda, Sumagna Patnaik, Uma Maram Reddy and Mini. You taught me many things, computer science and otherwise through the many interactions we had. I would also like to thank my friends, without whom life here, at the university, wouldn't have been the same: Lavanya, Ganga, Dharshini, Swathi, Usha and Ghouse.

Last but not the least, special acknowledgement goes to the two men in my life: Mr. P. Gopinathan, my father, who is my inspiration and my strength and B. Arunkumar, my better half, for his never ending enthusiasm, encouragement and endless love. I thank my mother, P. Swarnalatha, who taught me to dream and live them. But for them, I would not be here to realize my dream. I also pay my gratitude to my sisters, Sajna and Sadna for their support and encouragement.

. . . Thank you all!

Sapna P.G.

Contents

1 Introduction **1**

1.1 Research Problem 2

1.2 Contribution 3

1.3 Terminology 6

1.4 Thesis Structure 6

2 Survey **8**

2.1 Introduction 8

2.2 Specification based Testing 10

2.2.1 Definition 11

2.2.2 UML and Specification based Testing 12

2.2.3 Advantages and Disadvantages of Specification based Testing using UML 13

2.3 Usage of UML 14

2.4 Checking Consistency of UML diagrams 17

2.4.1 Classification of consistency techniques 18

2.4.2 Literature Review 20

2.4.3 Discussion 22

2.5 Test Scenario Generation 27

2.5.1 Literature Review 27

2.5.2 Discussion 29

2.6 Prioritization 33

2.6.1 Classification of prioritization techniques 34

2.6.2 Classifying Metrics 38

2.6.3 A Taxonomy of Features for comparing work on Test Scenario Prioritization 40

2.6.4 Discussion 41
2.7 Test Case Selection 46
2.7.1 Classification 47
2.7.2 A Taxonomy of Features for comparing work on Test Scenario Selection 50
2.7.3 Discussion 51
2.8 Use of Ontology in Software Engineering 56
2.8.1 Ontology - Definition 56
2.8.2 Classification of Ontologies 58
2.8.3 Use of Ontologies 59
2.8.4 Discussion 66
2.9 Conclusion 68

3 Checking consistency of specification 71
3.1 Introduction 72
3.2 UML and Consistency Checking 75
3.3 UML and the Object Constraint Language 78
3.4 Related Work on Consistency Checking 79
3.5 Relationship among UML Models 82
3.6 Transformational Approach to Consistency Checking 84
3.7 Design Guidelines 88
3.7.1 Intra-model consistency checking rules 91
3.7.2 Inter-model consistency checking rules 92
3.8 Applying the Transformational Approach 93
3.8.1 Translation of UML model to Database Schema 93
3.8.2 Code Generation 94
3.8.3 Consistency Checking - The Process 96
3.8.4 Prototype tool 98
3.9 Summary 100

4 Managing Test Scenarios 102
4.1 Introduction 102
4.2 Generating scenarios from UML Activity Diagrams 105
4.2.1 Introduction 105
4.2.2 Why Activity Diagrams ? 106

4.2.3 Activity Diagrams and Scenarios 108
4.2.4 Related Work on Scenario Generation using UML Activity Diagrams 111
4.2.5 Approach to Deriving Test Scenarios for Concurrent Activities inside Fork-Join 115
4.2.6 Test coverage criteria 125
4.2.7 Scenario Generation: Summary 126
4.3 Prioritizing Scenarios 128
4.3.1 Introduction 128
4.3.2 Need for prioritization 130
4.3.3 Related Work on Prioritization 131
4.3.4 The Approach 132
4.3.5 Customer Prioritization 134
4.3.6 Prioritizing use cases 138
4.3.7 Combined prioritization of use cases 142
4.3.8 Prioritization of scenarios 143
4.3.9 Prioritization: Summary 152
4.4 Scenario Selection 154
4.4.1 Introduction 154
4.4.2 Related Work on Test Case Selection 156
4.4.3 Selection Criterion 158
4.4.4 Scenario Selection based on Levenshtein Distance 159
4.4.5 Scenario selection based on Common Substrings 165
4.4.6 Clustering based Test Selection 170
4.4.7 Scenario Selection: Summary 175
4.5 Summary 176

5 Ontology-based Scenario Management 178
5.1 Introduction 178
5.1.1 Motivating Examples 179
5.1.2 Ontology for Repository and Reasoning 181
5.2 State of the Art 183
5.3 The Proposed Approach 185
5.3.1 Building an Ontology for testing 185
5.3.2 Design of Ontology 185

5.3.3 Implementation using OWL 190
5.4 Querying the Ontology 194
5.5 Extension of Ontology 195
5.6 Summary 197

6 Results and Prototype Tool 200

6.1 Introduction 200
6.2 Architecture of TestGen 201
6.3 Functionality supported 202
6.3.1 Modules of the System 207
6.4 Case Study 208
6.4.1 Results of the Case Study 210
6.5 Discussion 228
6.6 Summary 230

7 Conclusion 231

7.0.1 Summary 231
7.0.2 Future Work 234
7.0.3 Publications 234

A Rules for Consistency Checking 256

A.0.4 Intra-Model consistency checking rules 256
A.0.5 Inter-Model consistency checking rules 258

List of Figures

1.1 Outline of Thesis 7

2.1 APFD is higher for test case orders that reveal most faults early. . . 39
2.2 Classification of Ontologies 58

3.1 Sequence Diagram for 'Customer places an order' 74
3.2 Fragment of Class Diagram related to 'Customer places an order' . 74
3.3 Representation of Class 'Employee' 79
3.4 Relationship among UML models 84
3.5 Transformational Approach to Consistency Checking 85
3.6 Modeling elements in the UML Metamodel 89
3.7 Metamodel incorporating design guidelines 90
3.8 An example of intra-model inconsistency 91
3.9 An example of inter-model inconsistency 92
3.10 Consistency Checking Process 97
3.11 Architecture of Prototype Tool for consistency checking 98
3.12 Selection of application folder to check for inconsistency 99
3.13 Selecting rules to check inconsistency 99
3.14 List of errors after application of rules 100

4.1 Constructs used in an activity diagram 109
4.2 Activity diagram for "Booking a Package Tour" 110
4.3 Interleaving of activities inside fork-join 118
4.4 Interleaving of activities inside fork-join 123
4.5 Steps in prioritization 133
4.6 Constructs of the Use Case Diagram 139
4.7 Diagram showing various kinds of use case relationships 140
4.8 Steps in Preprocessing 140

4.9 Calculating priority of use cases 142
4.10 Activity Diagram for use case 'Validate User' 145
4.11 Graph representation of Activity Diagram 146
4.12 Conversion to Tree - Branch 147
4.13 Conversion to Tree - Loop 147
4.14 Conversion to Tree - Concurrent Activities 148
4.15 Tree T after assigning node weights 149
4.16 Tree T after assigning edge weights 150
4.17 Tree T showing prioritized paths 152
4.18 Set of Scenarios (i)Graph (ii) 5 out of 9 scenarios generated 163
4.19 Activity diagram showing a. (i) slices b. (ii) - (vii) 6 scenarios in (a) 167
4.20 Diagrammatic representation of clustering 170
4.21 Dendrogram obtained by Clustering 172
4.22 Dendrogram obtained by Clustering - AHC 174

5.1 Entity Relationship diagram for the relation 'Use case has scenarios, and each scenario is made of classes' 180
5.2 Class diagram showing relation among concepts of the ontology . . 189
5.3 The transformation process 190
5.4 Concepts in the ontology 191
5.5 Creating a concept hierarchy 192
5.6 Class diagram showing relation among concepts of the ontology . . 192
5.7 Properties of concepts 193
5.8 Instances belong to concepts 193
5.9 Querying the ontology - an example 194
5.10 Querying an ontology - List all use cases that include a particular use case 195
5.11 Querying an ontology - List all actors related to use cases 'Perform-Billing' and 'Manage User Account' 196
5.12 Querying an ontology - List all actors associated to a particular use case 196
5.13 Class diagram showing relation among concepts of the ontology . . 198

6.1 Architecture of the testing tool 202
6.2 Use case diagram for the testing tool 203

6.3 Testing tool: TestGen - Selecting the Software Under Test 204
6.4 Testing tool: TestGen - Linking Activity Diagram 205
6.5 Testing tool: TestGen 206
6.6 Obtaining priority from the Customer(default value assigned to 1, lowest priority) 207
6.7 Computing Structure based Priority 209
6.8 Use case diagram for the 'CoffeeMaker' 214
6.9 Comparison of APFD values obtained for CoffeeMaker System . . . 215
6.10 Defect detection using Levenshtein distance as distance measure for selection 216
6.11 Defect detection using Levenshtein distance as distance metric for selection-applying different methods for picking one of two scenarios 217
6.12 Defect detection using similarity measure based on common sub-scenarios as distance measure for selection 218
6.13 Selecting one of two similar scenario based on similarity measure . . 219
6.14 Clustering based selection of scenarios 220
6.15 Use case diagram for 'Supermarket Automation System(SAS)' . . . 221
6.16 Comparison of APFD values obtained for SAS 222
6.17 Defect detection using Levenshtein distance as distance measure for selection 223
6.18 Selecting one of two similar scenario based on Levenshtein distance 224
6.19 Defect detection using similarity measure based on common sub-scenarios as distance measure for selection 225
6.20 Defect detection using similarity measure based on common subscenarios as distance measure for selection-applying different methods for picking one of two scenarios 226
6.21 Clustering based selection of scenarios 227

A.1 Diagrammatic representation of Rule 8 257
A.2 Diagrammatic representation of Rule 13 257

List of Tables

2.1 Testing phases and corresponding UML Diagrams 12
2.2 Comparison of work on Consistency Checking 23
2.3 Comparison of approaches for generating scenarios from UML diagrams 30
2.4 Test suite with faults 39
2.5 APFD is higher for test case orders that reveal most faults early. . . 40
2.6 Comparison of prioritization techniques 42
2.7 Comparison of approaches to selecting test cases 52

3.1 Strength of inter-relationships among models 83

4.1 Testing Approaches using Activity Diagrams 112
4.2 Scenarios from activity diagram 'Book Package Tour' 115
4.3 Test Scenarios for 'Book Package Tour applying priority-based selection' 119
4.4 Test Scenarios for 'Book Package Tour applying level-based selection' 122
4.5 Summary: Factors with rankings for successful, challenged and impaired projects[Gro] 135
4.6 Scale for requirements prioritization 137
4.7 Calculated weights for the five paths 151
4.8 Prioritized scenarios 152
4.9 Set of scenarios generated from graph in Figure 4.18 164
4.10 Levenshtein distance calculated between scenarios in Table 4.9 . . . 164
4.11 Distance table after selection of scenario 9 165
4.12 Distance values between scenarios using similarity measure for Figure 4.19 168
4.13 Distance matrix after deletion of scenario 1 168

4.14 Distance calculated between scenarios using metric for Figure 4.19 . 174
4.15 Distance matrix after clustering scenarios 3/4 174

6.1 Sample of Faults 210
6.2 Results in terms of APFD values for CoffeeMaker System 216
6.3 Results in terms of APFD values for SAS 223

Chapter 1

Introduction

The increasing use of software is giving rise to development of highly complex software systems. Further, software systems are required to be of high quality as a defect can have catastrophic effect on business as well as human life. Software development life cycle includes testing with high priority to ensure production of quality software. Testing is defined as the process of executing a program with the intension of finding error[Mye79]. Software testing is an expensive process of the software development life cycle consuming nearly 50% of development cost. Testing is required to verify and validate customer requirements, to validate incomplete/abstract/changing requirements, to uncover errors and build faster and better quality software[Bei02]. There are two strategies to testing, white box and black box. In white box testing, test cases are developed to exercise internal paths and structure of the Software under Test(SUT) thereby ensuring code coverage. Black box testing is done based on requirements and specification. Hence, test cases are built to ensure that software meets user specifications.

Specification refers to the intended behaviour of a system. Specification provides information for testing by focusing on aspects of the system that have to be implemented. Advantage of testing based on specification are better understanding of the objectives and behaviour of the system, early detection of defects, reducing effort and cost in debugging and rectification and easy verification of the system(traceability). Specification of a system can be captured using formal/informal techniques. Formal techniques involve using a formal language like B and Z. The advantage of a formal technique is the mathematical foundation offering means to reason on specification; but, it is difficult for professionals with less incli-

nation on mathematics. Probably, that's the reason why the technique is less used in comparison to UML: the Unified Modeling Language with graphical notations consisting of various diagrams to capture requirements.

UML is a widely accepted standard for modeling software systems. It consists of a set of modeling concepts(primitives) to support an object oriented approach to software development. UML consists of a set of diagrams that model both static and dynamic behaviour of a system. Various aspects of the system are elaborated at different levels of abstraction using diagrams like use case diagram, class diagram, activity diagram, sequence diagram and state diagrams.

The objective of this work is to investigate testing of software systems developed from UML specifications. Smith and Robson [SR92] discuss four levels of testing object oriented systems: the algorithm level, to test methods, the class level, to test interactions between class attributes and functions, the cluster level, to test interactions between classes, and system level that tests the entire system. Jorgensen and Erickson[JE94] in their work divide testing into unit, integration and system level. This work focusses on testing a system to ensure conformance of system requirements.

Testing of software system is critical as well as expensive. There has been attempts to reduce test efforts without compromising the quality of software. Specification based testing is viewed in terms of testing scenarios, where a scenario is a sequence of activities. This work focusses on scenario generation and selection of representative scenarios for testing so that test effort is reduced at the same time not compromising on quality. In the next section, problems investigated and reported in the thesis are detailed.

1.1 Research Problem

Problem 1: An approach to ensure consistency of specification : The inherent overlapping of elements of different UML diagrams used for system specification may give rise to specification inconsistency. Ensuring consistency in UML diagrams is pivotal since the quality of a system is highly dependent on its specification. This problem is being investigated and the resultant method is supported by a tool. The research reported here envisions a rule based approach for consistency checking and a prototype tool is developed and the viability is demonstrated

with case studies.

Problem 2: A technique for test scenario generation from UML activity diagrams : Activity diagrams elaborate scenarios related to each use case. A scenario is defined as the sequence of activities starting from the start activity to the end activity. The number of scenarios generated from activity diagrams using automated generation is exhaustive, especially in case of concurrent activities. Hence, there is need to generate scenarios in an optimized way.

Problem 3: Technique to prioritize scenarios : The order in which test cases are executed has bearing on both testing time as well as test coverage. An ineffective test case ordering could be time consuming to reveal critical faults. This motivates the study of techniques to introduce an ordering of test cases that reveal defects as early as possible while widening test coverage.

Problem 4: Technique for test scenario selection : Given constraints of cost and time, it may not be always possible to run the entire set of test cases. Hence, there is need to introduce techniques that provide a selection of effective subset of scenarios for testing.

Problem 5: Approach for test management : Given the size of software systems, and the constraints of cost and time, there is need for better test management. This thesis proposes an ontology based test scenario management for querying on test scenario repository.

1.2 Contribution

The contributions of this thesis can be summarized as below:
Consistency checker to deal with Problem 1. A methodology that follows transformational approach is proposed for consistency checking. Well Formedness Rules(WFRs), following structural relationship among UML diagram elements are extracted and applied on UML diagram repository for checking. The usage of this method is demonstrated with a prototype tool developed for the purpose.

The benefits of the proposed approach are two fold. One, consistency rules

are defined as Well Formedness Rules which are generic in nature. Second, the transformational approach is a common format to store model elements. Also, the relational model has inherent mechanism to check for inconsistency through primary and secondary keys as well as triggers and assertions which can be used to enforce consistency.

Test Generation technique to deal with Problem 2. Functional requirements are recorded using use case diagrams. Each use case is elaborated using activity diagrams. Thus, each path in an activity diagram from the start activity to the end activity constitutes a scenario. A modified Depth First Traversal algorithm is proposed to generate scenarios from activity diagrams given that the number of scenarios generated could be large, especially in cases where concurrent activities are involved. Dependency information among activities are used to reduce the number of scenarios generated. Concurrent activities are annotated with either priority or level information. So, activities having higher priority are to be performed before activities having a lower priority. Alternatively, all activities at a higher level have to be completed before activities at a lower level.

The benefit of the technique is that based on dependency information, the number of scenarios generated can be reduced. This is especially effective in case of concurrent activities represented using fork/join constructs in an activity diagram.

Test Prioritization technique to deal with Problem 3. Given a large number of test scenarios(test cases), the order of execution gains importance given constraints of cost and time. Defects have to be detected early in the testing process so that feedback can be given at the earliest for bug fixing. In this work, two types of priority are considered: user defined and structural complexity. In the former, use cases are prioritized by the customer according to business needs. Secondly, structural priority due to diagram structure is obtained by considering interactions between primitives of use case diagrams, due to include, extend and generalization features. In case of scenarios, priority of a scenario is calculated by considering the primitives of the activity diagram, namely, activity, fork/join, branch/merge and concurrent activities. The combined priority of the use case and scenario is taken as the priority of the scenario.

The advantage of the technique is that besides customer input on priority of

use case, priority calculation is automated as it is based on the primitives of the UML use case and activity diagrams. It's observed that the priority obtained by the proposed method is realistic as it considers inputs from user as well as domain complexity reflected through specification diagrams.

Test Selection technique to deal with Problem 4. Prioritization provides an ordering of scenarios. However, given constraints of cost and time it may not be possible to test all scenarios. Hence, there is need for selecting scenarios for testing. A close look at scenarios reveal the commonality among scenarios so that testing of common scenarios should be avoided and more dissimilar scenarios are to be selected for speeding the test process and for revealing critical defects early.

In this work, three techniques that use distance measures to determine similarity are presented. The first technique is based on Levenshtein distance[Lev65] as a measure of dissimilarity between scenarios. For each pair, the Levenshtein distance is calculated. The least value indicates scenario pairs that are least dissimilar. The second technique for test scenario selection is based on Longest Common Subsequence. A subscenario is a contiguous set of activities within a scenario. The technique looks at similarity between scenarios in terms of its length and position of the common subscenarios in the scenario. The third technique is based on clustering of scenarios based on their similarity computed by a distance metric like Levenshtein distance. Agglomerative Hierarchical Clustering is used to cluster scenarios based on a certain degree of similarity defined by a metric.

The proposed techniques are comprehensive, well defined by distance metric and algorithmic, so that these are computable and can be automated with ease. They provide a clear cut guideline to choose representative scenarios for testing.

Ontology for Test Management to deal with Problem 5. Management of test scenarios involves ordering and selecting a set of scenarios for testing with the objective of fulfilling a criteria like maximizing coverage or discovering defects as early as possible. For this, there is a need to maintain knowledge on the main activities in the domain and the interactions between them. Ontologies provide a mechanism to share and reason on knowledge that is captured. Ontologies are used in this work to aid in test management.

1.3 Terminology

The terms related to software testing used in this dissertation comply with the Standard Glossary of Terms used in Software Testing V.2.0, Dec, 2nd 2007 produced by the Glossary Working Party International Software Testing Qualifications Board. Some of the most used terms are presented in this section.

code coverage	An analysis method that determines which parts of the software have been executed(covered) by the test suite and which parts have not been executed, e.g. statement coverage, decision coverage.
priority	The level of (business) importance assigned to an item, e.g. defect.
system testing	The process of testing an integrated system to verify that it meets specified requirements.
test scenario	A document specifying a sequence of actions for the execution of a test. Also known as test script or manual test script.
test suite	A set of test cases for a system under test.
test coverage	The degree, expressed as a percentage, to which a specified coverage item has been exercised by a test suite.

Regarding UML and its related terms, the definitions used in OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2 are adopted:

Actor	An actor specifies a role played by a user or any other system that interacts with the subject.
Use case	A use case is the specification of a set of actions performed by a system, which yields an observable result that is of value for one or actors or stakeholders of the system.
Activity	An activity execution is the execution of an activity, ultimately including the executions of actions within it.
Scenario	A sequence of activities from an activity diagram starting from the start activity to the end activity constitutes a scenario.

1.4 Thesis Structure

Organization of the thesis is shown in Figure 1.1. Chapter 2 surveys recent work on software testing with particular reference to consistency checking, test scenario generation, prioritization, selection and ontology generation within the context

of using UML diagrams for specification. Chapter 3 discusses a transformational approach to consistency checking. Techniques for test scenario generation, prioritization and selection is discussed in Chapter 4. In Chapter 5, generation of an ontology for test management is the focus. Chapter 6 presents the tool and experiments conducted to evaluate the performance of the test scenario generation, prioritization and selection technqiues. Finally, Chapter 7 concludes the work and discusses directions for future work.

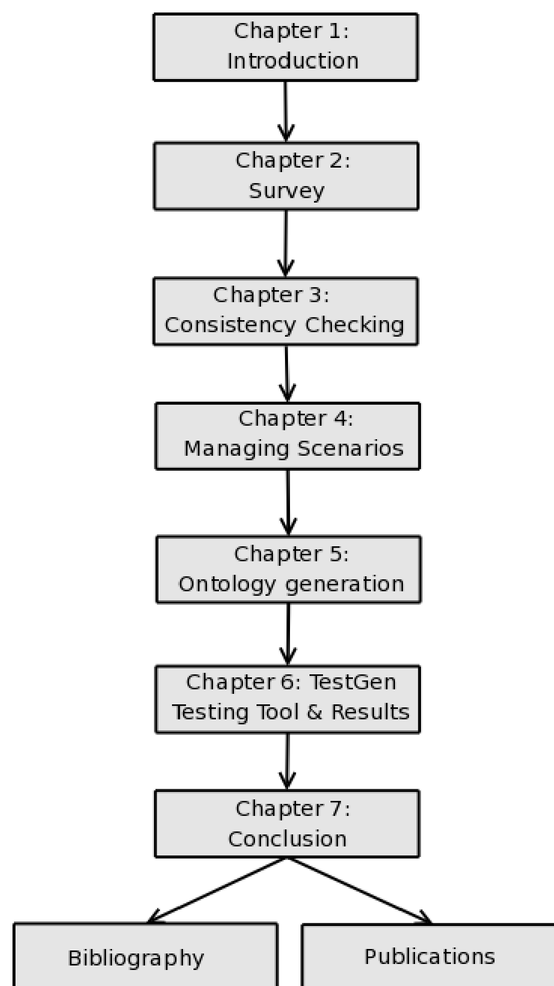


Figure 1.1: Outline of Thesis

Chapter 2

Survey

Specification based testing using UML, its usage, advantages and disadvantages are discussed. Activities related to testing a software system, namely, scenario generation, prioritization and selection are discussed. A taxonomy of different features for scenario generation, prioritization and selection is given. Based on the taxonomy, various works in literature are compared to understand the basis for the definition of current techniques, especially with reference to UML. Also, the state of art in the area of ontologies in software engineering and in particular, UML, is also surveyed.

2.1 Introduction

Testing is the process of executing a program with the intent of finding an error [Mye79]. It shows the existence of a bug in the software. Testing is a necessary activity to verify and validate customer requirements. The objective of testing is to uncover errors in a complete manner with minimum effort and time [Pre05]. Testing can be done either at 'the end', using structured testing methodologies, 'everywhere' by testing at every phase of development, or 'test first' where a test is written first(Test driven development)[Bec02]. Software testing is thus an important activity that encompasses the whole development and maintenance process.

Testing techniques are broadly classified into two: white box and black box. White box techniques are based on internal paths, structure, and implementation of the Software Under Test(SUT). The objective is to check code coverage. Black

box testing on the other hand is based on requirements and specification. Test cases check whether functional requirements are met by the developed software. Also, tests can be designed for a system based on its code (code-based testing) or its specification(specification based testing) or from a model that describes functional aspects of the software(model based testing).

Specification based testing involves testing software using specification to ensure conformance. The Unified Modeling Language(UML) is a standard language for specifying, visualizing, constructing and documenting artifacts of a software system [uml07]. Consisting of graphical notations and a comprehensive set of diagrams and elements, the diagrams capture various aspects of a system from different perspectives or from difference abstraction levels. For example, the use case diagram captures requirements of the system and is understandable to both customers and developers. Besides, both static and dynamic behaviour of a system can be captured using UML diagrams. This work looks at using UML to capture requirements of a system. Specification based testing, its relation with UML and its advantages and disadvantages are discussed in Section 2.2.

One of the main issues when capturing requirements is the mismatch that occurs leading to consistency issues. In case of UML, various aspects of the system are captured through various diagrams and incoherent diagrams lead to inconsistency in specification. Consistency has to be handled at two levels, intra-model and inter-model. Intra-model consistency is concerned with inconsistency within a model. On the contrary, inter-model consistency deals with inconsistency between elements belonging to different models(e.g. between class and sequence diagrams).

Once consistency of UML models is ensured, it can be used confidently for other activities involved in the testing process, namely:

- Define criteria for testing
- Scenario generation

- Scenario prioritization
- Scenario selection

This chapter surveys existing work in the area of consistency checking between UML diagrams, discussed in Section 2.4. For scenario generation, two diagrams, namely, the use case and activity diagrams have been used to capture requirements of the system. A use case captures a requirement which is elaborated using an activity diagram. Activity diagrams are used to generate scenarios belonging to a requirement(use case). Section 2.5 discusses existing work in the area of scenario generation from UML diagrams with specific focus on scenario generation from UML activity diagrams. The number of scenarios generated through an automated approach being exhaustive, there is need to order scenarios to achieve the objective stated for testing like fault detection or code coverage. Section 2.6 compares existing work on test case prioritization.

Given constraints like cost, effort and time involved in software development, exhaustive testing becomes infeasible. Hence, there is need to select a subset of scenarios that is as effective as the complete set for testing to make the process effective. Section 2.7 compares existing work on test case selection.

2.2 Specification based Testing

Requirements refer to the features that a software system should have to meet the user demands pertaining to its business domain. The terminology used to specify requirements may differ from one user to another. In contrast, software specification defines in a complete, precise and verifiable manner, the requirements and behaviour of the system to be built. It is a precise and detailed description of the system's functionality and constraints. Also, the intention is to communicate in clear terms, requirements to developers. Terminology used is related to the

implementation(building the software). A requirements document is used as the reference in building the specification.

2.2.1 Definition

Specification based Testing(SBT) refers to the process of testing a program to check if the implementation conforms to the specification[NPLTJ03]. i.e. testing is done irrespective of the implementation. Also, test cases are built before or in parallel to the actual development of the software. The test suite is developed based on the specification to satisfy a test criterion, like coverage or fault detection. The objective of specification based testing is to check if the developed software meets specification as defined by the customer.

Specification based testing differs from Model based Testing(MBT)[DJK+99]. In the former, specification document is used as the basis for all development activities including testing. i.e. test suite is developed from specification. Model based Testing is similar to specification based testing except that a separate model is built specifically for the purpose of testing, using specification as basis. This model therefore represents some aspects of the specification and is used as basis for testing only. The advantage of the former lies in the fact that specification is used to derive the test suite ensuring that software matches customer requirements. It is the reference point for test case generation. In case of the latter, the model is derived from the specification and is built separately. This may cause defects to be introduced in the model due to wrong understanding and misinterpretation of specification. Also, additional effort is involved in building the model[OXL99].

2.2.2 UML and Specification based Testing

The Unified Modeling Language is a general-purpose visual modeling language used to specify, visualize, construct, and document artifacts of a software system [uml07]. Developed and propagated by the OMG group, UML can be used across all phases of the software development process (requirement, analysis & design, testing and documentation).

One or more diagrams can be used to represent the system. UML models can be classified as static models and dynamic models. Static models represent the structure of the system, whereas dynamic models are used to represent the behaviour of the system. Thus, a combination of the models may be used to suit the type and domain of the software to be developed. A UML diagram is not refined to provide all relevant aspects of an application. The semi-formal nature of UML leads to ambiguities in representation and interpretation of stated requirements. To overcome this, the Object Constraint Language(OCL) is used to write constraints on model elements. OCL expressions are used to specify invariants on classes, define pre- and post conditions on operations and methods, describe guards, constraints on methods as well as specify derivation rules for attributes for an expression over a UML model [uml07]. Hence, OCL is used along with UML to make up for the lack of formalism.

Table 2.1: Testing phases and corresponding UML Diagrams

Type of Testing	**Coverage Criteria**	**UML Diagram**
Unit	Statement,Class	Class & State diagrams
Integration(Feature Testing)	Between Classes & Between Modules	Class diagrams & Interaction diagrams
System	Usage Scenarios	Use case & Activity diagrams
Regression	Usage Scenarios	Use case & Activity diagrams

UML can be used at various levels of testing, namely, unit, integration, system and regression testing. Table 2.1 shows the different testing levels as well as the UML diagrams that can be used for testing at the level with the coverage criteria [Wil99]. Specification using UML can be used to design functional tests for the SUT. It helps in software testing by providing information required to generate test cases, select test cases, measure test adequacy as well as check correctness of output. Besides, the test cases can be created in parallel with the development process and can be used when the software is developed. Besides, inconsistencies and ambiguities in specification can be identified, which can be resolved early saving time and cost.

2.2.3 Advantages and Disadvantages of Specification based Testing using UML

UML as a modeling language has both advantages and disadvantages [CDJ+02, OXL99]. Advantages of specification based testing using UML include:

1. *Ease of understanding requirements.* Requirements represented using UML are easily understandable by the stakeholders of the system when compared to other modelling approaches like FSM, Petrinets and formal methods like Z and B due to its semi-formal nature.

2. *Scalability.* Software is subject to change as requirements change or are added throughout the software life cycle. UML models are easy to change and scale as the software grows.

3. *Specification document as reference.* Specification is used as the reference point across all phases, namely, analysis and design, testing and documentation.

4. *Customer viewpoint.* Testing must be done from the customer's viewpoint, keeping in mind the need of the customer.

5. *Independent test generation.* Testing activity is independent of influence as the development and test team work independently though using the same specification document.

6. *Tools.* A large number of tools exist for creating and editing UML models.

Disadvantages of using UML as a specification language:

1. *Nature of UML.* UML is a semi-formal modeling language. Hence, it leads to ambiguities in representation and misinterpretation. Also, the semantics of communication within UML is only partially defined.

2. *Choice of diagrams.* UML provides a number of diagrams and the choice of diagrams is an important factor.

3. *Creating models.* The language provides features that can be used to create models of deceptive complexity.

2.3 Usage of UML

Semiformal modeling languages are a powerful means of describing requirements [Gli00]. UML has been accepted as the standard for object-oriented analysis and design and finds use in modelling requirements and design of a system through the various diagrams. UML consists of 13 diagrams, both static and dynamic, with different diagrams conveying different information. A survey of UML usage [DP06, DP08] finds class diagrams, sequence diagrams, use case diagrams, use case narratives, activity diagrams, statechart diagrams and collaboration diagrams(in order) to be the most used of UML diagrams.

Different diagrams have been used to describe system functionality as they convey different information. Also, usage of diagrams depend on the ease of understanding and level of detail that a diagram can represent. Use case narratives, activity diagrams and use case diagrams find highest usage in verifying and validating requirements with clients while class, sequence and use case narratives are used more in specifying requirements for programmers [DP06, DP08]. For communication among technical team members, sequence and class diagrams find highest usage whereas class diagrams are used more to document for future maintenance and enhancements. Also, the style and rigor in modelling using UML depends on factors like analyst's knowledge, client requests, time constraints and system domain [DP08].

Some UML diagrams find minimal usage in software projects. The reasons include[DP08]:

- not well understood by analysts
- not useful for most projects
- insufficient value to justify cost
- redundant capture of information
- not useful with clients
- not useful with programmers

User/client involvement in various phases of system development has been considered a crucial factor in the success of projects. UML diagrams find varied usage in development, review, and approval activities where customer involvement is high. Use case narratives, use case diagrams and activity diagrams find highest usage where client involvement is essential. According to Dobing et al [DP08], development, review and approval of use case narratives and use case diagrams

are activities clients were actively involved with. Also, activity diagrams are the easiest for clients to understand besides use case narratives and use case diagrams.

The findings show that class diagrams are the preferred UML model used by project teams to communicate requirements whereas collaboration diagrams are least used[DP06, DP08]. Also, the survey by Dobing et al supports a use case-driven approach due to the involvement of users in establishing requirements of a system. Besides, the finding also suggest that user/client involvement extends beyond use cases. With respect of testing, the use of UML models for testing is still limited due to uncertainty regarding productivity impact of using UML for testing. This maybe due to the fact that use of UML models for creating testing plans is not a common practice [NC08].

Thus, UML diagrams find varied levels of use in development of software. Also, only a subset of UML diagrams find widespread use in practice. Another interesting finding is the degree to which UML diagrams are used in development, review and approval activities that involve clients. This level of involvement indicates that UML is not solely used by software professionals and strengthens the view that it facilitates communication between users and analysts which is crucial for the success of a software product. According to Nugroho et al [NC08], majority of respondents find UML helpful in making design, analysis and implementation activities productive. However, respondents of the survey were uncertain about the productivity impact of using UML on testing with only 38% considering UML helpful in testing and maintenance activities. However, they conclude that the use of UML models has to be considered for testing as they will provide a simpler, structured and more formal approach to the construction of functional testing and non-formal specification. Also, the development of test cases from specification has the additional benefit of producing test cases before implementation.

2.4 Checking Consistency of UML diagrams

The Unified Modeling Language, a standard for modeling of object oriented software consists of a collection of diagrams that are loosely-connected [uml07]. It has become the standard for object-oriented modeling and is being widely used due to the different diagrams it provides. For example, structural characteristics can be depicted using class diagrams and dynamic behaviour can be depicted using interaction diagrams and activity diagrams. UML diagrams can be enriched with assertions and constraints written in OCL [ocl06], a specification language. Problems creep into software due to its size, complexity and collaborative development. Further plurality of UML diagrams add to complexity in system development. Factors like omission, multiplicity of stakeholders, addition of new features as the system evolves, interdependency of UML diagrams and overlapping of model elements lead to the problem of inconsistency in specification. Therefore, a need for consistency management arises. Consistency management is a process comprising activities like consistency identification, consistency handling(tracking and removal of anomalies) and management of policies for consistency checking [ZK01].

UML models are semantically rich. A system is modeled with different types of models, namely, use case model, class model, sequence model. Hence, it is possible to model the requirements of a system elaborately. Still, there is no guarantee for the correctness, completeness or consistency of the models. Models are said to be consistent when they are coherent with one another with reference to the requirement for which modeled. Reliability of design needs models that are coherent. Erroneous models have a huge impact on the development process in terms of added cost, time and effort.

UML CASE tools designed for a wide variety of users which include Software Engineers, System Analysts, Design Engineers and Test Engineers are meant for building large scale software systems using the Object Oriented Approach. Tools

like IBM Rational and VP for UML(VP-UML) form part of this. Built with the purpose of providing an easy to develop interface for UML diagrams and model elements, these tools can also be used to check consistency of diagrams and generate code from specification. Most UML tools provide basic consistency using model elements by building a class repository that stores classes and other model elements usable across UML diagrams as in VP-UML. IBM Rational follows an approach where actors, use cases and scenarios(sequence diagrams) are built in order. Consistency is maintained by relating every instance(object) to a class. Rational also allows multi-user mode that helps multiple users work on the same model. Here, when an object element is deleted from a model, all references to it are removed to maintain consistency ensuring consistency of model elements. Also, some intra-model consistency rules are defined using the Object Constraint Language(OCL) and written as Well-formed Rules in the UML superstructure document. A few of these rules are implemented in some of the conventional CASE tools. However, inter-model consistency rules are not defined in the UML superstructure document by the OMG.

Classification of consistency techniques is discussed in Subsection 2.4.1. Review of literature is given in Subsection 2.4.2 followed by a discussion in Subsection 2.4.3.

2.4.1 Classification of consistency techniques

Consistency techniques can be classified based on:

- **Level :** Consistency between UML diagrams can be checked at the intra-model or inter-model level. Intra-model level is concerned with checking consistency within model elements. For example, checking consistency of elements within the class diagrams is intra-model. Inter model consistency is concerned with the way each model is designed with reference to the other

models so as to be meaningful.

- **Type :** There are two perspectives to consistency: syntactic/structural consistency, and behavioural consistency. Syntactic consistency requires that a model conforms to its abstract syntax (specified by its metamodel). Semantic consistency requires that models' behaviour be semantically compatible. It is essential that models be syntactically and structurally consistent before they can be checked for behavioural consistency. Also, there are two ways of checking consistency: active and passive. Active consistency checking is done as the diagrams capture requirements thereby indicating inconsistency in real time. They monitor and control models edition for preventing inconsistencies[dFBRG06]. However, the disadvantage is that strict enforcement of consistency limits the modeler's possibilities for exploring conflicting or tradeoff solutions[dFBRG06]. Thus, it may be an impediment to the process of capturing requirements. Passive consistency checking involves checking all or a subset of models capturing requirement. The advantage of passive consistency checking is that rules guiding consistency are applied at all models and results obtained. E.g. IBM's Rose model checker uses activity consistency checking.

- **Approach :** Consistency checking is done on both static and dynamic diagrams using one of the following approaches: a) Direct approach b) Transformational approach. The first approach uses the constructs of UML and OCL to check for inter model consistency. The transformational approach involves transforming one model to another or to a common notation before checking for consistency.

- **Dimension :** UML diagrams must be consistent vertically and horizontally. A model consists of different submodels because a system is modeled

from different viewpoints allowing different aspects to be captured in varied models. However, different viewpoint specifications must be consistent. This type of consistency problem is called horizontal consistency. Also, models are used to record specification at different levels of abstraction. A model can be transformed into another model by replacing with one or more submodels. Then, the replaced submodel must be a refinement of a previous submodel. This type of consistency problem is called vertical consistency.

2.4.2 Literature Review

The objective of the literature review is to understand:
'What are the existing work in inconsistency management, what problems have been handled using current approaches, modelling language used and the solutions provided thereof ?' To answer the question, the following set of specific questions have been framed:

1. **Q1 - Technique :** What is the technique used to check consistency of specification captured using UML ?

2. **Q2 - Version :** What is the version of UML used to record specification of the system ?

3. **Q3 - UML Diagrams :** What kind of UML diagrams have been used in the approach ? UML consists of 13 different diagrams. This question tries to find out what diagrams are mostly used to capture specification of the system.

4. **Q4 - Formal Method :** Is a formal method used for consistency checking ? The advantage of using a formal method lies in the precise and unambiguous

nature which helps avoid inconsistency problems faced in the interpretation of a modelling language like UML.

5. **Q5 - OCL :** Is the Object Constraint Language used ? UML is a declarative language used to describe rules that apply to UML models. As UML diagrams cannot depict all aspects of a specification, OCL is used.

6. **Q6 - Approach :** What is the approach used to check for inconsistency, direct or transformational ? The former uses the constructs of UML and OCL to check for consistency whereas the transformational approach involves transforming one model to another or to a common notation before checking for consistency.

7. **Q7 - Level :** At what level is consistency checked ? Consistency can be checked at both intra-model and inter model level. Intra model involves checking for consistency between diagrams of the same type (e.g. class diagrams) whereas inter model involves checking for consistency between different diagrams (e.g. class diagram and sequence diagram).

8. **Q8 - Type :** What aspects of consistency are checked in the work - structural or behavioural ?

9. **Q9 - Automated Support :** Is a prototype tool developed to faciliatate the use and learning of the approach ?

10. **Q10 - UML Diagrams :** Has integration of approach been done with CASE tools ? Integration with CASE tools aid designers to use the approach with existing modelling tools.

The objective of the review is to obtain:

- Information regarding state-of-the-art with reference to consistency management of specification using UML

- Understand the strength and weakness of current techniques and directions for future work

Section 2.4.3 discusses the results of the review and Table 2.2 provides a comprehensive view of the review.

2.4.3 Discussion

Based on the survey(refer Table 2.2) the following conclusions can be made:

UML version. Most of the approaches presented, use a simple[Kri00, MSS, Inv01, KC04], basic[PAM99, Egy01, Der02] or restricted version[KFdB+04, BV06] of UML consisting of main primitives i.e. basic features. Also, most work use version 1.x[RS00, Egy01, HHKT02, RW02, HHS02, CPC+04] of UML. UML 2.0 brought in significant changes compared to UML 1.x. Activity diagrams were delineated from state diagrams and additional primitives were added. Another factor influencing use of UML models is the understanding and knowledge in UML of users including architects, designers and developers[DP08].

Technique. Most work use a formal method to handle inconsistency[HK03, KC04, MA, YS06, Shi06]. A model checker like SPIN[1] is used to simulate behaviour and check for inconsistencies. The advantage of using a formal approach being mathematical for precise and unambiguous representation of requirements. Formal techniques though used in work in literature find limited use in industry due to lack of expertise among practitioners. Besides, the cost of training personnel, difficulty in use and the time involved in applying a formal technique, limitation in terms of feedback and the difficulty in understanding by non-experts stand as barriers to their use.

OCL. The Unified Modeling Language is a collection of diagrams that is

[1]Simple Promela Interpreter(SPIN) is a tool for verifying correctness of distributed software models. Available at http://www.spinroot.com.

Table 2.2: Comparison of work on Consistency Checking

Ref#	UML Ver	Technique	UML Diagrams: UC	SD	AD	CD	STD	CLD	Other	FM	OCL	Appr: Trf	Dir	Level: Intra	Inter	Type: Struc	Beh	AS	CASE
[PAM99]	B	Z,CLIPS	×	✓	✓	✓	✓	×	×	✓	×	✓	×	×	✓	✓	×	×	×
[Kri00]	S	PVS	✓	✓	✓	✓	×	×	×	✓	×	✓	×	×	✓	×	✓	×	×
[RS00]	1.1	ORDBMS	×	×	×	✓	×	×	×	×	✓	✓	×	✓	×	✓	×	✓	×
[Gli00]			✓ (T)	×	×	✓	×	×	✓(UCN)	×	×	?	?	✓	×	✓	×	×	×
[SM00]		Object Petrinets	×	×	×	✓	✓	×	×	✓	×	✓	×	×	✓	✓	×	×	×
[MSS]	S	Description Logic	×	✓	×	✓	✓	×	×	✓	×	✓	×	×	✓	✓	×	×	×
[LLH01]	2.0	CSP	×	✓	×	✓	✓	×	×	✓	×	✓	×	×	✓	×	✓	×	×
[XLKB04]	xUML	S/R COSPAN	×	×	×	×	×	×	×	✓	×	✓	×	?	?	?	?	×	×
[GKW01]		Activity Graphs	✓	×	×	✓	×	×	×	×	×	✓	×	×	✓	✓	×	×	×
[ZK01]		Knowledge based	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓	×	×	×	×
[Inv01]	S	SPIN	×	✓	×	×	✓	×	×	✓	×	✓	×	×	×	×	✓	×	✓
[Egy01]	1.3(B)		✓	✓	✓	✓	✓	×	×	×	✓	✓	×	×	✓	✓	✓	×	×
[HHKT02]	1.3	OCL	×	×	✓	✓	✓	×	×	×	✓	✓	×	×	✓	✓	×	×	×
[Der02]	B	Object Z,LOTOS	×	✓	×	✓	✓	×	×	✓	×	✓	×	×	✓	×	✓	×	×
[BDM02]		Petrinet	×	✓	×	×	✓	×	×	✓	×	✓	×	×	✓	✓	×	×	×
[RW02]	1.5	Object Z,CSP	×	×	×	✓	✓	×	×	✓	×	✓	×	×	✓	✓	✓	×	×
[LCA02]			×	×	×	✓	✓	×	×	×	×	×	✓	×	✓	×	×	×	×

✓ - means use of technique/method and × indicates absence of technique/method; ? - Not known to the author; A - Specification annotated with details; C - UML used as base model and then converted to another; B- Basic UML model, S = Simple UML model, R - restricted model of UML, CTL - Computation Tree Logic, CSP - Communication Sequential Process, PVS - Theorem prover, SPIN - Model Checker; A blank indicates that data could not be gleaned

Comparison of work on Consistency Checking (Contd/-)

	UML		UML Diagrams									Appr		Level		Type			
Ref#	Ver	Technique	UC	SD	AD	CD	STD	CLD	Other	FM	OCL	Trf	Dir	Intra	Inter	Struc	Beh	AS	CASE
[HHS02]	1.4(B)	Graphs	×	×	×	✓	×	×	✓(OD)	×	×	✓	×	✓	×	✓	×	×	×
[LTY03]	BVUML		×	✓	×	×	✓	×	×	×	×	×	✓	✓	×	×	×	✓	×
[HK03]	1.4(B)	CPN	×	✓	✓	✓	✓	×	×	✓	×	✓	×	×	✓	✓	✓	×	×
[CPC+04]	1.3	OCL	×	×	×	✓	×	×	×	×	✓	✓	×	✓	×	✓	×	×	✓
[KC04]	S	Object Z	×	×	×	×	✓	×	×	✓	✓	✓	×	×	✓	×	✓	×	×
[HS04]		CPN	×	×	×	×	✓	✓	×	✓	×	✓	×	×	✓	✓	×	×	×
[MA]	2.0	Maude Spec	×	×	×	✓	×	×	✓	✓	×	✓	×	×	✓	✓	×	×	×
[KFdB+04]	1.5(R)	PVS	×	×	×	✓	✓	×	✓	✓	✓	✓	×	×	✓	✓	×	✓	×
[LBMS05]	1.5	Rule based	×	✓	×	✓	×	×	×	×	✓	✓	×	×	✓	✓	×	✓	×
[YS06]	S	Petrinet-ECPN	×	✓	×	×	✓	×	×	✓	×	✓	×	×	✓	×	✓	×	×
[ZLQ06]	S	Split Automata	×	✓	×	×	✓	×	×	✓	×	✓	×	×	✓	×	✓	✓	✓
[BV06]	1.5(R)	Rule based	✓	✓	×	×	×	×	×	×	×	×	✓	×	✓	✓	×	✓	×
[Shi06]	2.0	Petrinet CPN	✓	✓	✓	×	✓	×	×	✓	×	✓	×	×	✓	✓	×	×	×
[Egy07]	1.3(B)	UML Analyzer	×	✓	×	✓	✓	×	×	×	×	×	✓	×	✓	✓	✓	✓	✓
[ZGG07]		OCL	✓	×	×	✓	×	×	×	×	✓	✓	×	×	✓	✓	×	✓	✓
[SBHKR08]	2.0(B)	Knowledge Based	×	✓	✓	✓	✓	×	×	×	×	✓	×	×	✓	✓	✓	×	✓

✓ - means use of technique/method and × indicates absence of technique/method; ? - Not known to the author; A - Specification annotated with details; C - UML used as base model and then converted to another; B- Basic UML model, S = Simple UML model, R - restricted model of UML, CSP - Communication Sequential Process, CPN - Colored Petri Nets, ECPN - Extended Colored Petri Nets; A blank indicates that data could not be gleaned

loosely-connected[uml07] and is used to represent various aspects of a system through different diagrams. However, UML diagrams cannot by themselves provide all relevant aspects of a specification. Hence, Object Constraint Language (OCL)[ocl06], a specification language, was developed to describe additional constraints about objects in the model that cannot be expressed using standard diagrammatic notations. The advantage of using OCL is that it helps write guard conditions, constraints, pre- and post conditions on model elements[CPC+04, KC04, KFdB+04, Shi06, ZGG07].

Diagrams. Consistency checking using class, state diagrams and sequence diagrams is the focus of most work due to its widespread use[MSS, LLH01, Der02, HK03, SBHKR08]. Activity diagrams and use case diagrams are considered by relatively fewer works[Kri00, Egy01, Shi06]. Interestingly, collaboration diagram which is similar to sequence diagram is used in only one work[HS04]. In the table, 'Other' relates to use of object diagrams by [HHS02]. Use case narratives(text) has been used along with use case diagrams in [Gli00]. The lesser use of activity diagrams may be due to the fact that in version 1.x, activity diagrams were considered similar to state diagrams. Also, there is very few work that involves use case diagrams showing the lack of connect in UML between use case and other diagrams.

Approach. Most work in literature use a transformational approach to check for inconsistency using either formal methods like Object Z[RW02, Der02, KC04], model checkers like PVS[Kri00] and SPIN[Inv01], Petrinets[SM00, BDM02, Shi06, YS06], graphical approaches[GKW01, HHS02], writing rules[BV06, LBMS05] or by representing in a knowledge base[SBHKR08]. Direct approach is used by [LCA02, LTY03, BV06].

Level. Consistency checking can be done at two levels, intra model(for e.g.[RS00, Gli00, HHS02, CPC+04]) and inter model (for e.g.[Kri00, SM00, MSS, LBMS05,

Shi06, Egy07, ZGG07, SBHKR08]). Most work in literature involve two or more diagrams and check for consistency between information represented in constructs of the diagrams. However, some work exist where intra-model consistency checking is done as either only one diagram is considered or the specification is captured using one diagram and transformed into another before checking for consistency. Checking consistency of specification within a diagram is easier. However, UML diagrams capture various aspects of the specification and hence, it is not possible to completely transform aspects captured in one diagram to another.

Type. Two aspects, namely, structural(e.g. [Gli00, HHS02, Egy07, ZGG07, SBHKR08]) and behavioural(e.g. [Kri00, Inv01, HK03, KC04, YS06, ZLQ06]) can be checked when handling consistency between diagrams. Different work in literature handle structural and behavioural aspects.

Automated Support and CASE Integration. Some work provide CASE integration[Inv01, CPC+04, ZLQ06, ZGG07, SBHKR08] thereby making it possible for designers to use the techniques to check consistency of UML models. In other cases, automated tools[RS00, LTY03, KFdB+04, LBMS05, ZLQ06, BV06, ZGG07] in the form of prototypes have been developed to show workability of the technique.

Thus, survey shows that consistency checking is an issue when capturing specification using UML diagrams despite substantial work in the area. Different means to handle consistency have been explored, namely, variation in technique being used, approach followed, level at which consistency is checked, type of diagrams used as well as aspects of consistency being handled. UML defines constraints as well formedness rules, applicable at the intra-model level and not at the inter-model level. Navigation between metamodels of all diagrams is not currently present, and hence it is not possible to check for consistency between models in a wholesome manner.

Work in literature handle inconsistency between UML diagrams based on interaction specified between models specified in the superstructure specification[uml07]. UML defines constraints on models as well formedness rules(WFRs). The well formedness rules are applicable at the intra-model level and not at the inter-model level i.e. they do not suffice in detecting inconsistencies, especially those across diagrams. Hence, additional constraints are required to ensure consistency and completeness to help detect and prevent errors during specification of a system.

2.5 Test Scenario Generation

Scenarios repesent the sequence of events in a software system and defines a systems behaviour. Use case, activity, sequence and collaboration diagrams can be used to represent scenarios. Each scenario can be said to represent a requirement goal of the system. In practice, generation of scenarios is mostly done manually making it labor-intensive and error-prone. Hence, there is a need to generate test scenarios to obtain test cases reasoning on test adequacy to achieve desired quality software [Mat07]. In this regard, automation of test scenario generation gains importance. This section discusses test scenario generation from UML diagrams.

Scenarios can be generated from use case diagram [GEMT06], sequence diagrams [FL02] and activity diagrams [MXX06]. The difference lies in the level of details present in each diagram. Diagrams used at various levels of testing have already been shown in Table 2.1. Table 2.3 shows various work in literature that generates scenarios from UML diagrams.

2.5.1 Literature Review

The objective of the literature review is to understand:

What is the state of work on scenario generation from UML diagrams, technique

used, whether consistency of diagrams is ensured before generation, is any transformation methodology used before generation, are diagrams additionally annotated and whether CASE integration is done to aid the process ?

In this section, various features used for comparing existing work on scenario generation are presented. The features give an idea of the techniques used for scenario generation from UML diagrams. To answer the question, the following set of specific questions have been framed:

1. **Q1 - Model used :** What is the UML model used to generate scenarios ? The scope of this work is confined to using UML diagrams as the basis for software development.

2. **Q2 - Testing Level :** What is the testing level for which scenarios are generated ?

3. **Q3 - Version :** What is the version of UML used ?

4. **Q4 - Consistency Checking :** Do approaches consider consistency checking as a factor to be performed prior to scenario generation ?

5. **Q5 - Annotation :** Is additional annotation of UML diagrams done to aid generation of scenarios ?

6. **Q6 - Transformation :** Is some transformation done before generating scenarios ?

7. **Q7 - Automated Support:** Is a prototype tool developed to faciliatate the use and learning of the approach ?

8. **Q8 - Case Integration :** Has integration of approach been done with CASE tools ? Integration with CASE tools aid designers to use the approach with existing modelling tools.

2.5.2 Discussion

Based on the survey of approaches to generate scenarios(refer Table 2.3) the following conclusions can be made:

Usage of UML Models. UML consists of a set of diagrams, both static and dynamic. One or a combination of the diagrams can be used to capture specification. The same is used to generate scenarios for the purpose of testing. UML usage can be limited to one diagram(e.g. sequence/activity diagram) or can be a combination of diagrams(e.g. use case and activity diagrams). In case of system testing, use case and activity diagrams are used as the basis for generation of scenarios(refer Table 2.1). Also, the type of diagram used indicates the level of testing it aims at. For example, state charts are mainly used for unit testing whereas sequence and activity diagrams are used for subsystem and system testing.

Testing Levels. Testing levels for object oriented systems is classified as: algorithmic level, class level, cluster level and system level[Bin99]. Algorithmic level refers to testing a method[BHR+00, DL04]. Class level (otherwise unit testing) refers to testing methods related to a class[YHS+99, HIM00, TSYP03, GLM04]. Cluster level testing(integration testing or subsystem testing) involves testing a collection of classes[HIM00, LJX+04, LL05a, XLL05, Seo06, MXX06, XLLP08]. System testing involves generating scenarios for testing functionalities of the system[LL05b, LS06, CNM07, BNM08, KK09b]. UML diagrams can be used for testing at various levels(refer Table 2.1).

Version. Version of UML used is an indication of primitives used in modeling UML diagrams for test generation. Most work consider core UML with basic features whereas some work look at advanced features of UML diagrams.

Consistency Checking. Consistency checking is the process of checking if the model/diagram is consistent with reference to elements within the model and other models associated with it. Checking for consistency before scenario

Table 2.3: Comparison of approaches for generating scenarios from UML diagrams

Reference#	Source-UML diagram	Testing Level	Version	Consistency Checked	Additional Annotation	Intermediate Transformation	Automated Support	Case Integration
[CLSC98]	State	System		×	✓	×	×	×
[OA99]	State	System		×	×	×	✓	✓
[YHS+99]	State	Class		×	×	✓	×	×
[HIM00]	Use case, Activity,	Class, cluster,	2.0	×	×	×	✓	✓
[AO00]	Collaboration	Cluster	1.3	×	×	×	×	×
[BHR+00]	State	Algorithmic		×	×	✓	✓	×
[CDJ+02]	Class, Object, State	Class	1.4	×	×	×	✓	×
[CTF01]	State	Cluster		×	×	×	×	×
[FL02]	Sequence	System		×	×	×	✓	×
[RPG03]	Use case, State	System		×	×	✓	✓	×
[BDZ03]	Sequence diagram	System		×	×	✓	✓	×
[OLAA03]	State	System		×	×	×	✓	✓
[TSYP03]	Class, State	Class	Simple	×	×	✓	×	×
[KR03]	State, Sequence	Class, System	1.3	×	×	×	×	×
[JH03a]	Use case, Sequence	System		×	×	✓	✓	×
[GLM04]	State	Class		×	×	×	×	×
[DL04]	State	Algorithmic		✓	×	×	✓	×
[LJX+04]	Activity	Cluster, System	1.5	×	×	×	✓	×
[CLL05]	Activity	System	2.0	×	×	✓	✓	✓
[LL05a]	Activity	Cluster, System	1.x	×	×	✓	×	×
[XLL05]	Activity	Cluster, System		×	×	✓	✓	×
[LL04]	State	Cluster		×	×	✓	✓	✓

✓ - means use of technique/method and × indicates absence of technique/method; Blank indicates that information could not be gleaned

Comparison of approaches for generating scenarios from UML diagrams (Contd/-)

Reference#	Source-UML diagram	Testing Level	Version	Consistency Checked	Additional Annotation	Intermediate Transformation	Automated Support	Case Integration
[BS05]	Activity	System	2.0	×	×	✓	✓	×
[LL05b]	State	System		×	×	✓	✓	×
[GEMT06]	Use case narratives	System	2.0	×	×	✓	✓	×
[LS06]	Sequence	System	1.0	×	×	✓	×	×
[Seo06]	State	Class, Cluster		×	×	✓	✓	×
[Sok06]	Sequence & State	Class, Cluster	2.0	×	×	×	×	×
[JW06]	Use case	Acceptance	2.0	×	×	×	×	✓
[DTGF06]	Class, Sequence	Cluster, System		×	×	✓	×	×
[MXX06]	Activity	Cluster, System	2.0	×	×	✓	✓	×
[JW07]	Use case charts	System	2.0	×	×	✓	✓	×
[PAK+07]	Class, Sequence	Cluster, System		×	×	✓	×	×
[CQX+07a]	Activity	System	2.0	×	×	✓	✓	×
[XLL07]	Activity	Cluster, System	2.0	×	×	✓	✓	×
[KKBK07]	Activity	Cluster, System	2.0	×	✓	✓	×	×
[SMK07]	Communication	Cluster		×	×	✓	✓	×
[CNM07]	Sequence	System		×	×	✓	✓	×
[BL01b]	Activity, Sequence	System		×	×	✓	✓	×
[BNM08]	Activity	System		×	×	×	✓	×
[XLLP08]	Activity	Cluster, System	2.1	×	×	✓	✓	×
[CMK08]	Activity	System	2.0	✓	×	×	×	✓
[KK09b]	Activity	System	2.0	×	×	✓	×	×

✓ - means use of technique/method and × indicates absence of technique/method; Blank indicates that information could not be gleaned

generation ensures quality of test scenarios used for testing[CMK08].

Additional Annotation. UML diagrams by themselves may not be able to convey all details about the system. Hence, there may be need to annotate UML diagram constructs as an aid to better understanding as well as ease of use in the automation process. Input Output Activity Diagrams(IOAD) are used in [KKBK07] to aid the process of generating test cases by annotating each activity in the activity diagram with input and output thereby helping in automation of the test process. The drawback is the overhead involved in annotating each activity. However, this overhead is useful in automating the process of test case generation.

Transformation. Specification captured using a model can be used directly for generation of scenarios[HIM00, CDJ+02, CTF01, FL02, RPG03, XLL05, LL04] or be transformed to an intermediate form[KKBK07, SMK07, CNM07, PAK+07, BL01a, XLLP08, KK09b]. The objective of the transformation is one, to transform into a form amenable to generation or two, to use common elements especially in cases where two or more models are concerned. The drawback is the additional overhead involved in transforming one model to another. Most work in literature transform sequence and activity diagrams into graphs and trees to aid the process of scenario generation.

Automated Support and CASE Integrations. Most work in literature provide prototype tools that exhibit the advantages involved in automated generation of scenarios[MXX06, JW07, CQX+07a, XLL07, BL01a, BNM08, XLLP08]. Chevalley et al[CTF01] and Kansomkeat et al[KR03] integrate their techniques with Rational Rose used for modeling.

Survey shows that work on scenario generation based on UML diagrams looks at generating scenarios for various levels of testing. One or a combination of diagrams are used, either directly or by transforming it to aid generation. Most work do not check consistency of diagrams or assume that consistency is checked

before scenario generation. Use case, sequence and activity diagrams are most used for generating scenarios at the cluster and sytem level. The number of scenarios generated using automated techniques is exhaustive. Hence, there is need to use techniques that optimize the number of scenarios generated using some heuristic.

2.6 Prioritization

Testing is an expensive phase of the software development life cycle accounting for a large proportion of the software development effort. It has a major impact on the quality and reliability of delivered software. The objective of testing is to check whether a software meets the requirements of its customer. The size and complexity of software makes exhaustive testing impossible. Hence, testing must be done in a judicious way keeping in mind factors like, customer requirements, cost and time. For this, there is a need to develop test cases and exercise them to gain maximum throughput.

The objective of test scenario prioritization is to order test cases in such a way that a reasonable ordering of test cases can to done to maximize an objective function like rate of fault detection, rate of code coverage, rate of increase of confidence in reliability, rate of fault detection for specific code changes [GRCJ99, EMR00]. Rothermel et al [RUCH01] give a formal definition:

Given: T, a test suite; PT, the set of permutations of T; f, a function from PT to the real numbers.
Problem: Find $T' \epsilon$ PT such that

$(\forall T'')(T'' \epsilon PT)(T'' \neq T')[f(T') \geq f(T'')]$.

Here, PT is the set of all possible orders of T, and f is an objective function

that, applied to any such order, yields an award value for that order. Prioritization of test cases helps in early detection of bugs, inturn helping improve quality of software.

The growing size and complexity of software makes it difficult to test software adequately, especially if the desired level of reliability involves demanding adequacy criteria. Besides cost and effort, time becomes an important factor. The testing team should be able to perform testing in an optimized way, so as to provide early and effective feedback to the development team. To meet this objective, test cases are developed and executed according to a ranking based on their relative importance. Test case prioritization aims at ordering test cases to maximize an objective function like rate of fault detection, rate of code coverage, rate of increase of confidence in reliability, rate of fault detection for specific code changes [GRCJ99, EMR00]. Prioritization of test cases helps in early detection of defects, thereby improving the defect detection-bug fix cycle.

2.6.1 Classification of prioritization techniques

Existing prioritization techniques have been classified based on the level of testing they apply to as well as the methodology used.

1. System testing or Regression testing [RUCH01]

   Test case prioritization can be used either in the initial testing of software or in regression testing. System testing refers to testing conducted on a completed, integrated system to check whether it complies with requirements. Regression testing refers to retesting software following modifications(bug fixes). Prioritization techniques applied to the above two differ. In the case of the latter, there is the added advantage of using information obtained from previous runs of test cases in performing prioritization. No such information is available in the former as the build is new.

2. General test case prioritization and Version-specific test case prioritization [RUCH01]

   Test cases are prioritized with the intent of finding an ordering that will be useful over a succession of subsequent modified versions of the program. Version-specific test case prioritization, looks at prioritizing test cases to find an ordering that will be useful for a specific version. Version specific prioritization uses information about modified versions of software whereas general test case prioritization does not do so.

3. Fine granularity and coarse granularity techniques [EMR02]

   Fine granularity techniques refer to prioritization applied at the level of source code statements whereas coarse granularity techniques operate at the function level.

4. Execution, History, Requirements and Metrics based [TAS06]

   Execution based techniques are those that prioritize test cases based on the level of coverage reached at various levels(statement, block, function) during previous runs. Ranking of test cases is based on the number of statements executed/covered. The test cases covering more lines of code are placed early in the test order. In the case of branch and function coverage, tests are prioritized based on the program branches or program functions covered, respectively. History based techniques use historical data like assigning higher priority to test cases not executed recently or which revealed faults recently. Prioritization techniques based on requirements look at four factors, namely, customer-assigned priority(CP), requirements complexity(RC), requirements volatility(RV), and fault proneness(FP). Metrics based techniques base prioritization on some metric like fault proneness index.

2.6.1.1 Classifying prioritization techniques

Through a survey of existing work, test case prioritization techniques are broadly classified in this work as:

- *Requirements based Test Case Prioritization.* [EMR01, SWO05]

  Test cases are prioritized based on the properties of requirements, namely, requirement volatility, customer need, complexity and fault proneness. Usually, these measures are provided by the customer based on project and customer needs.

- *Coverage based Test Case Prioritization.* [GRCJ99, RUCH01, EMR02]

  Coverage-based techniques rank test cases based on the level of coverage achieved i.e. test cases are ranked based on the number of statements executed/covered. The technique is based on the belief that higher the number of statements executed, earlier the defects get detected. Branch and function coverage techniques prioritize test cases based on branches or program functions covered. Thus, coverage can be computed at various levels like statement, block, branch, condition/decision and function. Additional coverage, a variant of the techniques takes into account additional coverage provided by each test case with reference to the coverage achieved at various levels.

- *History based Test Case Prioritization.* [KP02]

  Test cases are prioritized based on data captured from previous runs of the test case. Factors used for prioritization include test cases which have not been executed recently, or which revealed faults recently or which cover functions not yet covered.

- *Risk based.* [SMP08]

  Risk based test case prioritization looks at the probability of a fault and

the damage that the fault can cause when failure occurs as the means for prioritization.

- *Model based Test Case Prioritization.* [Moi00, FW07, KS07, SMP08, KK09a] Model based prioritization techniques look at using models of the system as the basis for prioritization. Factors considered include importance of use cases(requirements) on the basis of stakeholder ratings, business goals, influence of use case on business goal provided by each stakeholder and ranking of use case based on usage metrics. Other techniques include using scenarios obtained from use case diagram and risk related to activity diagrams as basis for prioritization.

- *Metaheuristic and evolutionary techniques.* [TAS06, RM09] Metaheuristic and evolutionary techniques involve the use of machine learning algorithms and genetic algorithms for prioritizing test cases.

2.6.1.2 Classifying Coverage based Prioritization Techniques

Literature gives various techniques used for prioritizing test cases based on coverage achieved[GRCJ99, RUCH01, EMR02, SMP08].

Comparator Techniques. Comparator techniques include random and optimal [GRCJ99]. In case of random, the test cases in the test suite are ordered randomly. Optimal ordering orders test cases by the highest number of revealed faults among those yet to be discovered. This ordering can only be approximated.

Statement Level Technique. Statement level techniques order test cases according to the number of covered statements[GRCJ99, RUCH01, EMR02]. Variations of statement level techniques include additional statement coverage, branch coverage, additional branch coverage, Total Fault Exposing Potential(FEP) and Additional FEP.

Function Level Technique. Function level techniques sort test cases in order of the total number of functions they execute[GRCJ99, RUCH01, EMR02]. Variations include additional function coverage, total FEP(function level), additional FEP, total fault index(FI) and additional FI.

Risk based Technique. Risk based techniques sort test cases in descending order of associated risks[SMP08]. Additional risk score is a variation of the previous technique taking into consideration those test cases where risk has not been covered.

2.6.2 Classifying Metrics

Two metrics have been proposed in literature to calculate the efficiency of techniques used for prioritization, namely Average Percentage of Faults Detected(APFD) and Cost-Cognizant Average Percentage of Faults Detected ($APFD_c$)[RUCH01, MRRE06]. In this work, the APFD metric is used to calculate the rate of fault detection.

Average Percentage of Faults Detected(APFD)

Rothermel et al[RUCH01] introduced the metric, APFD, to measure the effect of prioritization. They use a weighted average of the percentage of faults detected over the life of the test suite. Values for APFD range from 0 to 100; higher values for APFD indicates faster fault detection rates. APFD can be computed using the following equation:

$$APFD = 1 - \frac{\sum_{j=1}^{k} pos(e_j)}{nk} + \frac{1}{2n} \tag{2.1}$$

where n is the number of test cases, k is the number of revealed faults and $pos(e_j)$ is the position (between 1 and n) of the first test case revealing fault e_j, in the

prioritized sequence.

Table 2.4: Test suite with faults

test	fault				
t_1	×	×	×	×	
t_2	×			×	×
t_3	×		×		
t_4					×
t_5			×		×

To illustrate APFD, consider the example in Table 2.4 representing a test suite that consists of 5 test cases t_1 - t_5 and 12 faults. Thus, test suite T = $t_1, ...t_5$ and we know that the tests detect faults of type f_1,...,f_5 in program P according to Figure 2.1.

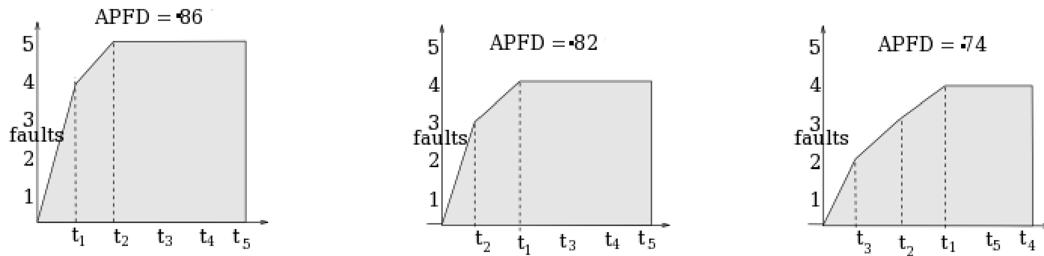


Figure 2.1: APFD is higher for test case orders that reveal most faults early.

Taking three different orderings of the tests cases, the APFD rates are as shown in Figure 2.5. Thus, for test suite T_1 we have APFD = .86, for T_2 .82 and .74 for T_3. T_1 has greater weighted average of the percentage of faults detected than T_2 and T_3. This is due to the fact that the test ordering in T_1 which are able to detect faults in P are executed first when compared to T_2 and T_3. Thus, APFD is higher for test case orders that reveal most faults early as shown in Figure 2.1. In this work, APFD is used to measure the fault detection index.

Cost-Cognizant Average Percentage of Faults Detected ($APFD_c$)

In the case of APFD, it is assumed that test case costs and fault severities are uniform. However, there is a tradeoff between costs of testing and the cost of un-

Test Suite	Order	APFD
T_1	$t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow t_4 \rightarrow t_5$	86%
T_2	$t_2 \rightarrow t_1 \rightarrow t_3 \rightarrow t_4 \rightarrow t_5$	82%
T_3	$t_3 \rightarrow t_2 \rightarrow t_1 \rightarrow t_5 \rightarrow t_4$	74%

Table 2.5: APFD is higher for test case orders that reveal most faults early.

detected faults. This tradeoff is important when considering the order of executing test cases. In [EMR00], the authors incorporate the rate of fault severity detected per unit test cost. They adapt the APFD metric to include the cost factor, called cost-cognizant APFD, $APFD_c$.

2.6.3 A Taxonomy of Features for comparing work on Test Scenario Prioritization

In this section, various features used for comparing existing work on test scenario prioritization is presented. The features show the basis for prioritization. A comparative analysis of surveyed work is then presented.

2.6.3.1 Features for Test Scenario Prioritization

The following gives the set of features based on which prioritization of test scenarios is done to order test cases in a test suite. The comparison of existing work(refer Table 2.6) is based on the features discussed below.

1. **Technique:** What is the technique used for test case prioritization ? The choice of technique is based on the requirement for testing related to the project in hand. Prioritization of test scenarios can be based either on
i) requirements[EMR01, SWO05] ii) coverage[GRCJ99, RUCH01, EMR02] iii) history[KP02] iv) risk[SMP08] v) model[Moi00, FW07, KK09a] or vi) metaheuristic and evolutionary techniques[TAS06, RM09].

2. **Customer inputs required :** Does the technique require customer inputs to be used as the basis of prioritization ? Customer inputs provide added advantage in terms of customer involvement, gaining knowledge of customer needs and serves as a feedback loop providing continued inputs for further development of software.

3. **Model used :** Is a model used to aid the process of prioritization ? In case a model is used, whether UML or otherwise. The objective is to understand the use of UML models in prioritization.

4. **Type :** Is prioritization used for system or regression testing ?

5. **Objective :** What is the objective of prioritization ? The objective of test case prioritization is either to ensure a high rate of a) Fault Detection b) Coverage.

6. **Automated Support :** Was a prototype tool developed to faciliatate the use and learning of the approach ?

7. **Criterion :** Has single or multiple criterion used as basis for prioritization ?

2.6.4 Discussion

Based on the survey(refer Table 2.6) the following conclusions can be made:

Technique. Prioritization techniques have been categorized into requirements based, coverage based, history based, risk based, model based and meta-heuristic and evolutionary techniques. The advantage of requirements based techniques [KR97, BBM02, TAS06, FW07, QNXZ07, YHTS09, BEG09] is that it

Table 2.6: Comparison of prioritization techniques

Ref #	Techniques used						Cust I/p	Mod	If model		System Testing	Regression Testing	Objective		Aut. Tool	Index	
	Req	Cov	Hist	Risk	Model	MH & Evol			UML	Other			FD	Cov		Single	Multiple
[KR97]	✓	×	×	×	×	×	✓	×	×	×	✓	×	×	✓	×	✓	×
[GRCJ99]	×	✓	×	×	×	×	×	×	×	×	×	×	✓	×	×	✓	×
[Moi00]	×	×	×	×	✓	×	×	✓	✓	×	✓	×	×	✓	×	✓	×
[RUCH01]	×	✓	×	×	×	×	×	×	×	×	×	×	✓	×	×	✓	×
[EMR01]	×	✓	×	×	×	×	×	×	×	×	×	×	✓	×	×	×	✓
[EMR02]	×	✓	×	×	×	×	×	×	×	×	×	✓	✓	×	×	✓	×
[BBM02]	✓	×	×	×	✓	×	✓	✓	✓	×	✓	×	×	✓	✓	✓	×
[ST02]	×	✓	×	×	×	×	×	×	×	×	×	✓	✓	✓	✓	✓	×
[KP02]	×	×	✓	×	×	×	×	×	×	×	×	✓	✓	×	×	✓	×
[YNC03]	✓	×	×	×	×	×	×	×	×	✓	×	×	×	×	✓	✓	×
[JH03b]	×	✓	×	×	×	×	×	×	×	×	×	✓	×	✓	×	✓	×
[ASK04]	×	✓	×	×	×	×	×	×	×	×	×	✓	×	✓	×	✓	×
[DRK04]	×	✓	×	×	×	×	×	×	×	×	×	×	✓	×	×	✓	×
[ERKM04]	×	✓	×	×	×	×	×	×	×	×	×	✓	✓	×	×	✓	×
[SWO05]	✓	×	×	×	×	×	✓	×	×	×	✓	✓	✓	×	✓	×	✓
[WSKR06]	×	×	×	×	×	✓	×	×	×	×	×	✓	✓	×	×	✓	×
[GEB06]	✓	×	×	×	×	✓	×	×	×	×	✓	×	×	✓	×	✓	×
[TAS06]	✓	✓	×	×	×	✓	✓	×	×	×	✓	×	✓	×	×	✓	✓

✓ - means use of technique/method and × indicates absence of technique/method; Blank indicates that information could not be gleaned

Comparison of prioritization techniques(Contd/-)

Ref #	Techniques used						Cust I/p	Mod	If model		System Testing	Regression Testing	Objective		Aut. Tool	Index	
	Req	Cov	Hist	Risk	Model	MH & Evol			UML	Other			FD	Cov		Single	Multiple
[JG06]	×	✓	×	×	×	×	×	×	×	×	×	✓	✓	×	×	✓	×
[LHH07]	×	×	×	×	×	✓	×	×	×	×	×	✓	×	✓	×	✓	×
[FW07]	✓	×	×	×	✓	×	×	✓	×	×	×	×	✓	×	✓	✓	×
[ZNXQ07]	✓	×	×	×	×	×	×	×	×	×	×	×	✓	×	×	✓	×
[MT07]	×	×	×	×	×	✓	×	×	×	×	×	✓	✓	×	×	✓	×
[QNXZ07]	✓	×	×	×	×	×	×	×	×	×	×	✓	✓	×	×	✓	×
[PRB08]	×	×	✓	×	×	×	×	×	×	×	×	✓	✓	×	×	✓	×
[MZ08]	×	✓	×	×	×	×	×	×	×	×	×	✓	✓	×	✓	✓	×
[SMP08]	×	×	×	✓	✓	×	✓	✓	✓	×	✓	×	✓	×	×	✓	×
[KK09a]	×	×	×	×	✓	×	×	×	×	×	×	✓	✓	×	×	✓	×
[KM09]	✓	×	×	×	×	✓	×	×	×	×	✓	×	✓	×	✓	✓	×
[YHTS09]	✓	✓	×	×	×	×	✓	×	×	×	✓	×	✓	×	×	✓	✓
[BEG09]	✓	✓	×	×	✓	×	×	✓	×	×	✓	×	×	✓	×	✓	×

✓ - means use of technique/method and × indicates absence of technique/method; Blank indicates that information could not be gleaned

focuses on the properties of requirements as basis for prioritization. The customer plays an important role in assigning priority of requirements. Technical complexity of developing requirements is not considered during prioritization or it is assumed that the technical complexity is taken into consideration when assigning values during prioritization. Risk based techniques[CP03a, SMP08] again look at prioritizing test cases from the perspective of requirements by considering the risk(probability) that a requirement(use case) will have a defect. Again, this method of prioritization is dependent on manager's knowledge of the strength of the software domain as well as the software development team. Coverage based techniques[ST02, JH03b, ASK04, DRK04, ERKM04, JG06, MZ08, YHTS09, BEG09] work more efficiently when the size of the software is small or medium. In case of large size of software it is therefore better to introduce constraints of time and cost to contain the size of the test suites. History based techniques[KP02, PRB08] look at using data from previous run of test suites to aid in prioritization. The advantage of history based techniques is that it helps understand areas of defects in the software. But, changes in software made across iterations may alter the defect location. Model based testing techniques[Moi00, BBM02, FW07, SMP08, KK09a, BEG09] have focussed primarily on three UML models, namely, the use case, sequence and activity diagram. Again, all the models have been looked at from the perspective of requirements. The technical complexity and the interdependencies is not considered for prioritization.

Customer Inputs. In case of requirements based prioritization techniques, various parameters about requirements are gathered from both the customer as well as the technical team[KR97, BBM02, CP03a, SWO05, TAS06, SMP08, YHTS09]. In case of risk based prioritization techniques, risk value is assigned by the customer/developer to each activity in the activity diagram[YNC03,

SMP08]. The main objective of building software is to satisfy customer requirements. Customer inputs play an important role especially where prioritization values are inferred based on requirements. Hence, there is need to obtain customer inputs during prioritization.

Model used. Use of models aid in communication between various stakeholders of the system. Use case, sequence and activity diagrams have been used as models besides ESG(Event Sequence Graph) to capture requirements which in turn have been used to aid the process of prioritization[Moi00, BBM02, FW07, SMP08]. However, in case of UML, models have been used to capture requirement, and priority has been assigned to use cases and scenarios[BBM02] by customers or by technical team members. Risk is a factor used for prioritization by [Moi00] assigned to activities in an activity diagram.

Type. System testing involves testing all features related to a particular version to check if they work together in a comprehesive manner. Regression testing involves testing a set of features that could be affected due to modification of a feature or function. The modification could have been caused by resolving a bug or by an enhancement. [EMR02, ST02, KP02, CP03a, JH03b, ASK04, ERKM04, SWO05, WSKR06, LHH07, MT07, PRB08, MZ08, KK09a] present techniques for regression testing.

Prioritization Index. Most work use a single prioritization index[MT07, QNXZ07, ZNXQ07, PRB08, MZ08, KK09a] for prioritization while [EGR01, SWO05, TAS06, YHTS09] use a multiple priorization index.

Automated Tool. An automated tool aids practitioners in using the technique to prioritize test cases. With reference to UML models, which is the focus of this work, Basanieri et al[BBM02] have developed an automated tool that aids in calculating the priority of scenarios in their Cow-suite tool. Other tools for prioritization include [ST02, YNC03, SWO05, FW07, MZ08, KM09].

The survey shows a wide set of approaches to prioritize test cases for both system and regression testing. Again, different criterion have been used to prioritize scenarios. To compare the viability of method with others, various techniques are used like comparator techniques, function level, statement level and risk based techniques. The focus of this work is on using UML models as the basis for prioritization. Use case and activity diagrams have been found best suited to communicate requirements between various stakeholders of the system, namely, customers, designers, developers and test engineers. Due to this advantage and the ease of understanding primitives by all the stakeholders, use case and activity diagrams are used in this work for prioritization. Again, based on survey, UML models have been used to obtain customer inputs on requirements. For example, activity diagrams are used to obtain risk value connected to each activity. This method becomes tedious if the size of the software system increases due to the increase in activity diagrams. It would become cumbersome and error prone if prioritization or risk values have to be assigned to each activity or scenario as the case maybe. Hence, there is need for an automated technique that can be used along with customer inputs for prioritization to make it effective. Also, such a technique becomes effective only if it can be used with modelling tools available. Such a tool helps developers and test engineers understand complexity and priority of requirements and exploit information obtained to better the results of testing.

2.7 Test Case Selection

Full testing involves running all the tests in the test suite. This is exhaustive and will consume an inordinate amount of time and money. Hence, an ordering of test cases aids in early detection of faults. However, ordering and running a large test suite is still infeasible, as it would not be possible to run all tests at every

iteration.

Test case selection provides several benefits in the automated testing process. Exhaustive testing being impossible, there is need to determine a subset of test cases that ensure test objectives, namely, maximum coverage, and early fault detection. For this, there is need to select an effective subset of the original test suite. A test case selection method is used to identify test cases for testing according to a selection criterion. The aim is to cover some important aspects of the SUT, with objective that coverage will lead to early fault detection. Different test selection techniques focus on different coverage criteria. The techniques attempt to maintain quality of throughput by reducing the size of the test suite and at the same time increasing efficiency. Thus, choice of test case selection method depends on the requirements to be tested, the associated coverage criteria of the test case selection method, and the types of faults that the test case selection method targets [Lev06].

2.7.1 Classification

Techniques used for test case selection can be classified into similarity based (distance), heuristic based, risk based and model based.

Similarity based techniques

Cartaxo et al and Chen et al [CANM08, CNM07, CLM04] use similarity measures for test case selection. Test cases are selected based on the distance value computed between two test cases. Techniques used include pairwise comparison in [CNM07] and Euclidean distance in [CLM04]. In [CLOM06], the authors extend the idea of distance to testing object oriented programs by defining a distance for objects, the 'object distance'. The object distance computes distances between arbitrary objects. A model for representing the differences between two objects

is presented, and Levenshtein distance is used to calculate distance between class names, object names and its contents.

Heuristic based

[BLFR02] in their work consider heuristic driven techniques for test selection. They use four factors, namely, risk, coverage, cost and efficiency, which helps in classification and selection of test cases according to different criteria. Rothermel et al [RH96] in their work present a framework for analyzing regression test selection techniques. The framework consists of four categories: inclusiveness, precision, efficiency, and generality. They analyze different techniques on the four factors.

Risk based

Risk is an event that threatens successful competion of a project. Scenarios with high risk need to be given higher priority and tested completely especially in case of regression testing. Regression testing is performed in order to guarantee that newly introduced changes in a software do not affect unchanged parts of software. A simple way to regression testing is to retest-all but is expensive. Besides, the time limit imposed on testing does not allow exhaustive testing. Most regression testing techniques are code based which is advantageous in case of unit testing but faces scalability issues as size of software increases. Risk based test selection is the focus of work done by Chen et al [CP03b]. Risk information pertaining to test cases is provided by test engineers using experience gained. Risk metrics are used to measure quality of the test suite. Risk is based on the cost of each test case(valued by both customer and test engineer) as well as severity. Risk exposure(RE), a product of cost and severity is taken as the basis for test selection. Scenarios that cover most critical cases are considered first.

Model based

In [RH97], the authors construct control flow graphs(CFG) for a procedure or program and its modified version. They use the CFGs to select tests that execute changed code from the original test suite. Activity diagrams are used for regression test selection by [CPS02]. Regression test selection techniques involves selecting a subset of test cases to determine if the modified program has the same behaviour as a previous, acceptable version of the program running on T, a set of test cases.

The Cow Suite tool by Basanieri et al [BBM02] use UML use case and sequence diagrams to record requirements. Each use case diagram is elaborated using a sequence diagram, to scenarios forming a tree structure. Weights are assigned based on functional importance such that the sum of weights at any level equals to one. The test generation algorithm used by Cow Suite generates all possible test cases. Selection is done based on the weight of the scenarios obtained by product of weights of all nodes in the path leading to particular scenario.

Random

Random selection of test cases is generally used in case of black box techniques. Here, test cases are chosen in random based on a uniform distribution. The advantage of random testing includes availability of algorithms to generate test cases. The rate of failure-causing inputs i.e. failure rate, is used to measure effectiveness. Adaptive random testing, a new type of random testing is introduced in [CLM04] based on patterns of failure-causing inputs.

2.7.2 A Taxonomy of Features for comparing work on Test Scenario Selection

In this section, various features used for comparing existing work on test scenario selection is presented. The features show the basis for selection. A comparative analysis of surveyed work is then presented.

2.7.2.1 Features for Test Scenario Selection

The following gives the set of features based on which selection of test scenarios is done to form a test suite. The comparison of existing work is based on the features discussed below.

1. **Input:** The input to test scenario selection has a bearing on the selection process. Specification of the system using a modelling language like UML or a formal method can be used as the basis for selection of scenarios. Another approach is to use the developed code by deducing a Control Flow Graph(CFG) to obtain paths and use as input for test case selection.

   Thus, the source for selection of scenarios can be based on:
   (a) Specification (b) Source code

2. **Strategy :** What is the strategy used for selection of scenarios? The strategy used for test selection is based on the need of an organization with reference to the corresponding software under test. Strategies include use of a heuristic like risk, coverage and cost or can be based on similarity index, customer input and priority of scenarios.

3. **Specification language :** What is the choice of specification language used ? The choice of specification language can be a modelling language, either UML or other, like Labelled Transition Systems(LTS).

4. **Adequacy Criterion :** An adequacy criterion is a set of test obligations that the test suite has to satisfy. For example, activity coverage adequacy criterion is achieved by test suite S if atleast one test case TS executes the activity. Two important criteria used to judge effectiveness of test selection techniques are:

   i) Coverage ii) Fault Detection

   Adequacy criterion provides guidance in devising a thorough test suite. The question asked here is: What is the adequacy criterion used for selection of scenarios ?

5. **Automated Support:** Is a prototype tool developed to facilitate the use and learning of the approach ?

6. **Case Integration :** Has integration of approach been done with CASE tools?

7. **Domain Specific.** Is the technique generic or domain specific ?

8. **Criterion:** The objective of test selection can be a single criterion or multiple criterion. Former involves using a single criterion like risk for evaluation. Multiple criterion involves use of two or more criterion to check for effectiveness.

2.7.3 Discussion

The survey(refer Table 2.7) gives an overall view of the capabilities of each technique and is used to make the following conclusions:

Table 2.7: Comparison of approaches to selecting test cases

Ref#	Type		Strategy										
	Spec based	Source based	Model based	History based	Heuristic				Similarity based	Random	Cust input	Priority	Other
					Risk	Coverage	Cost	Efficiency					
[RH96]	×	✓	×	×	×	×	×	×	×	×	×	×	×
[RH97]	×	✓	×	×	×	×	×	×	×	×	×	×	×
[HHU00]	✓	×	✓	×	×	×	×	×	×	×	×	×	×
[HJL+01]	×	✓	×	×	✓	✓	×	×	×	×	×	×	×
[CPS02]	✓	×	✓	×	✓	×	×	×	×	×	✓	×	×
[BRFIGC+02]	✓	×	×	×	✓	✓	✓	✓	×	×	✓	×	×
[BBM02]	✓	×	✓	×	✓	×	×	×	×	×	×	✓	×
[CP03b]	✓	×	×	×	✓	×	×	×	×	×	✓	×	×
[CLM04]	×	×	×	×	×	×	×	×	✓	×	×	×	×
[WE05]	×	✓	×	×	×	×	✓	×	×	×	×	×	✓
[CLOM06]	×	×	✓	×	×	×	×	×	✓	×	×	×	×
[Lev06]	×	×	×	×	×	×	×	×	×	×	×	×	×
[HBB06]	×	×	✓	×	×	×	×	×	×	×	×	×	×
[Hes06]	✓	×	✓	×	×	✓	×	×	×	×	×	×	×
[CNM07]	✓	×	✓	×	×	×	×	×	✓	×	×	×	×
[YH07]	×	×	×	✓	×	×	✓	×	×	×	×	×	×
[TZP+07]	✓	×	✓	×	×	×	×	×	×	✓	×	×	×
[CANM08]	✓	×	✓	×	×	×	×	×	✓	×	×	×	×
[BMSS09]	A	✓	✓	×	×	✓	✓	×	×	×	×	✓	×

A - Specification annotated with details; C - UML used as base model and then converted to another; ? - Unknown

Comparison of approaches to selecting test cases (Contd/-)

Ref#	Model used		Adequacy Criterion		Automated Support	Case Integration	Domain Specific	Optimization	
	UML	Other	Fault Detection	Code Coverage				Single Objective	Multiple Objective
[RH96]	×	✓	×	×	×	×	×	×	×
[RH97]	×	✓	✓	×	×	×	×	×	×
[HHU00]	✓	*C*	✓	×	✓	×	×	✓	×
[HJL+01]	×	✓	✓	×	✓	×	×	✓	×
[CPS02]	✓	×	×	✓	?	×	×	✓	×
[BRFIGC+02]	×	✓	×	✓	?	×	×	✓	×
[BBM02]	✓	×	×	✓	✓	×	×	✓	×
[CP03b]	✓	×	×	✓	?	×	×	✓	×
[CLM04]	×	×	✓	×	×	×	×	✓	×
[WE05]	×	✓	×	✓	✓	×	✓	✓	×
[CLOM06]	✓	×	×	✓	✓	×	×	✓	×
[Lev06]	×	×	✓	✓	×	×	×	✓	×
[HBB06]	✓	×	×	✓	×	×	×	✓	×
[Hes06]	×	✓	×	✓	✓	×	✓	✓	×
[CNM07]	×	✓	×	✓	×	×	×	✓	×
[YH07]	×	×	✓	✓	×	×	×	×	✓
[TZP+07]	×	×	×	✓	×	×	✓	✓	×
[CANM08]	×	✓	×	✓	×	×	×	✓	×
[BMSS09]	✓	*C*	×	✓	✓	×	×	✓	×

A - Specification annotated with details; C - UML used as base model and then converted to another; ? - Unknown

Input. Test case selection technique are based either on specification or source code. In the former, specification captured using graphical methods are used for selection [CANM08, CNM07, CLM04]. In case of the later, Control Flow Graphs(CFGs) are deduced from the code and used for selection especially in the case of regression test selection techniques where the difference in CFG is used as the basis [RH96, RH97, HJL+01, BMSS09]. While the former is a test oriented approach wherein test cases are developed and selected parallelly with development and provided as input for testing, the latter involves deducing paths from developed software and then matching with the design to determine scenarios that need to be selected for further testing.

Strategy. Various techniques are used as the basis for test case selection namely, model[HHU00, CPS02, BBM02, CLOM06, HBB06, Hes06, CNM07, TZP+07, CANM08, BMSS09], similarity[CLM04, CLOM06, CNM07, CANM08], random[TZP+07] and other heuristics like risk[HJL+01, CPS02, BRFIGC+02] and cost[BRFIGC+02, WE05, YH07, BMSS09]. The choice of selection is dependent on the type of software and the requirements of testing.

Models used. Model based selection is an efficient way of capturing specification of a system. Models both UML[BBM02] and others(Labelled Transition Systems[CNM07]) have been used for test selection. Scenarios are represented using models. Distance measures have been used with LTS as models, to calculate the similarity of scenarios and are used as the basis for selection. Random selection is used to pick a representative set from scenarios that are similar.

Adequacy Criterion. Two criterion for adequacy of the test selection approach considered are fault detection and code coverage. The objective of fault detection[RH97, HJL+01, CLM04, Lev06] is to see if the selection gives a subset of scenarios that detect faults early or reveal more or as much faults as the basic set. [CPS02, BRFIGC+02, BBM02, CP03b, CANM08, BMSS09] use code coverage as

adequacy criterion.

Automated Support and CASE Integration. Automated support and integration with CASE tools aid test engineers to use the approach with existing modelling tools. Also, in case UML models are used as the basis for test selection, there is scope for integrating the techniques with existing CASE tools. Test selection is done using the Cow suite by Basanieri et al[BBM02] by defining the stage at which testing is to be done and selecting nodes(here requirements) that need to be tested. For each sequence diagram elaborating a use case(requirement), test cases are selected by considering those scenarios of highest weight obtained from the weighted tree. Here, the selection is done at the level of requirements. However, it is possible that there are scenarios common to more than one requirement, or that they are similar which might get selected.

Domain Specific. The technique can either be domain specific or generic in nature. A domain specific technique as in [WE05, Hes06] are suited to applications related to particular domains. A generic technique has the advantage of being applicable across domains and hence widely usable.

Criterion. Most work use a single criterion as the basis for selection. [YH07] use a multiple criterion, namely, fault detection and coverage as the basis for selection.

Thus, different techniques have been used for test selection, both based on specification and code. Also, Specification based Testing is a widely used method for testing wherein the same specification is used for purposes of testing. Models like LTS and UML have been used for specification which is used as the basis for test selection also. Distance measures have been used for determining similar scenarios. To pick one among similar scenarios, random selection has been adopted. Random selection though effective in many cases may not always produce best results. Hence, there is need to study if techniques besides random,

based on specification can be used for picking a representative set from a similar set of scenarios. Automated support and CASE integration is important in making selection techniques effective especially when using modeling tools.

2.8 Use of Ontology in Software Engineering

A clear understanding of the software, entities, attributes and relation between its entities is required for testing. Use case models represent functional requirements of a system in terms of actors, their goals(represented as use cases) and the dependencies between use cases. Use case diagrams do not provide details of how the use case is to be implemented. Activity diagrams elaborate a use case and together, they provide information required for testing. The relation between use cases and activities can be made more explicit using ontologies. Ontologies are used in cases where there is need for representation of entities having significant number of attributes and relations between them. Complexity in software arises due to the need to map the large number of entities and the relations between them. The complexity of software induces defects. Representation of the entities, attributes and relations between them using an ontology can uncover the relations and other important attributes, thereby helping in tracing defects. The deeper representation of the system also helps in the system analysis required for generating test cases and in test management.

2.8.1 Ontology - Definition

An ontology is 'a formal explicit specification of a shared conceptualization' [Gru08]. An ontology defines a common set of concepts and relations used to describe and represent domain knowledge [BHHS06]. Na et al [HSNL06] give the following definition:

An ontology is a tuple O:=(C,P,I,R) where:

1. C = c_1, c_2, c_3, ... c_n is a set of concepts(classes), where each concept c_i refers to a set of real world objects(instances).

2. P = p_1, p_2, p_3, ...p_n is a set of properties, which can be divided into attribute and relations.

3. I is a set of instances. Instances are individual members of concepts.

4. R is a set of restrictions. Each property has a set of restrictions on its values, such as cardinality and range.

An ontology includes machine interpretable definitions of basic concepts in the domain as well as the relations among the concepts. The objective of using an ontology is to reduce ambiguity in terms of concepts and terminologies used among members who need to share documents and information of various kind [aPV04].

Thus, an ontology consists of concepts(otherwise, called classes or entities), properties of each concept describing various features and attributes of the concept(otherwise called roles or properties) and restrictions on properties [HSNL06]. An ontology together with a set of individual instances of classes constitute a knowledge base. Creating ontologies is a difficult and time consuming process [aPV04]. The task of building a domain ontology includes defining basic concepts and structures (objects, properties, relations and axioms) that are applicable in the target domain [HSNL06].

For example, consider the 'Pizza' ontology. The main concepts include 'Pizza', 'Pizza base' and 'Pizza topping'. Properties include 'has_topping', 'has_base' and 'has_spiciness'. Restriction on 'has_spiciness' property includes 'hot_value', 'medium_value' and 'mild_value'. Similarly, values for 'has_base' include 'thick_crust' and 'thin_crust'. Individuals of the Pizza ontology include 'Margherita' and 'Marinara' which are types of pizzas. Ontologies thus defined can be classified according to the knowledge they capture which is discussed in the next subsection.

2.8.2 Classification of Ontologies

Ontologies can be defined based on different features. Figure 2.2 shows different kinds of ontologies classified based on different criteria like generality, formality, level of granularity(level of detail) and the type of data they capture about the world.

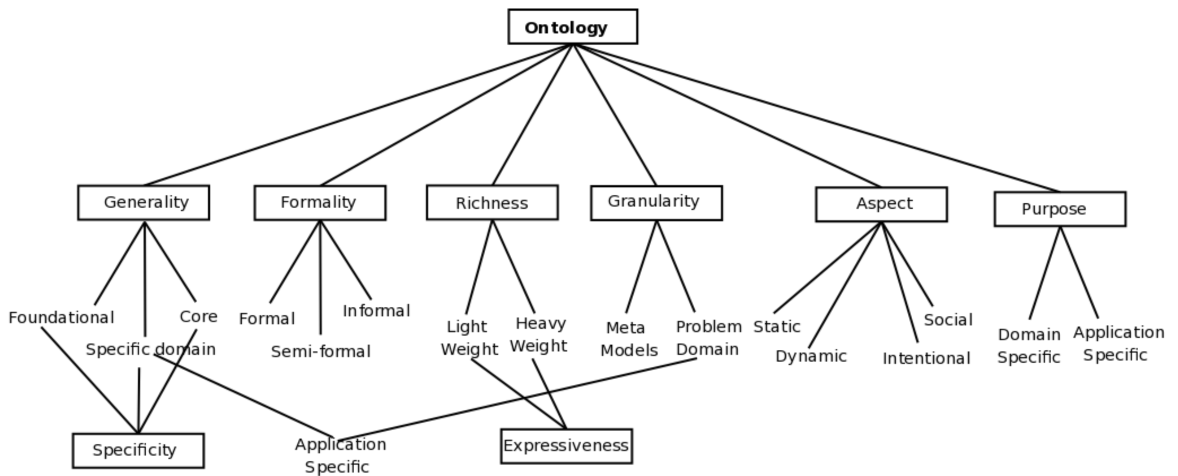


Figure 2.2: Classification of Ontologies

Navigli et al [aPV04] classify ontologies into three levels, foundational, core ontology and specific domain ontology based on generality. The level of formality(informal, semiformal and formal) is used as the basis by [Gru03, BCO09]. Saeki et al [SK06] classify ontologies into ontologies of meta models and ontology of problem domain based on the level of granularity. Jurisica et al [JMY04] classify ontologies as static, dynamic, intentional and social ontologies based on aspects they capture of the world. The level of richness of internal structure is the basis of classification used by Francisco et al [RH06]. They categorize ontologies into lightweight and heavyweight ontologies. Oberle [Obe06] classifies different kinds of ontologies according to purpose, specificity and expressiveness.

Common classification of ontologies group them by the level of abstraction and expressiveness. When trying to understand how ontologies can be applied

to aid the process of testing, there is need to classify ontologies according to the type of knowledge, level and granularity they need to capture so as to maximize benefits for testing. In this chapter, a simple classification of ontologies is proposed based on the what type of knowledge needs to be represented at what level and granularity.

Hence, ontologies are classified by:

- Perspective - What kind of knowledge is being captured i.e. it includes foundational, core and domain specific ontologies defined by Navigli et al and Saeki et al as well as the purpose and specificity classes of ontology classification defined by Oberle.

- Structure - At what level of granularity is the knowledge represented i.e. it includes informal and formal ontologies defined by Bezerra et al as well as the expressiveness class defined by Oberle.

- Attributes captured - What type of knowledge is being captured i.e. ontologies that capture some aspect of a system as defined by Jurisica et al.

Thus, each type of ontology captures knowledge in an area and is meant for use by different users for different purposes. The next subsection discusses the use of ontologies.

2.8.3 Use of Ontologies

An ontology is thus a formulation of entities relevant to a domain as well as the relationship between the entities and have been used primarily to facilitate efficient exchange of information with people as well as with external agents [CP99, MMB09]. Thus, one of the primary purposes of an ontology is to allow different users to understand the content and interact intelligently. They provide a means for describing and reasoning about data, objects and relations in a domain.

Reasons for building an ontology include [Gru93, NM01]:

- sharing common understanding of the structure of information among people or software agents
- enable reuse of domain knowledge
- make domain assumptions explicit
- separate domain knowledge from operational knowledge and analyze domain knowledge

Uschold et al [UJ99] broadly group the use of ontologies into three areas:

- *Communication.* Ontologies are used to assist in communication between people. An ontology of an application domain can be used by various stakeholders of the system to understand the entities that exist and the relation among them.
- *Inter-Operability.* An ontology can be used as an interchange format for translation between different modeling methods, paradigms, languages and software tools. Thus, ontologies aid in inter-operability among systems.
- *System Engineering.* Ontologies are used in system engineering in various phases as well as for knowledge acquisition. In case of specification, ontologies assist in identifying requirement and defining system specification. Formal representation of the system using ontologies helps check consistency of the system making it reliable. Also, the specification thus captured is reusable and maintainable. An ontology also helps in searching for information.

Ontologies have been applied extensively and are used in varied areas like Computer Networks, Semantic Web, Artificial Intelligence, Law, Health and Medicine,

Library Sciences, Systems engineering and Software Engineering [KG95, Soe99, eaCH+02, JMY04, Val05, SCG, Sic06, DCW08, Gru08]. The advantage of using ontologies is that once knowledge is formalized, it is possible to reuse, perform inference, process using computers and is communicable between people and software [BCO09]. The scope of this survey is to study the use of ontologies in software engineering, specially with reference to UML and testing. Subsection 2.8.3.1 discussed the use of ontologies in that area of software engineering. Subsection 2.8.3.2 discusses work done on UML and ontologies while subsection 2.8.3.3 discusses how ontologies have been used in testing.

2.8.3.1 Software Engineering and Ontologies

According to IEEE Computer Society's Software Engineering Body of Knowledge, Software engineering is the 'application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software'[Ala04]. Ontologies have been used in various activities related to the software development process. To bridge the gap between knowledge of SE practitioners and Artificial Intelligence techniques, proposals have been made to use ontologies in Software Engineering [Soe99]. A software engineering ontology defines common sharable software engineering knowledge including particular project information [WCS07]. A software engineering ontology abstracts concepts and instances from the software engineering domain.

Today, software teams are distributed across geographical locations. This geographical distribution makes it difficult to develop a consistent understanding of terminology or understanding of project agreement due to the absence of face to face communication [WCD05]. Also, the ability, knowledge level and preferences of members of such teams vary widely[GdF04]. Hence, there is need to capture concepts relating to a project in such a way that it is unambiguous. Also, knowl-

edge of a system must be shared in such a way that it means the same to different people(architects, developers and test engineers). Thus, a software engineering ontology facilitates collaboration of remote teams in multi-site distributed software development [WCS07].

Wongthongtham et al [WC05, WCD05, WCS07] use ontologies to aid communication between distributed teams. Two types of software engineering ontologies namely, generic and specific have been defined. Generic software engineering ontology represents all software engineering concepts, their relationships, and their constraints. It is an abstract level of representation without ontology instances. On the contrary, a specific software engineering ontology represents specific instances for the corresponding software engineering concepts. These instances contain actual project data/information/agreement. The specific ontology contains the set of actual data or instances of the concepts and assertions that the instances are related to according to the specific relations in the concepts.

In the field of software engineering, specific ontologies have been identified. Dillon et al [DCW08]propose a software engineering ontology consisting of sub-ontologies namely, software requirements ontology, software design ontology, construction ontology, software testing ontology and software tools and methods ontology to aid sharing of knowledge. Thus, the software engineering ontology defines common sharable software engineering knowledge including those of a particular project.

Mendes et al [MAQ] use the Software Engineering Body of Knowledge(SWEBOK)[2] as a source of defined terms that can be used to communicate across people, organizations and applications. Also, it provides a shared understanding regarding the domain as well as makes explicit assumptions concerning objects pertaining to a certain domain of knowledge. Thus, knowledge already acquired, structured,

[2]Software Engineering Book of Knowledge provides a consistent view of software engineering, its boundary and contents. Available at www.swebok.org/

validated and made available by the SWEBOK is used to build and validate an ontology for software engineering using OWL.

An ontology for Software Configuration Management is presented in [SHP06]. There is need to keep track of hundreds of use cases, relationships and dependency constraints as systems grow, especially in medium and large systems. Current approaches to handle this are adhoc and proprietary. Also, detection and pinpointing of inconsistencies by human is not feasible in terms of effort and time. Inconsistencies that arise due to the large number of artifacts can be handled using ontologies. Also, added constraints causing inconsistencies are easily seen as the effect of changes on global configuration.

Thus ontologies have been used at various phases of the software development life cycle to aid in producing quality software.

2.8.3.2 UML and Ontology Development

Ontologies are increasingly used to provide semantic foundation for technologies such as software agents, e-commerce and knowledge management [MFRW00]. The Unified Modeling Language is widely used for modeling domain of a system. UML is used to develop ontologies due to the advantage in using graphical notation in modeling domains as well as for providing a standard mechanism for defining extensions for specific application contexts such as ontology modeling.

Work in literature on UML and ontologies involve using UML diagrams to extract ontology of the domain [CP99, CP02, BKK+02]. Cranefield first introduced the use of UML in ontology engineering. They introduced the similarities that exist between concepts in UML and ontologies:classes, relations, properties and inheritance. Cranefield and Purvis [CP99, CP02] use UML class diagrams for representing classes in ontologies and object diagrams for representing instance knowledge. Methods from UML activity or sequence diagrams are used to extract

properties related to an ontology [SK06].

Baclawski et al [BKK+01] propose the use of UML tools and techniques for not only visualizing complex ontologies but also for managing the ontology development process. Transformation of UML diagrams into DAML+OIL ontologies enabling the preservation of UML concepts semantics was proposed by Falkovych[FSS03]. The objective was to use ontological knowledge available in UML design documents of existing applications.

An approach to agent-oriented software engineering based on the use of UML to model aspects of multi-agent systems is proposed by Bergenti and Poggi [BP00]. The work proposes use of 'ontology diagram' wherein classes represent agents and domain entity types and associations represent domain predicates coded using Knowledge Interchange Format(KIF).

Wongthongtham et al [WC05, WCD05, WCS07] define an 'object class ontology' and 'use case ontology', built to represent classes as well as use cases of a system. Meta models of the class and use case models in UML form the ontologies. More ontologies like activity, state chart, package, sequence and collaborative ontology can be built similarly based on the meta models. The objective of the effort is to provide a means of distributing knowledge about the application domain to various users.

Thus, UML is used as an ontology modeling lanugage and the process of transformation is used for the same [CP99, BKK+02]. Approaches to transformation used for the purpose may be classified into two. First, an Extensible Style Sheet Translation(XSLT) based approach is used to generate Web Ontology Language(OWL) specifications from a UML model. The work shows usability of UML for ontology development by transforming requirements captured using UML diagrams to transform into Semantic Web languages like RDF Schema, DAML, DAML + OIL. A second aproach to facilitate transformation is to introduce ex-

tensions to UML through profiles [BKK+01, BKK+02]. A UML profile provides an extension mechanism for customizing UML models for particular domains. Stereotypes and constraints are mechanisms used to define profiles applied to specific model elements like classes, attributes and activities.

2.8.3.3 Use of ontology in testing

Ontologies used to represent knowledge of the domain can be used as a basis in generating tests to aid automation of the testing process [CP99, CP02, NPT08]. Use of ontologies in testing include[PK10]:

- Test planning and specification - The objective is to use an ontology that provides knowledge of different testing activities, their order and relationships
- Semantic querying - Using ontologies in different test activities like test plan, sepcification, execution and result evaluation enables automatic generation of test process documents. The objective is to use semantic querying to retrieve test process information.
- Test generation - The objective is to use different test generation methods based on type of test to be performed.
- Test oracle - An oracle judges results of test execution deciding whether a test passed/failed and the judgement is based on a set of criteria. The objective is to specify evaluation criteria of each test type in the ontology for use by automated oracle.

Bezerra et al propose an integrated software test ontology of Linux systems. It focuses on three ontologies, OSOnto (Operating System Ontology), SwTO (Software Test Ontology), SwTOi (Software Test Ontology Integrated). The Software

Test ontology is built using SWEBOK as the basis to extract concepts related to testing Linux systems as well as the relationships between them[BCO09]. The ontology is used by a test sequence generator to generate tests.

Work in literature also propose ontology based test case generation technique for testing web applications[PK10] and web services[WBLH07, BLTC08]. Paydar et al[PK10] present use of ontologies for web application testing. Wang et al[WBLH07] use a Petrinet model that captures the structure and operational semantics of a web service described using OWL-S(Web Ontology Language for Web Service). Based on the same, test processes are generated to cover various execution paths. Inputs, Outputs, Preconditions and Effects(IOPE) of a service functionality are described using an OWL ontology with OWL-S. Test data is generated by reasoning of the IOPE ontology. Bai et al[BLTC08] build a Test Ontology Model(TOM) to specify test concepts, relationships and semantics for both test design and execution.

2.8.4 Discussion

Ontologies have been used in the area of software engineering across various phases of software development, namely, requirements, design, development, configuration management and testing[HS06]. Ontologies are also suited to capturing requirements and changes in requirements. Further, the representation of requirements using ontologies is easily understandable and reduces language ambiguity[PT06]. Besides, traceability as well as automated validation and consistency checking are important benefits compared to a semi-formal approach like UML. Also, an ontology based approach helps in reusability of components by providing knowledge based querying mechanism. In case of development, ontologies can be used to generate code. Kalyanpur et al[KPBP04] present a method of generating Java code from OWL ontologies. The advantage of this approach

is the consistency of code with the specifications as well as the possibility of generating online documentation. With reference to software configuration management, ontologies have been used to deal with requirements and the relationships and dependency constraints as the system grows. Ontologies have been used for test case generation for web services[WBLH07, BLTC08] and for multi-agent systems[NPT08]. Thus, ontologies can be used at various phases of the software development lifecycle.

In case of capturing requirements, work in literature look at using UML for ontologies in two ways. As mentioned before, Cranefield et al[CP99, CP02] use UML as an ontology representation language by defining mappings between language constructs. However, Baclawski[BKK+01] use UML as modeling syntax for ontology development. In case of the using mapping between language constructs, an Extensible Style Sheet Translation(XSLT) based approach is used to generate Web Ontology Language(OWL) specifications from a UML model. Classes extracted from class diagrams form concepts(classes) in an ontology and attributes and methods(from activity/sequence diagram) form properties. However, in this case only partial transformation is possible as they cannot use the full meta-model. For example, it is not possible to translate the cardinality(roles) of associations to an ontology representation. Requirements thus captured using UML diagrams are transformed into Semantic Web languages like RDF Schema, DAML, DAML + OIL. However, the models thus generated have to be refined further by an ontology specialized tool(ontology editor) for use i.e. produced ontology needs to be additionally refined further by ontology development tools. Extensions to UML profiles [BKK+01] has been proposed to overcome the disadvantage in transformation. This approach despite having additional overhead in extending the UML with profiles has the advantage of representing an ontology that needs limited refinement. The Ontology Definition Metamodel (ODM)[odm06] has been proposed

to standardize mappings between knowledge representation and conceptual modelling languages[HS06]. It specifies a set of MOF metamodels for RDF Schema and OWL, informative mappings between those languages, and profiles for a UML-based notation.

One of the important challenges facing testing is how to build software with good quality at reduced cost, time and effort. Various processes, methodologies and techniques have been proposed to answer the above. A recent development is to use ontologies to aid the process of testing. Ontologies find use in generating basic test cases either for web services or multi-agent systems. UML diagrams are used for testing at various levels: class and state diagrams at the unit level, class, sequence or activity diagrams at the integration(sub-system) level and the use case and activity/sequence diagrams at the system level. As far as our knowledge goes, the usage of ontologies evolved from UML specification, particularly for testing has not been explored. One way of using information extracted from UML diagrams into ontologies is to extract data namely, classes, attributes, methods and the relations between them to form an ontology for test management. The objective is to use semantic querying to retrieve test process information to aid test engineers in planning the testing activity.

2.9 Conclusion

In summary, contemporary work on testing using UML for specification focuses on issues like consistency management, scenario generation, scenario prioritization and selection. However, there is room for further investigation like:

- Software development being iterative, there is need to introduce consistency management techniques that help the designer decide rules that need to be applied and select models based on the need of the organization, customer

and project. Also, there is need to integrate consistency management tools with modeling tools.

- Involve customers at every phase of development by using UML diagrams besides use case diagrams. One of the biggest reasons for software failure is mismatch between customer requirements and delivered software. The objective of involving the customer is to aid better communication to achieve quality of software, here, meeting customer requirements.

- Current scenario generation techniques generate an exhaustive set of scenarios which is impossible to test given constraints of cost and time. Hence, there is need to generate scenarios in an effective manner to aid testing.

- The order of execution of test cases helps early detection of defects improving the defect detection-bug fixing cycle. Most prioritization techniques use customer inputs for prioritization or are execution based. In case where UML models are used, customers assign priority or risk values to use cases/activities/scenarios which is used as the basis for prioritization. However, in case of large systems, assigning priority to all scenarios and risk values to activities will become tedious. Hence, there is need to evolve an automated technique to prioritize scenarios for testing.

- Exhaustive testing being impossible, there is need to select a subset of scenarios for testing. Distance measures are used as the basis to determine similarity between scenarios. To pick scenarios that best represent a set of similar scenarios, random selection is used. There is need to introduce heuristics to pick scenarios as well as improve the selection process to make testing effective.

- Semantic querying to obtain test information is not possible in current frame-

work. To overcome this, ontologies can be used to capture information of the domain to aid the process of software development. Semantic querying of ontologies help obtain information that helps in better management of testing.

Chapter 3

Checking consistency of specification[1]

Specification based testing is an approach that uses specification, represented as a model, to generate test cases. The quality of test cases thus generated is highly dependent on the quality of specification[Bei02]. The Unified Modeling Language (UML) is a general-purpose visual modelling language that is used to specify, visualize, construct and document the artifacts of a software system[JRB01]. The primary issue related to quality is consistency of specification captured using the model. Need for consistency checking between UML diagrams are discussed. Rules are defined for checking intra-model and inter-model consistency between UML diagrams are presented and a transformational approach to consistency checking is proposed alongwith a prototype tool.

[1]A part of the work stated here is reported in
[1] Sapna P.G., Hrushikesha Mohanty, 'Ensuring Consistency in Relational Repository of UML Models', *in Proceedings of the 10th International Conference on Information Technology (ICIT'07)*, pp.217-222, IEEE Xplore Digital Library.

3.1 Introduction

The objective of software development being delivery of quality software to the customer, the need to capture requirements without errors is of primary importance. Inconsistency management has been defined in Finkelstein et al.[FST96] as 'the process by which inconsistencies between software models are handled so as to support the goals of the stakeholders concerned'. A number of techniques and tools have been developed for capturing requirements to suit the needs of different domains and applications.

The area of software specification has two classes of techniques: formal techniques (e.g. Z which emphasizes formality at the cost of ease of use and understandability) and rigour based techniques (e.g. UML, which emphasizes ease of use and understandability, at the cost of formality). As a rigour based technique lacks mathematical foundation, it is free to be interpreted differently by different people, leading to inconsistency. The Unified Modeling Language (UML) is used to specify, visualize, modify, construct and document the artifacts of a software system and is used at different levels and stages of the software development life cycle. UML models presented by different diagrams view a system from different perspectives or from different abstraction levels[SKS]. Thus, the various UML models of the same system are not independent specifications but strongly overlapping i.e. they depend on each other in many ways. Therefore, a change in one model affects another. The usage of several diagrams within the UML to capture specification leads to inconsistency that may arise due to mismatch between different diagrams. This has given rise to the need for checking consistency of requirements captured using different diagrams.

Consistency management is a process comprising activities like specification and management of consistency handling policy, consistency checking, detection of overlaps, diagnosis and consistency handling and tracking [ZK01]. A model

is consistent when it conforms to the semantics of all the domains involved in the development process. e.g. application domain, modelling domain, language domain[SC02]. Inconsistency arises when some model expression violates a principle like same name for attribute and method. The Unified Modeling Language (UML) defines well formedness rules as constraints on the modeling primitives to prevent errors in modelling. However, they do not suffice in detecting inconsistencies, especially those across diagrams. Therefore, additional constraints are required to ensure consistency and completeness to help detect and prevent errors during specification of a system.

There are different dimensions to consistency checking[Sha06]: intra-model Vs inter-model, structural/static Vs behavioural/dynamic, horizontal Vs vertical. Intra-model consistency is concerned with the issue of consistency between diagrams of the same type e.g. consistency between use case diagrams. Inter-model consistency looks at the way each model is designed with reference to the other models so as to be meaningful. Structural/static consistency concerns consistency among the structural elements used in representing requirements. Behavioural/-dynamic consistency concerns checking consistency during run time. Horizontal consistency checking involves checking consistency between diagrams at the same level of abstraction whereas vertical consistency involves checking consistency between diagrams at different levels.

Example:

An example for inconsistency in UML diagrams is shown in Figure 3.1 and Figure 3.2. Figure 3.1 shows a sequence diagram for the scenario, Customer places a order. Also, the classes involved in the scenario are shown using the class diagram in Figure 3.2.

A customer places an order. The order is checked with the items in the inventory and a notification is given. The customer then confirms the order after

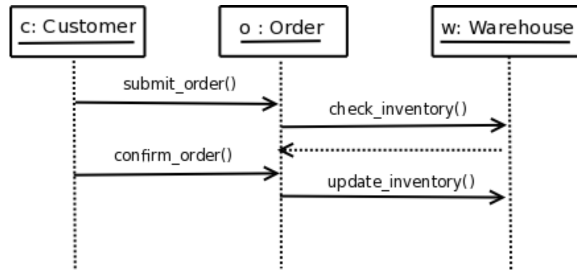


Figure 3.1: Sequence Diagram for 'Customer places an order'

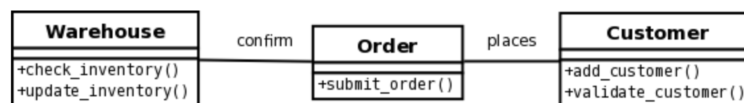


Figure 3.2: Fragment of Class Diagram related to 'Customer places an order'

which the inventory is updated. A comparison of elements between Figure 3.1 and Figure 3.2 show that the sequence diagram does not match the operations stated in the class diagram. Rule 6 states that a class in a class diagram must have an operation that corresponds to a message in a sequence diagram. In the example, the message, confirm_order() is not specified in the class diagram. This is a case of inconsistency between the sequence diagram and class diagram. There is a need to add the method confirm_order() in the class diagram.

UML defines constraints on models as well formedness rules. These well formedness rules are applicable at the intra-model level and not at the inter-model level. Current work in literature handles inconsistency between UML diagrams based on the interaction specified between models in the UML superstructure specification[uml07]. As navigation between metamodels of all diagrams is not present, it is not possible to check for consistency of specification in a wholesome manner (e.g. no relation between use case and sequence diagram). In this work, interactions between the metamodels is introduced by annotating model elements. A set of rules are written based on the introduced interactions to check for consistency of specification. Also, a prototype tool is built to validate the approach. The approach shows the possibility of increasing consistency among UML diagrams sig-

nificantly thereby ensuring completeness and consistency of specifications leading to customer satisfaction.

In this chapter, rules to manage inconsistencies that can occur when capturing specifications using the UML are presented. Section 3.2 elucidates different consistency dimensions in the context of UML. Section 3.3 explains how the Object Constraint Language(OCL) aids in expressing constraints on UML modelling elements. A survey of work in the area of consistency management with focus on UML models is presented in Section 3.4. This work looks at two aspects of consistency management, namely, intra-model and inter-model consistency checking(Section 3.5).

Focus is on checking structural consistency, between UML models using a transformational approach discussed in Section 3.6. In this work, consistency checking activity is approached as a two pronged approach: first, rules for inconsistency checking are defined using OCL terminology; second, repository is created and rules are implemented as SQL queries, triggers and views discussed in Section 3.8. The architecture of the prototype tool and its usage is also discussed. Finally, Section 6.6 summarizes the contribution of this chapter and introduces issues to be tackled in the next chapter.

3.2 UML and Consistency Checking

The Unified Modeling Language (UML) is used as a standard for modeling object oriented software. UML helps in modeling different aspects of a system through the use of various diagrams, like use case diagrams, activity diagrams, sequence diagrams, class diagrams and component diagrams. Each aspect of a system is represented using a particular type of UML diagram and a set of such diagrams is termed as a model.

UML diagrams represent two different views of a system model, namely, static (structural) view and dynamic(behavioural) view[BRJ05]. Static view emphasizes the structure of the system while the dynamic view tries to project the run time interactions of the system. Static diagrams include class diagram, object diagram, component diagram, deployment diagram, package diagram, composite structure diagram and profile diagram. Dynamic diagrams includes use case diagram, activity diagram, sequence diagram, statechart diagram, collaboration diagram, interaction overview diagram and timing diagram.

The advantage of using UML is the fact that different diagrams can be used to model varying aspects of a system. i.e. the diagram that best represents an aspect of the system to be captured can be used. This same aspect leads to difficulties too, in the form of maintaining completeness and consistency within and between UML diagrams. This problem is further magnified as software systems are built today in a manner that requires collaborative and collective effort of people physically distributed across many locations. The phases of requirements gathering, analysis and design involves more than one UML model. Any error in these initial phases of software development will be carried over through the rest of the software development phases and will require major effort to be corrected. The following are the factors that have been found to lead to the problem of inconsistency:

- Omission
- Lack of Standardization
- Multiplicity of Stakeholders
- Addition of new features during system evolution
- Interdependency of diagrams
- Overlapping of model elements

Erroneous models have a huge impact on the development process in terms of added cost, time and effort. Therefore, the need for consistency management arises.

In this work, a subset of the UML models that are widely used has been selected with the objective of studying the points of inconsistency and proposing ways to overcome them. The models chosen are, both static and dynamic, namely, use case, activity, sequence, class and state diagrams. Being the widely used set of diagrams, the objective was to study the points of inconsistency and ways to overcome them.

The problem can be specifically broken down into two issues:

1. How do we ensure that the model captures the specification of a system ?
2. How do we check that the models capturing specification are consistent, both, within and between themselves ?

The first question requires that each use case diagram capturing requirements be elaborated by one or more diagrams that capture the dynamic view. For example, each use case in a use case diagram must be elaborated by one or more activity diagrams. In case of the second question, the objective is to ensure intra-model and inter-model consistency. An example for intra-model consistency in the case of use case diagrams can be that two uses cases with the same name must refer to the same requirement. An example for inter-model consistency is 'a function belonging to a class in an activity diagram must have a corresponding method in a class in the class diagram'.

To check whether a model is consistent(intra-model consistency), the following questions need to be answered:

1. Is a diagram well defined ?

2. Is a diagram consistent with other diagrams in the model ?

To check whether models are consistent among themselves (inter-model consistency), questions that need to be answered are:

1. Is the class diagram consistent with the sequence diagram ?
2. Is the class diagram consistent with the activity diagram ?
3. Is the state diagram consistent with the class diagram ?

UML being a semi-formal modelling language, it is not possible to define all aspects of a specification. To aid specifying constraints on modelling elements, the Object Constraint Language(OCL) is used which is the focus of the next section.

3.3 UML and the Object Constraint Language

The Object Constraint Language (OCL)[ocl06], is a formal language used to describe expressions on UML models. UML diagrams are not expressive enough to provide all aspects of a specification. Hence, the need to describe additional constraints about objects in a model. OCL is used as a specification language along with UML to augment the expressiveness of UML. Constraints are expressed as an OCL rule of the form:

```
context <model element>
invariant <constraint name> :
OCL expression
```

OCL can be used as a query language, to specify invariants, to describe pre and post conditions on operations and to describe guards. Consider the example given in Figure 3.3. The class 'Person' with attributes and method is shown. Suppose, it is required that an employee must be above 18 to be employed by the company,

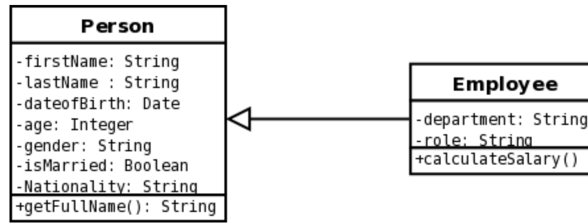


Figure 3.3: Representation of Class 'Employee'

then, this constraint on the age of the person cannot be represented through the class represented in the class diagram. Therefore, there is need for a language to describe additional constraints about the objects of the class 'Person'. The OCL invariant to specify the constraint is given below:

```
context Person
invariant checkAge:
self.age > 18
```

OCL also provides constraint and object query expressions on any Meta Object Facility(MOF) model or meta-model which otherwise cannot be expressed by the diagrammatic notation. Inter-model diagram navigation involves following links from one object to locate another object or a collection of objects. OCL can be used for the purpose of navigation between model elements. A constraint on a model can be defined as a restriction on one or more values of a model. In this work, OCL constraints are used to define consistency constraints on a model.

3.4 Related Work on Consistency Checking

A set of UML models is consistent when there are no conflicts between two arbitrary models in the set, which are presented by different UML diagrams[Shi06]. Conflict between models can be identified in many ways, e.g. method belonging to a class shown in the sequence diagram but not in the class diagram, same name for attribute and method of a class.

One of the important concerns of system modelling has been consistency. Different approaches on how to detect inconsistencies at the intra-model and inter-model level have been proposed. The broad classification is a) Direct Approach and b) Transformational Approach. Direct approach involves comparing design models directly [KK03, HHKT02, LLH01] whereas the transformational approach involves transforming design models from one to another or into some intermediate model, for comparison[TE00, Egy01, HS04, KC04]. Examples of intermediate models used in the transformational approach include graphical models, formal methods, petrinets and knowledge representation based techniques.

Direct Approach

Constraints defined on UML using OCL is used for consistency checking by [KK03]. Constraints written in OCL is subjected to syntactic and view dependent analysis using the Dresden OCL Parse Toolkit. The analysis is based on information about the metamodel (entities, attributes and links). Hnatkowska et al[HHKT02] define a consistency checking pattern consisting of the specification phase, algorithmization phase and checking phase. They define two types of constraints, general and specific on three layers, namely, system model layer, model layer and diagram layer. Li et al[LLH01] present a method to check static consistency between Use Case(UC) and Conceptual Model(CM) by defining semantics using pre and post conditions and state invariants.

Transformational Approach

In the transformational approach, conversion to graphical models is the basis for work done by [CPC+04, Rou03, TE00]. Formal approaches involves use of a formal notation like Z. Kim and Carrington[KC04] use Object Z to define integrity consistency constraints of UML model elements at the language level. Derrick et

al[Der02] develop a framework based on the Open Distributed Processing (ODP) standardization initiative. Consistency checking is done across viewpoints (class, state and sequence diagrams) using Object Z and LOTOS. Petrinets are used as the intermediate models for consistency checking by [HS04, SM00, BDM02, ZS02].

Egyed in [Egy01] proposes the VIEWINTEGRA approach where source diagrams are transformed into the diagram type of the target. This is done so that transformations of the source diagrams are conceptually close to target diagrams they need to be compared with. Then consistency comparison is done to compare the transformation of the source diagram with the target diagram. For example, if a sequence diagram is to be compared with a class diagram, then the sequence diagram would be transformed to an 'interpreted' class diagram which is then compared with the target class diagram or vice versa. The class, sequence, collaboration, object and statechart diagrams are taken into consideration.

Despite lot of work done in the area, there are still problems that need to be solved:

- UML superstructure specification[uml07] document defines rules for intra-model consistency while constraints for elements across models have not been specified.
- Even though using constructs of UML(meta-model) for checking consistency reduces the cost and time involved in the process, the relationship between the models is such that it does not allow for direct checking between all models. Similarly, though the OCL can be used to specify elements and relations, it is difficult to use the same in specifying the consistency rules between diagrams.
- The transformational approach converts the UML models into a secondary model and use an intermediate means like trees and graphs for verification.

The advantage provided by the transformational approach is in having the same intermediate model for all the diagrams(e.g. Petrinets). Also, once the rules for transforming a model are decided, it becomes easier by using automation techniques. Also, models can be forward and reverse engineered. The disadvantage of using the transformational approach is having to implement the transformation in addition to consistency checking resulting in larger overheads.

- Common CASE(Computer Aided Software Engineering) tools do not support consistency checking completely.

3.5 Relationship among UML Models

Based on interviews with UML practitioners, clients and a web survey, Dobing et al. and Nugroho et al[DP06, DP08, NC08] observe that use case diagrams and class diagrams have highest usage levels followed by sequence,activity and state diagrams. The class, sequence, activity and state diagrams provide additional information of requirements not captured by use case diagrams. Based on the same, it was decided to concentrate effort on checking consistency among five diagrams, namely, use case, activity, sequence, class and state diagrams. Using the UML superstructure specification as well as work by Dobing et al and Nugroho et al, the relationship between the models was studied with consistency checking as the goal. The strength of inter-relationships between diagrams is shown in Table 3.1 based on [SKS, uml07].

The strength of the relationship between the five diagrams(Table 3.1) is classified as 'Strong', 'Medium' and 'Weak'. 'Strong' indicates that both diagrams contain the same model elements to a large extent. 'Medium (A)' indicates that annotation is required or additional information is required without which the

Table 3.1: Strength of inter-relationships among models

	Use Case	Activity	Sequence	State	Class
Use Case	X	*Medium (A)*	*Medium (A)*	-	-
Activity	*Medium (A)*	X	-	Weak	Strong
Sequence	*Medium (A)*	-	X	Weak	Strong
State	Weak	Weak	Weak	X	Strong
Class	*Medium (A)*	Weak	Weak	-	X

relation will be weak or non existent. For example, lets take the relationship between use case and class diagram. A class diagram must be annotated with the use case it belongs to, for otherwise, the relationship does not exist. A 'weak' indicator means that the diagrams have very little common information. Again, the relationships need not necessarily be bidirectional. For example, in Table 3.1, the relationship between State - Class is defined as strong, whereas, the relationship between Class - State is not existent. Based on Table 3.1, a diagram showing relationship among the models was evolved which is depicted in Figure 3.4.

The relationship between model elements shows that the strength of inter-relationship between diagrams varies. Checking consistency using a direct approach [CPC+04] is possible in case of intra-model consistency checking as well as between static and dynamic models that share elements in the metamodel. Hence, using a direct approach is not best suited as it is not possible to check for consistency between all model elements. Transformational approach involves transforming UML models to either a) graphical model like Petrinets and using the primitives of the graphical model to check for consistency or b) formal method and using a model checker to ensure consistency. Both ways, the objective is to use the capabilities of the model/formal method in checking consistency. In this work, the objective is to use models, static and dynamic, both strongly and weakly connected(e.g. use case and activity diagram) to capture requirements of a system by annotating the models. The advantage of a transformational approach is that

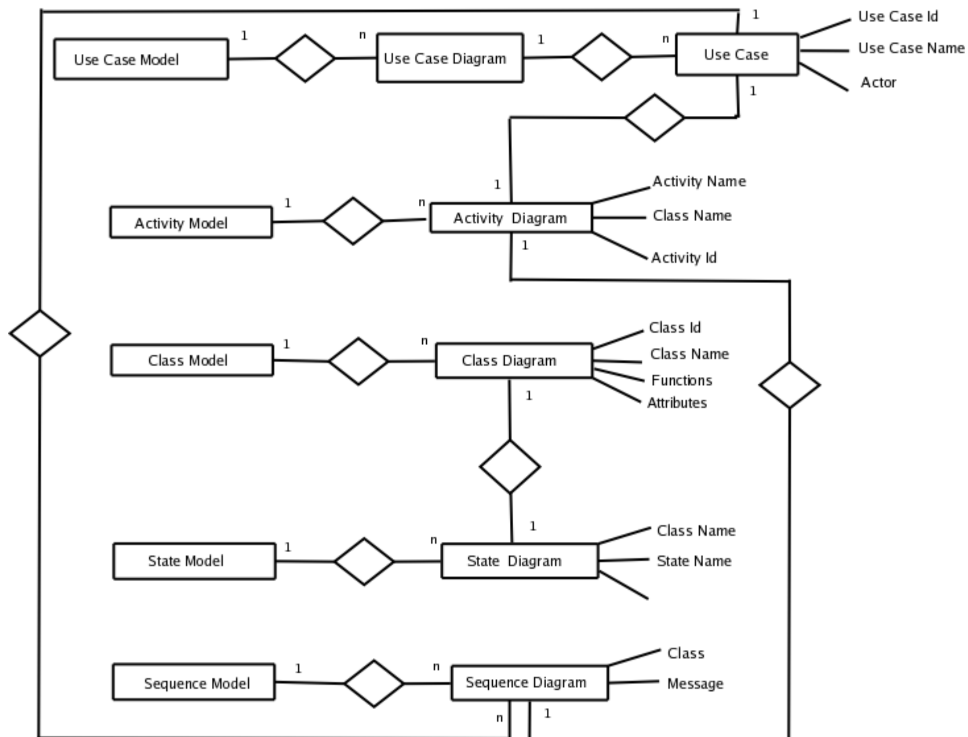


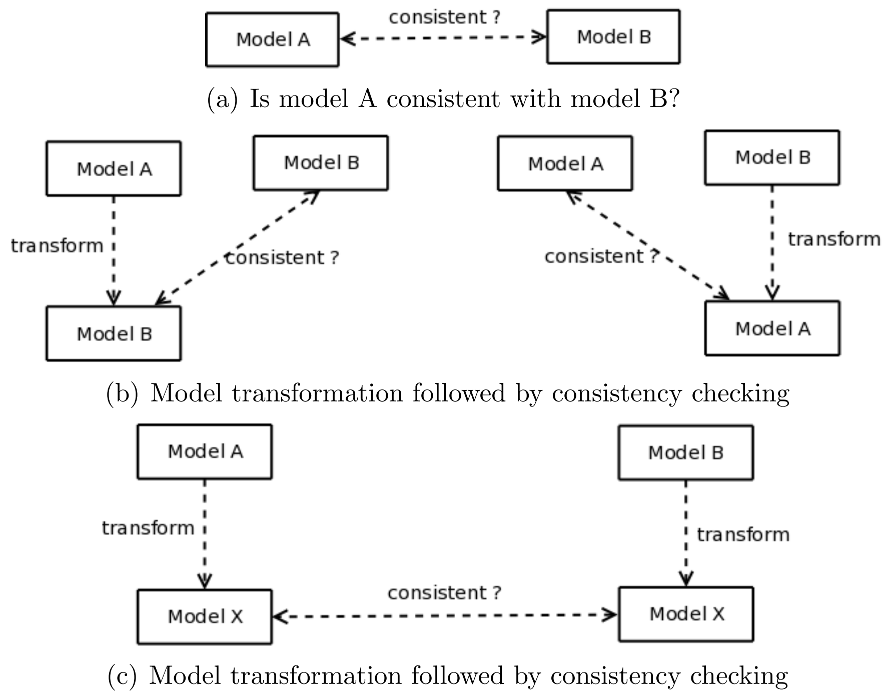
Figure 3.4: Relationship among UML models

inconsistencies are easier to find in case of diagrams of the same type [SKS]. As UML consists of different types of diagrams, a transformational approach is best suited to check for consistency by transforming the diagrams into a common form.

3.6 Transformational Approach to Consistency Checking

Transformational approach to checking consistency between models can be done in two ways as shown in Figure 3.5. In the first case, one model is transformed to another and checked for consistency as shown in Figure 3.5(b). Figure 3.5(c) shows the second case which involves transforming both models to a common model before checking for consistency.

Consistency constraints are conditions imposed on the use of notations, vari-



(a) Is model A consistent with model B?

(b) Model transformation followed by consistency checking

(c) Model transformation followed by consistency checking

Figure 3.5: Transformational Approach to Consistency Checking

ables and names, types and symbols so that a set of well-formed diagrams can be regarded as a meaningful model[BBG05]. Conditions can be imposed on the structural as well as semantic aspects of the diagrams. To enable automated checking of a model in an effective, efficient and simple manner, consistency constraints are written as rules.

Demuth et al discuss the use of UML and its constraint language, OCL, for the design of integrity constraints for relational databases in [DH99]. They provide a methodology for translating invariants written in OCL to SQL. Rule based approach to consistency checking is used by Chiorean et al[CPC$^+$04] and Zapata et al[ZGG07]. Chiorean et al use OCL to check for syntactic consistency among class diagrams. For this, they use Well Formed Rules(WFRs) written in OCL specified in the UML superstructure document. Zapata et al use rule based approach to look at the issue of inter-model inconsistency. They define rules to check consistency between UML use case and class diagrams designed using ArgoUML

and stored in XMI format. Mapping is done between use case and class diagrams by the condition that the noun in each use case become a class and the verb, a method of the class. Rules written using OCL are then translated to XQuery and XPath expressions used for validation. Alexander Egyed[Egy06] defines rules at the instance level for consistency checking between sequence, class and state diagrams. Consistency checking is done instantly as the model is developed thereby providing immediate feedback to the analyst. However, the evaluation time across size of the model remains a constant.

Different work, thus provide solutions to ease the consistency checking process. However, there are limitations. Chiorean et al propose using WFRs to check consistency. However, the WFRs are for class diagrams and hence consistency is ensured at the intra-model level only. Zapata et al present a novel approach to checking inter-model consistency between use case and class diagrams by mapping the noun in the use case to class name and verb in the use case to method name of the class. The approach requires that method name in a class be directly linked to a use case, introducing an additional constraint of having a use case for every method belonging to a class. Besides, the noun in the use case is the name of a class in the class diagram. The drawback of the technique is that for every class and method to be defined, there is need to define a use case in the use case diagram. The technique introduced by Zapata el al requires that use case diagrams be defined at the lowest level of granularity which is tedious and cumbersome. Egyed [Egy06] introduces an instant consistency checking approach providing the analyst with immediate feedback in case the model is inconsistent. In this approach, there is no way to predict which model elements are accessed for a consistency rule in advance. Also, there is need to study the need for instant consistency checking as the model is being developed versus applying consistency checking on the basis of need. The VIEWINTEGRA approach does not have the overhead of a third

party language in the process of transformation. However, in order to check for consistency between different diagrams, a minimum of ten transformation methods is required. In case pair-wise comparison between diagrams is to be done, then 55 transformation methods are required to check consistency between eleven models. The drawback of the method is the overhead involved in transformation and the accuracy of the transformation between diagrams.

In this work, the third approach(Figure 3.5(c)) shown in Figure 3.5 is adopted. Relational databases are used as the common model to check for inconsistency. Relational databases provide mechanisms for managing data. There are several advantages to storing UML models within a repository. First, relational databases guarantee a high level of interoperability. Also, a shared database ensures cooperation of designers and developers in the development of a system. Secondly, storage of designs in a repository helps in reuse. Third, reliability and scalability of data is ensured when a database is used. Navigation in RDBMS is provided through mechanisms like defining relations in the form of tables, view and joins. Other advantages provided by relational databases applicable for use in consistency checking is use of query language, SQL and support of views and scalability. UML is used to model small to large applications. Therefore, the amount of data can become voluminous making it difficult to handle. Integrity constraints can be used to enforce accuracy of data pertaining to an application. Also, rules written using SQL can be used to query the database.

A relational database is used to store data extracted from the model of an application stored in XMI format. Rules are defined based on the interactions between elements in the metamodel shown in Figure 3.4. The UML metamodel being clearly defined, the number of rules to be defined is static. Also, rules help in ascertaining the level of consistency to be achieved dependent on attributes of the project like size and complexity. The rules have been formulated by studying

the UML diagrams and looking at the constraints between them and are written using the Object Constraint Language(OCL). Rules can be translated from OCL to SQL queries, triggers and views.

For consistency to be checked between various models of UML, there is need to link models by annotating them. Also, certain guidelines have to be followed to ensure that requirements specified are elaborated.

3.7 Design Guidelines

A model M, can be said to be consistent with reference to constraint, C, if application of the constraint to the model evaluates to true; else, the model is inconsistent. Consistency constraints help define how specification using UML models should be constructed.

The advantage in applying such constraints is that anomalies in the model can be prevented and detected. As mentioned in Section 3.1, consistency checking of UML models can be implemented at two levels: intra-model and inter-model. This work focuses on intra-model and inter-model level and looks at the issue of structural consistency between the use case, activity, collaboration, class and state diagrams.

Design guidelines are rules that hold throughout the process of modeling. The relationships required to be established between models to enforce consistency is shown in Figure 3.4. A study of the metamodels depicted in Figure 3.6 shows that all relations cannot be enforced due to the lack of common elements among models. Hence, design guidelines were introduced to enforce these relations which are vital to the integrity of the database.

The following guidelines are required to be followed when building the models.

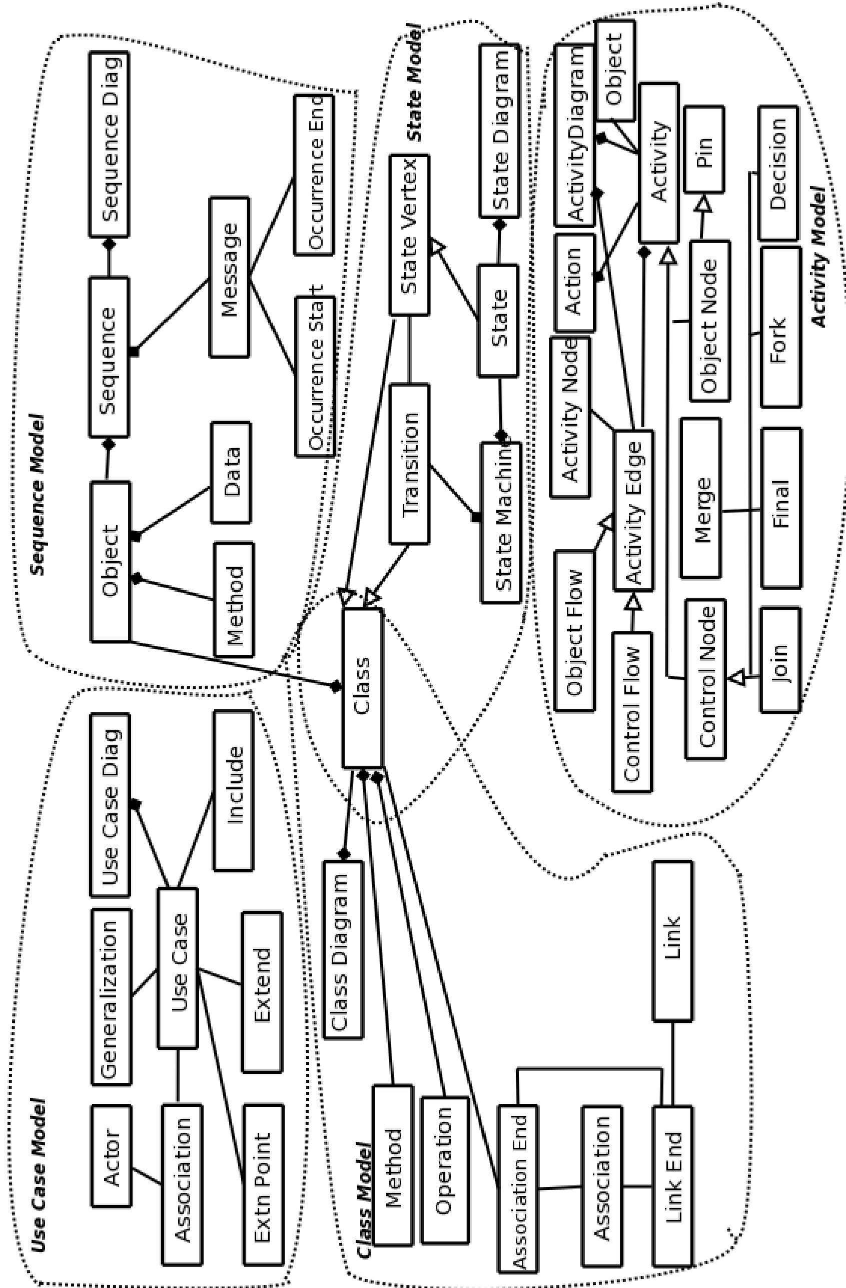


Figure 3.6: Modeling elements in the UML Metamodel

- Each use case in a use case diagram must have a corresponding activity diagram.
- Each use case in a use case diagram must have a corresponding sequence diagram.
- A class must be annotated with the name of the use case it is part of.
- Each class in a class diagram must have a state diagram.

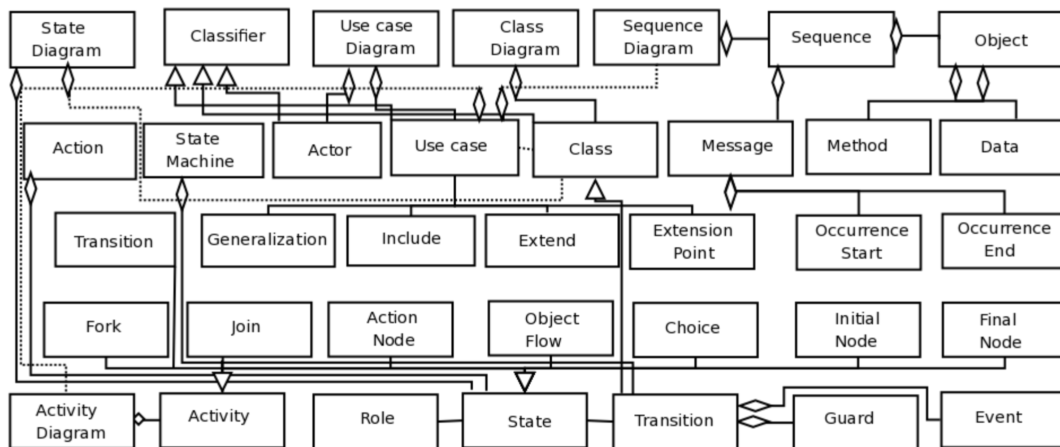
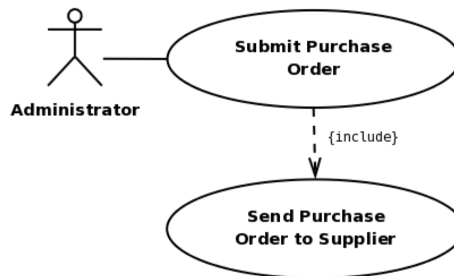


Figure 3.7: Metamodel incorporating design guidelines

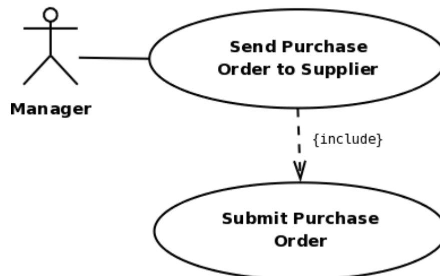
The metamodel developed based on the metamodels of the use case, activity, sequence, class and state diagrams incorporating the above guidelines is shown in Figure 3.7. Rules can be defined based on the interactions in the metamodel. Well formedness Rules(WFR) have been specified in the UML superstructure document to ensure consistency when developing UML diagrams. These rules are confined to diagrams of a particular type e.g. rules for use case diagrams. UML does not specify rules to be followed at the inter-model level. In this section, additional rules are defined at the intra-model level. Also, based on the UML metamodels and extending the idea of WFRs to inter-model level, a set of rules are formulated to check consistency between the use case, activity, sequence, class and state diagram.

3.7.1 Intra-model consistency checking rules

As mentioned earlier, intra-model consistency concerns checking for inconsistency that may exist within a model. For example, consider the use case model of a system shown in Figure 3.8, consisting of two use case diagrams specifying requirements of two actors 'Administrator' and 'Manager'. Part of the use case diagrams are shown in Figure 3.8(a) and Figure 3.8(b).



(a) Use case diagram A



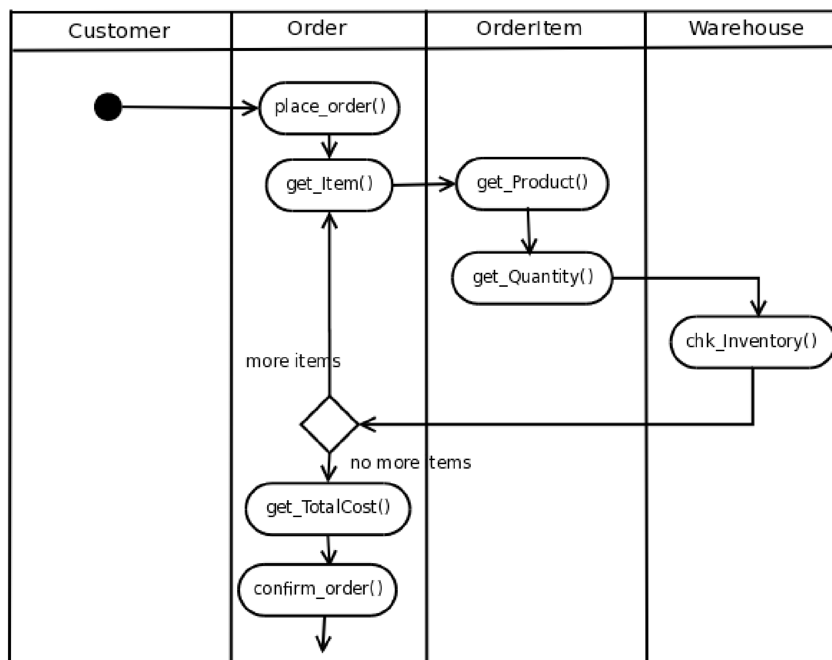
(b) Use case diagram B

Figure 3.8: An example of intra-model inconsistency

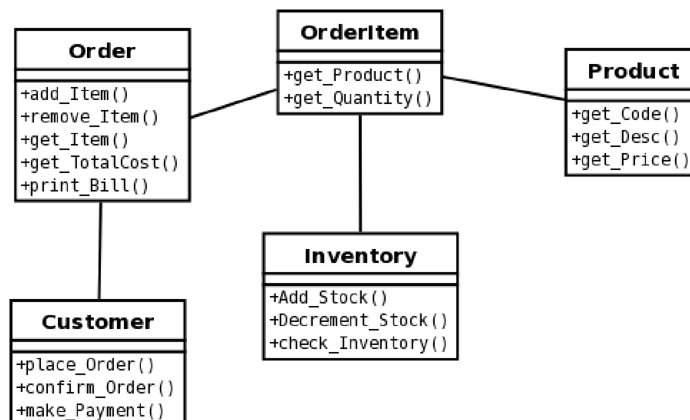
The diagrams show inconsistency with reference to Rule 10: No two use cases in a use case diagram can have the same name, unless it refers to the same requirement and Rule 8: A use case cannot include use cases that directly or indirectly include it. This is a case of intra-model inconsistency showing conflicts within the use case model. Other rules used to check for intra-model consistency in case of use case and activity diagrams are listed in Appendix A.0.4.

3.7.2 Inter-model consistency checking rules

Inter-model consistency concerns checking for inconsistency that may exist between diagrams belonging to two or more models. For example, consider the class diagram and activity diagram shown in Figure 3.9, elaborating on the use case 'Place order' by customer. Part of the activity diagram is shown in Figure 3.9(a) and the classes in the class diagram are shown in Figure 3.9(b).



(a) Part of activity diagram for use case 'Place Order'



(b) Part of class diagram

Figure 3.9: An example of inter-model inconsistency

The diagrams show inconsistency with reference to Rule 32: Each class and activity in an activity diagram must have a corresponding class and method in the class diagram. The method chk_Inventory() in the class diagram does not find a corresponding method in the class diagram. This is a case of inter-model inconsistency showing conflicts in requirements captured across UML models. A list of rules to check for inconsistency between models is listed in Appendix A.0.5.

3.8 Applying the Transformational Approach

The set of rules defined in Sections 3.7.1 and 3.7.2 are used for checking consistency between use case, activity, sequence, class and state diagrams based on the metamodel. Rules are classified as Warning, Moderate and Severe, dependent on the risk attached to a consistency rule. This classification of rules is done by the user. The inputs to the system are diagrams in XMI format. VP-UML, a Computer Aided Software Engineering(CASE) tool was used to capture requirements of a system. The UML diagrams are stored as XMI documents. Saxon parser[2] is used to parse XMI documents. Metamodel in Figure 3.7 is translated to tables. OCL constraints are translated to SQL triggers, views and procedures which are used to enforce consistency based on work done by [DH99]. The rule based system, checks for consistency and generates a report listing the instances where rules are violated. Further sections explain the steps in applying the rule based approach in detail.

3.8.1 Translation of UML model to Database Schema

The UML repository is based on the meta-model and is implemented using an RDBMS, MySQL. The meta-model is mapped to a relational database schema.

[2]Saxon Parser. http://saxon.sourceforge.net/

Data integrity is enforced by converting OCL invariants to assertions, views and triggers. The steps involved in translating a UML meta-model represented as a class diagram to a database schema is given below[DH99]:

1. Mapping association: Each class in the association relationship is represented as a table. Associations are implemented using foreign key references. In case of *one-to-one relationship*, a foreign key is defined in either class. For *one-to-many relationship*, a foreign key is defined in the class on the many side. In case of *many-to-many relationship*, an additional table is defined and the combination of primary key from each of the tables is taken.

2. Mapping inheritance: In case of *inheritance* between classes, there are three ways for mapping inheritance to a relational database.

a. One table per hierarchy

b. One table per concrete class

c. One table per class

Any or a combination of the above can be used for mapping inheritance relationship between classes to a relational database.

3. Mapping aggregation: In case of aggregation between classes, the aggregated class's attributes are put into the same table as the aggregating classs.

3.8.2 Code Generation

Structured Query Language (SQL) is used for accessing and manipulating databases. Methods for automatic translation of OCL expressions to SQL is discussed by Demuth et al [DH99, DHL01] and Mohanty et al [LBMS05] i.e. methods to translate OCL to SQL queries, views and triggers. SQL is used to check if a database satisfies the constraints as well as disallow insertion of data that does not satisfy constraints in [DH99, DHL01] through the OCL2SQL tool. An example of the

translation is given below:

Example:

(i) *Views:* A view is a virtual table that consists of columns from one or more tables and is a query stored as an object in the database. It derives its data from one or more tables. A view can be referenced like a table using queries and serves as a security mechanism by hiding data wherein users cannot view or access data in the underlying tables.

The class 'Classifier' from the UML meta-model has attributes name, id and a boolean flag which indicates whether the classifier isActor or not. This class is mapped to a single table with columns(attributes) name, id and isActor respectively. An OCL constraint on the class 'Actor' is 'All actors should have a name' i.e. not empty. The OCL expression is given below:

context Classifier

inv actorNotNull: (isActor=true) implies (self.name ⟶ notEmpty())

The translation of this invariant into corresponding SQL view involves evaluating all tuples of the table Classifier where the type of classifier is actor and name is not empty.

create view actorNotNull as

*(select * from Classifier*

where not ((isActor=true) and (name is not null));

(ii) *Triggers:* A trigger is a statement that gets executed automatically when a database is modified. A trigger defined on a class executes whenever the specified event occurs and the corresponding condition is satisfied.

Consider the rule where every activity diagram must be associated with a use case in the use case diagram. Hence, when an activity diagram is added, there must be a corresponding use case that it is associated with. This rule when converted to trigger is represented below:

```
create trigger Insert_AD
before insert on activitydiag ad
fore each row
begin
if ad.fk_ass is null
dbms_output_put_line(Each activity diagram must be associated with a use case)
end if;
end
```

3.8.3 Consistency Checking - The Process

The consistency checking process is shown, as a UML activity diagram, in Figure 3.10. The customer provides the requirements of a system. The requirements analyst, then extracts the specification and captures it using UML diagrams. The diagrams stored in XMI format are extracted by the consistency checking tool and stored in a model repository. The analyst then schedules execution of rule either by partial batch, full batch or user selected evaluation. Full batch involves execution of the complete set of rules whereas partial batch involves executing a class of rules like intra model rules pertaining to a particular diagram. User selected evaluation involves executing one or more rules based on user selection. Rules are executed and in case inconsistency is detected, details stored in the error file. The process continues until all rules are executed.

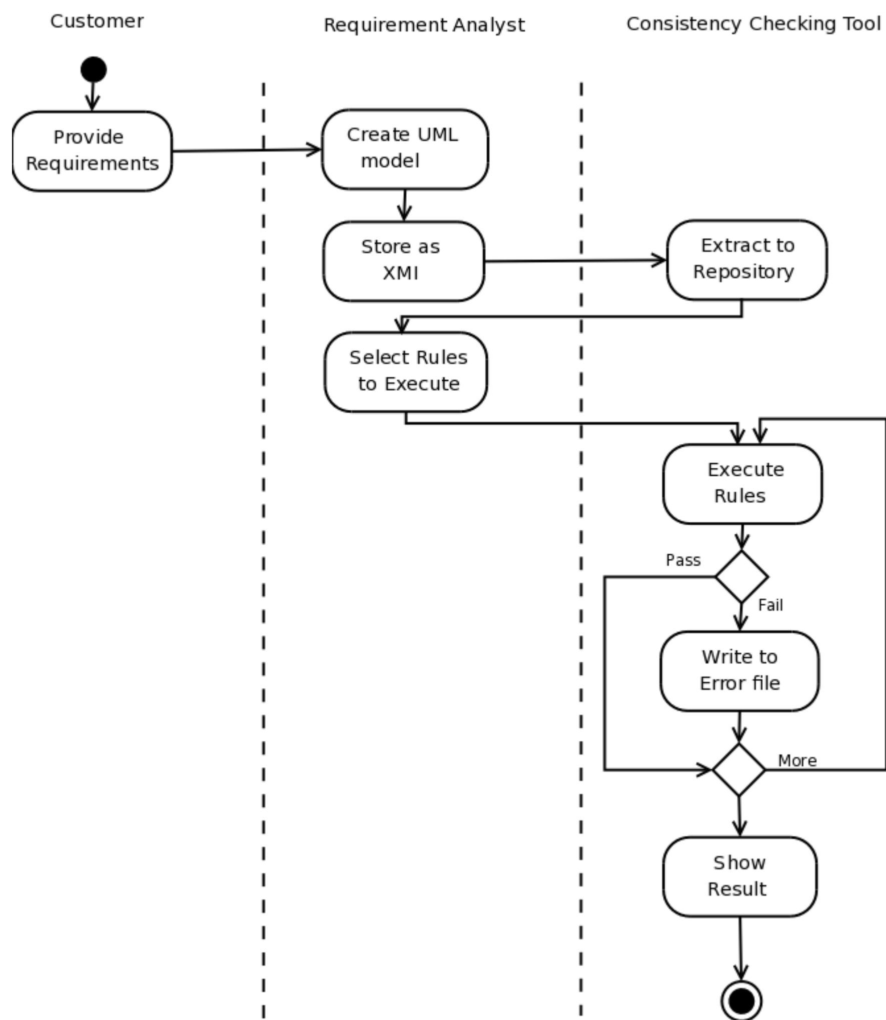


Figure 3.10: Consistency Checking Process

3.8.4 Prototype tool

A prototype tool is built to support the consistency checking process between UML diagrams. The architecture of the tool is shown in Figure 3.11.

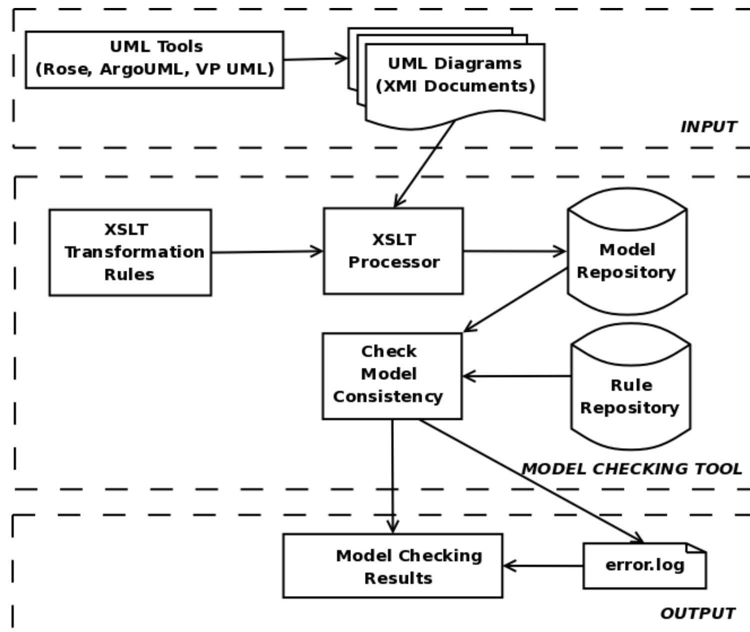


Figure 3.11: Architecture of Prototype Tool for consistency checking

The UML model that is input to the tool as XMI format is parsed and stored in the model repository. The user can decide to select a set of rules for execution or perform batch execution(all rules). Result of consistency checking, stored as an error file, is given to the user indicating the rule checked, the model element that has failed the rule and corresponding severity.

The process of checking the consistency of a UML model with regards to a set of rules is divided into two phases. The first phase is the extraction of data from XMI files, creation of the model repository and compilation of the rules in SQL creating the rule repository. Figure 3.12 depicts the selection of the application folder, for extraction into the model repository. This phase must complete successfully in order to allow the start of the second phase, namely, consistency evaluation.

The second phase involves the execution of consistency checking rules. As

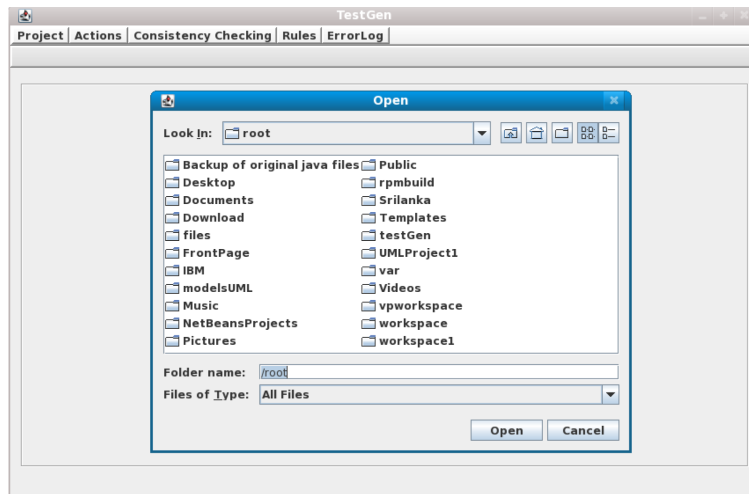


Figure 3.12: Selection of application folder to check for inconsistency

mentioned earlier, there are two ways of performing consistency check: complete and user selection as shown in Figure 3.13. Complete involves executing all rules whereas user selection requires the user to select one or more rules for execution.

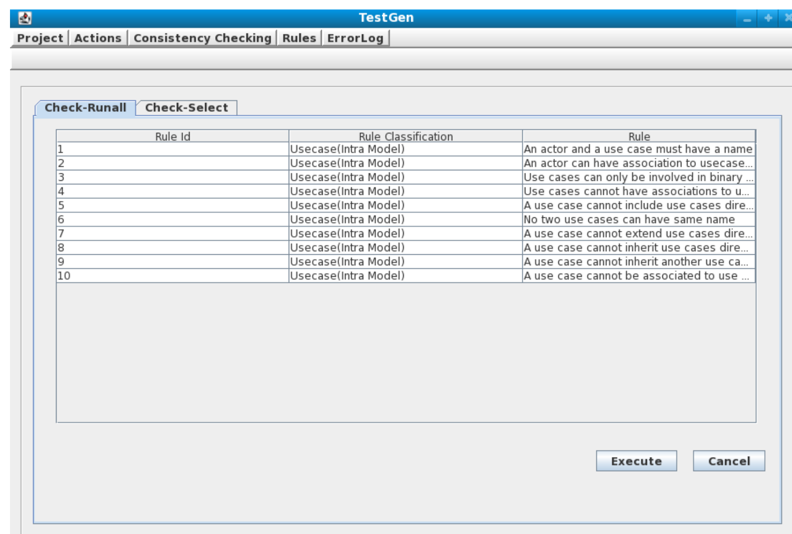


Figure 3.13: Selecting rules to check inconsistency

Once rules are executed, results of execution are reported with a list of rules that failed with respect to the model elements. Detailed information on the results(Figure 3.14) of consistency checking is provided showing the classification, model element failed as well as the severity. The results show the inconsistent

parts of the specification to the analyst.

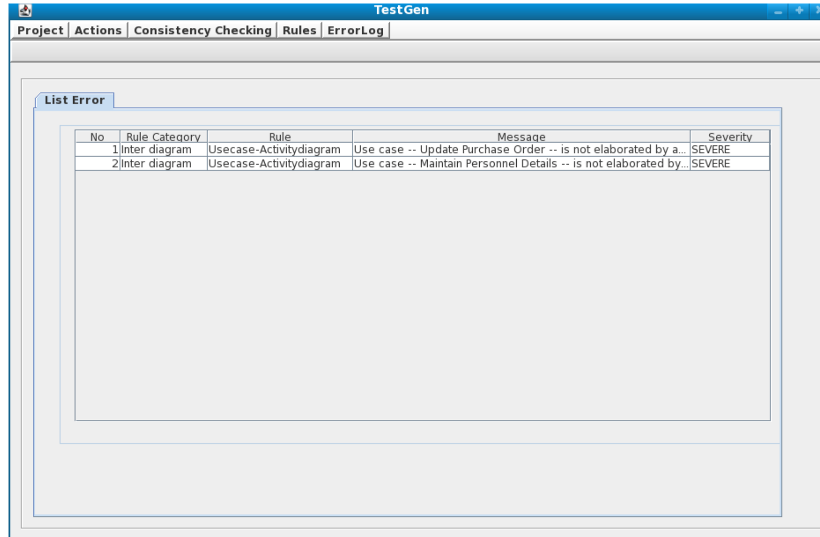


Figure 3.14: List of errors after application of rules

3.9 Summary

This chapter presented a transformational approach to consistency checking between UML diagrams, both at the intra and inter-model level. A study on the relation between UML and consistency checking as well as OCL is presented. Based on the study of relationship between UML models, a set of design guidelines were introduced to be followed when capturing specification using UML. A transformational approach was preferred due to the advantage of having a common intermediate form. A set of rules, both at the intra-model and inter-model level was evolved manually based on the relationship between UML models.

Existing work on transforming OCL to SQL [DH99, DHL01, LBMS05] is used as the basis for transformation. However, the rule transformation is based on the meta-model derived when capturing requirements following the design guidelines as there is no such constraint specified in the UML superstructure specification. Design guidelines involves annotating diagrams with extra details. For example,

annotating use case with corresponding activity diagram name.

In this work, a subset of diagrams and rules have been presented. The use case, activity, sequence, class and state diagrams have been considered and a set of rules both at intra-model and inter-model level have been presented. The objective was to demonstrate the ability of the approach in handling structural consistency between static and dynamic models. Additional rules can be written to make the process of consistency checking more comprehensive. Also, at present only five models have been considered on the premise that these five models are mostly used. However, additional models can be included by defining constraints making the approach adaptive to suit needs of the customer.

The approach can be integrated using current CASE tools as part of default consistency checking that tools should perform to help software designers in capturing requirements using UML. Also, the feedback provided by the method points out errors in design which aid designers in providing consistent models. Once it is ensured that the UML models are consistent, the next phase of testing can be done with confidence knowing that the models are consistent.

Chapter 4

Managing Test Scenarios

UML diagrams capture functionality of a system at various levels and can be used for activities like scenario generation, scenario prioritization and scenario selection. Dynamic diagrams are used to capture scenarios related to a use case. UML activity diagrams capture funtionality in a form understandable to all the stakeholders of a system. UML use case and activity diagrams can be used for automated generation of scenarios. Scenarios thus generated are exhaustive and given constraints of cost and time, they can be prioritized or a subset of the same selected for execution.

4.1 Introduction

Requirements of a system are captured through use case diagrams, with each use case ideally, representing a functionality of the system. Usage scenarios describe the way a system will be used by actors. UML activity diagrams capture logic related to a single use case i.e. they elaborate a use case by specifying the order of activities, called scenarios. Thus, use cases along with corresponding activity diagrams describe expected behaviour of a system. Further, in the process of development, activities are implemented through design diagrams leading to program implementation[BS05]. So, activity diagrams are considered for testing of a system to ensure all user requirements are implemented without any fault.

Stakeholders can participate in testing (say, for acceptance testing) as activity diagrams are well-understood by them. A scenario i.e. an activity path stretching from start activity to end activity can also be considered as a test scenario during the process of testing system requirements. The challenge testers face is the selection of test scenarios such that testing of requirements is maximized with higher coverage and lesser effort[RH96].

Test scenario management involves a gammut of techniques, namely, test scenario(case)[1] generation, test case prioritization, test case selection, test suite generation, test data generation, developing a test oracle as well as generating reports detailing test case execution progress. This chapter discusses techniques for generation, prioritization and selection of scenarios obtained from UML diagrams, specifically, use case and activity diagrams. Here, it is assumed that the models have been checked for consistency which has been discussed in the previous chapter. A test suite is a composition of tests i.e. it runs a collection of test cases. Scenarios obtained from UML diagrams can be used to compose a test suite.

Test scenario generation involves enumerating all possible scenarios from one or more UML models for testing. The number of scenarios generated being large, it is not feasible to test exhaustively given constraints of time, effort and cost. This work introduces techniques to reduce the number of scenarios generated, specifically, in case of concurrent activities using priority/level of activities as the basis(Section 4.2).

The order of execution of test scenarios affects the time at which objectives of testing are met. For example, if maximizing coverage is the objective, an inappropriate execution order would take a large number of tests to achieve the goal. Focus of Section 4.3 is prioritization, which involves ordering test cases in a test suite to achieve a test objective. Execution of scenarios in random does

[1]Test cases are derived from scenarios. Thus, a scenario consists of one or more test cases. Hence, test scenarios and test case are used synonymously here.

not always produce best results. Therefore, it is necessary to select a subset of all possible scenarios that best meet an objective(e.g. fault detection, coverage). Existing work looks at factors like customer inputs, nature of requirements and risk associated as the basis for prioritization. In this work, primitives of use case and activity diagrams are used to evolve a measure to aid prioritization of use cases and scenarios.

Exhaustive testing being impossible, it is necessary to find a subset of scenarios that best represents the entire set of scenarios for testing. This is especially crucial in case of testing given limited resources and time. Test selection techniques aim at selecting a representative subset of test scenarios from the total set of scenarios for testing(Section 4.4). *Levenshtein distance* is used to calculate the distance between scenarios. A second technique adapts the idea of Longest Common Substring [Mai78]. Subscenarios, their length and position in the sequence is taken into consideration to calculate distance between scenarios. A third technique introduced in this work uses clustering (Agglomerative clustering) for grouping scenarios based on a distance metric. Clusters group similar scenarios together, thereby aiding the process of selection.

Random selection of scenarios for selection is not feasible as they may select similar scenarios for testing [CLM04]. Risk, coverage, cost and efficency are used as factors to select scenarios in [BLFR02, CP03a]. Values for risk and cost are provided by the users of the system and hence require user intervention. The techniques are advantageous in terms of user inputs to ascertain scenarios most important to the customer. However, they lack in ease of use. This work looks at using distance measures calculated between scenarios as the basis for selection.

In this chapter, techniques to manage test scenarios are presented. Section 4.2 presents techniques to reduce the number of scenarios generated in case of concurrent activities. Section 4.3 presents techniques for prioritizing use cases(Section 4.3.6)

and scenarios(Section 4.3.8). Techniques for scenario selection is the focus of Section 4.4.

4.2 Generating scenarios from UML Activity Diagrams

4.2.1 Introduction

Specification-based testing, also called black-box testing, involves producing a test suite based on the specification. Using a formal language or a model for specification helps in automation of the test generation process. For large and complex systems, testing based on covering the control flow or data flow paths becomes infeasible. In this regard, an efficient set of test scenarios needs to be generated. One of the main objectives of testing is to check whether customer requirements are met. Scenarios help in generating sequence of activities that implement system objectives/requirements.

Requirements are well defined using activity diagrams and this has led to an increased interest on generating test scenarios using activity diagrams [KKBK07, CMK08]. Each path from the initial node to the final node in an activity diagram constitutes a scenario. The problem encountered following the strategy is exponential increase in scenarios especially when considering concurrent activities, represented in an activity diagram using fork-join nodes [BL01b, LL05c, KKBK07, XLL07]. It is observed that the growth in scenarios can be limited by considering domain dependency existing among concurrent activities[2] which is the focus of this section. A method is proposed to minimize scenarios generated in case of concurrent activities, to aid the scenario generation process.

[2]A part of the work stated here is published in the Proceedings of the 11th International Conference on Information Technology, 2008 and appears in IEEE Xplore Digital Library

Section 4.2.2 and 4.2.3 discusses the selection of activity diagrams to represent scenarios. Existing work in the area of scenario generation is discussed in Section 4.2.4. The approach used in this work to generate scenarios in case of concurrent activities is discussed in Section 4.2.5. Section 4.2.6 discusses coverage criteria to evaluate the technique and Section 4.2.7 summarizes the work.

4.2.2 Why Activity Diagrams ?

A use case capturing a requirement consists of scenarios i.e. each scenario can be said to represent a requirement goal of the system. Scenarios thus represent the sequence of events in a software system and defines a system's behaviour. A scenario has a defined goal, begin with a triggering event and end by either successful or unsuccessful completion of the task. In practice, generation of scenarios is mostly done manually making it labor-intensive and error-prone. Hence, there is a need to generate test scenarios to achieve test adequacy and to ensure software quality [Mat07]. Work in literature use activity, sequence/collaboration diagrams to represent scenarios[AO00, FL02, MXX06]. In this regard, automation of test scenario generation gains importance.

The survey by Dobing et al [DP08] on the use of UML diagrams show that UML diagrams(use case narratives, use case diagrams, sequence diagram and activity diagrams) find more use in case of organizations that are middle sized to large, employing an average of more than 50 employees, the size of the project and organizational usage. Another important factor affecting the same is availability of UML tools.

Work is available in literature wherein scenarios represented using sequence diagrams are used for test case generation. Frainkin et al [FL02] have developed SeDiTeC, a tool that supports automated testing based on testable sequence diagrams. Technique for testing polymorphic interactions defined in sequence di-

agrams is presented by Suravita [SS05]. Cengarle et al [CGW06] introduce two operators, 'variant' and 'repeat' to represent variability in interactions. They use 'variant' to represent optional behaviour and 'repeat' for repetition of instances. An algorithm is defined by Lund et al [LS06] for deriving tests from sequence diagram specifications that use operators 'neg' and 'assert' that represent invalid and universal behavior. Use case diagrams, corresponding sequence diagrams, class diagrams and OCL is used by Briand et al [BL02] for specification based testing. They introduce a methodology to build functional system requirements which is transformed to test cases, oracles and drivers once detailed design information is obtained.

Sequence diagrams are used to represent scenarios, especially, in the design phase. The objective is to find architectural, interface and logic problems as the sequence diagram gives details about interfaces, states, message order, assignment of responsibilities, timers and error situations. Besides, sequence diagrams are a useful tool for specifying system requirements for programmers as they clarify understanding of the application among technical members of the project team [DP08]. Thus, while being an effective tool for use by developers, the drawback of sequence diagram is in terms of understandability by different stakeholders of the system. Compared to the sequence diagram, activity diagrams find better acceptance among stakeholders in terms of customer involvement through verification and validation of requirements. The advantage lies in its simplicity and the ease of understanding the flow of logic of the system. Also, activity diagrams can be used at different levels of abstraction aiding both the customers as well as developers in capturing requirements[OMG].

Another development is the change brought in the UML 2 superstructure specification [OMG] wherein there is a clear delineation of activity diagrams from state diagrams. Also, non-availability of tools and techniques related to activity dia-

grams have been an impediment. The Object Management Group(OMG)[OMG] also classifies activity diagrams as Fundamental, Basic, Intermediate, Structured and Complete in terms of complexity in the process flow. The basic level includes control sequencing and data flow between actions. However, forks and joins as well as decisions and merges are not supported. The intermediate level supports concurrent control and data flow, and decisions. The complete level adds constructs that enhance the lower level models, such as edge weights and streaming. This work is concerned with the intermediate level of activity diagrams that include control and data flow, and decisions. Scenarios are generated from activity diagrams for the purpose of testing.

4.2.3 Activity Diagrams and Scenarios

An activity diagram represents the scenarios for each use case in a use case diagram [CPTT05]. UML activity diagrams are developed using two types of nodes, namely, action nodes and control nodes as shown in Figure 4.1. Action nodes include Activity, CallBehaviourAction, SendSignal and AcceptEvent. Control nodes include InitialNode, FinalNode, FlowFinal, Decision, Merge, Fork and Join. Also, UML 2.1 superstructure describes several levels of activity modeling: Basic, Intermediate, Complete, Structured, complete-Structured and Extra-Structured activities [OMG] as discussed in the previous section.

An activity diagram consists of activities and transitions, showing the flow of control from one activity to another. They can be used to model both sequential and concurrent activities. Also, an activity diagram can be viewed as a graph with nodes representing activities and edges labeled with transitions. Figure 4.2 shows an activity diagram for booking of a package tour by a customer. The activities inside a fork-join are concurrent activities and are executed in parallel. Other activities are sequential in nature.

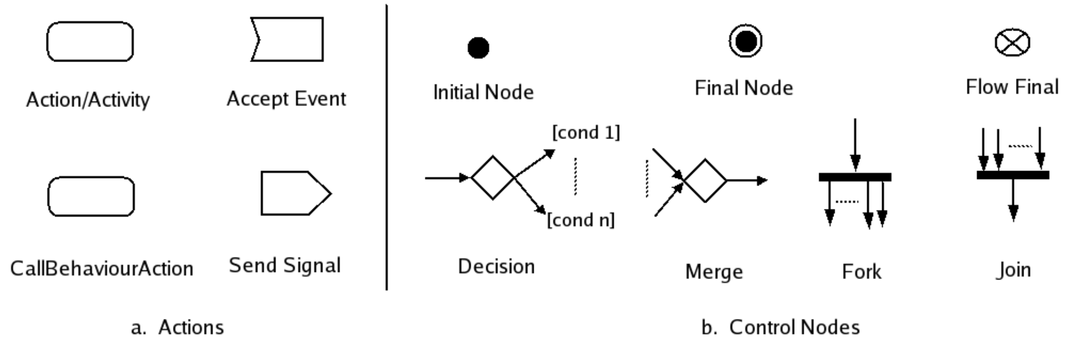


Figure 4.1: Constructs used in an activity diagram

Based on the above, the following definitions follow:

Definition 1: An activity diagram AD is a tuple,

AD = (A, T, F, J, R, a_I, a_F) where

A = a_0, a_1, a_2,...a_m is a finite set of activities,

Priority of an activity 'a' is denoted by a.p,

T = t_0, t_1, t_2,....t_n is a finite set of transitions,

F = f_1, f_2,.....f_k is a finite set of forks,

J = j_1, j_2,....j_k is a finite set of joins,

R = R $\subseteq$ (A $\times$ T)

a_I is the initial state, and a_F is the final state.

Definition 2: A scenario , s $\in$ S(S, being the set of scenarios), in an activity diagram, AD, can be defined as an execution path from the initial state to the final state consisting of activities and transitions.

i.e. $\forall$ s, where s $\in$ S,

s = $a_0 \rightarrow t_0 \rightarrow a_1 \rightarrow t_1 \rightarrow ...... \rightarrow t_n \rightarrow a_m$

where $a_i \in$ A, $t_i \in$ T, a_0 is the initial state, a_m is the final state and S is the set of scenarios.

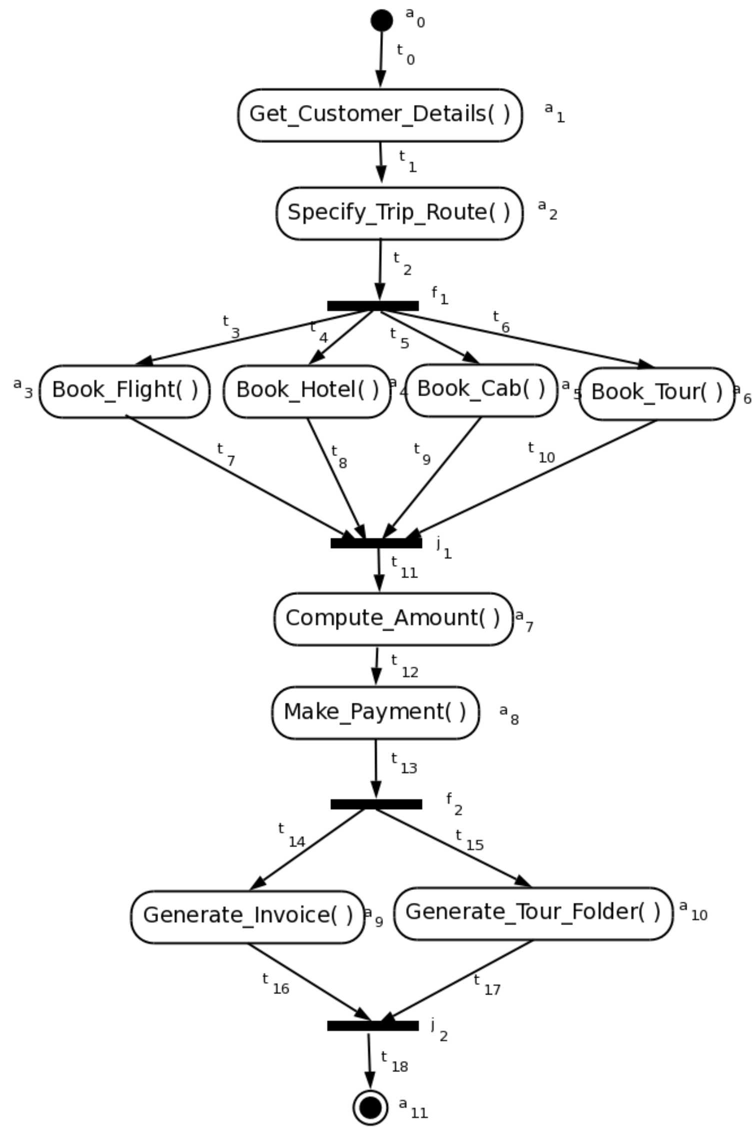


Figure 4.2: Activity diagram for "Booking a Package Tour"

4.2.4 Related Work on Scenario Generation using UML Activity Diagrams

Lionel et al [BL01b] in their work present an approach for UML-based testing using activity diagrams. They capture the sequence of usage scenarios related to the use cases. An activity diagram is transformed into a weighted graph from which the sequence of usage scenarios are identified. This helps in determining paths for testing. A modified Depth-First Traversal [DFS] algorithm is used for automated generation of test cases by Linzhang et al [LJX+04]. The objective was to elucidate basic paths for the activity diagram. An advantage of the technique is that basic paths are explored but then it does not generate all possible scenarios which is a drawback. Also, their algorithm works on the assumption that each branch/merge and fork/join has only two outgoing edges. Li et al [LL05c] use anti-ant like agents to generate test threads from UML activity diagrams. However, path explosion problem is the limitation of this work as it results in redundant exploration of the activity diagrams. Adaptive agents are used for test scenario generation by Xu et al [XLL05] to overcome the problem of path explosion. The algorithm uses one agent at the start and produces more agents when necessary. The assumption is that each cyclic loop is executed at most two times, similar to the assumption in [LJX+04]. They overcome the problem of path explosion in [LL05c] by creating agents only where necessary.

In order to derive test cases from UML activity diagrams, Kim et al [KKBK07] in their work convert the activity diagram into an Input/Output explicit Activity Diagram (IOAD). IOAD is an activity diagram that explicitly shows external inputs and outputs. The IOAD is then converted to a directed graph for extraction of test scenarios. Basic path is the coverage criteria. This work also looks at simple fork-join structures.

Xu et al [XLL07] in their work, overcome the limitation in previous work(i.e.

Table 4.1: Testing Approaches using Activity Diagrams

Reference#	Technique	Limitation	Tool
[BL02]	-ADs capture sequence of usage scenarios related to use cases	Transformation to weighted graph	TOTEM
[LJX+04]	-Modified DFS	Loops executed atmost once -Branch/Merge,Fork/Join have atmost two outgoing edges -Some scenarios not generated, like fork-join	UMLTGF
[CLL05]	-Modified DFS Poseidon 2.1 used for modelling	Scenario explosion in case of loops -Handles simple fork-join constructs -Sequentially connected execution paths between fork/join	AD2US
[LL05c]	-Anti-ant like agents -Redundant exploration of threads avoided	Scenario explosion in case of loops -Handles simple fork-join constructs	A(Unkwn)
[XLL05]	- Uses Adaptive agents -Agents produced when required -Redundant exploration of threads avoided	-Each nested fork-join considered as AD -Same as fork-join for branch-merge Overhead of merging ADs -Mixed fork joins not considered	TSGAD

AD - Activity Diagram; NA - Not available; A(Unkwn) - tool present,name unknown

Testing Approaches using Activity Diagrams

Reference#	**Technique**	**Limitation**	**Tool**
[CMX06], [CQX+07b]	-Objective to meet coverage criteria -Generates random test cases from JAVA program -Generates program execution traces -Compare traces with activity diagram to achieve coverage criteria	-Overhead of generating random test cases -Technique is exhaustive	AGTCG
[XLL07]	-Handles complex fork/join pairs	-Does not consider other constructs of AD	TSUMLAD
[KKBK07]	-Input/Output Activity Diagrams(IOAD) -Transformed to directed graph	-Specifying input/output of each activity	NA
[CMK08]	-ADs translated to NuSMV, a formal model -Properties as CTL/LTL formulas	-Additional overhead in converting ADs to NuSMV	A(Unkwn)
[XLLP08]	-Handles exceptions, expansion and interruptible activity regions		TGUML

AD - Activity Diagram; NA - Not available; A(Unkwn) - tool present,name unknown

simple fork-join) by considering complicated fork-join pairs i.e. fork-join pairs that are nested or those that have loops and branches. A prototype tool, TSUMLAD, was developed to automatically generate test scenarios from activity diagrams. All possible scenarios are generated by this technique. However, the number of scenarios thus generated is very large and it may not be possible to test all scenarios given constraints of resource and time. Table 4.1 compares and summarizes the various approaches to generating scenarios from UML activity diagrams.

Work in literature look at various ways of generating scenarios. However, one common issue faced is scenario explosion especially in case of fork-join constructs. Lionel et al [BL01b] and Linzhang et al [LJX$^+$04] handle this issue by constraining the number of outgoing edges a fork/join construct can have. Xu et al propose algorithms to generate all possible scenarios in case of complicated for/join constructs. While the former constrains the representation of requirements, the latter creates the problem of scenario explosion.

To overcome the problem of scenario explosion, there is a need to generate scenarios in an optimized way, especially in case of concurrent activities. This work concentrates on generation of scenarios in case of concurrent activities represented using fork-join constructs by taking precedence of activities into consideration. It is done in two ways: one, using priority of activities, obtained by giving an order in which activities are generated and two, by defining levels, wherein activities at one level have higher precedence over activities on another level[3]. The methods are discussed in the next section.

Table 4.2: Scenarios from activity diagram 'Book Package Tour'

$a_0 \to a_1 \to a_2 \to f_1 \to a_3 \to a_4 \to a_5 \to a_6 \to j_1 \to a_7 \to a_8 \to f_2 \to a_9 \to a_{10} \to j_2 \to a_{11}$
$a_0 \to a_1 \to a_2 \to f_1 \to a_3 \to a_4 \to a_5 \to a_6 \to j_1 \to a_7 \to a_8 \to f_2 \to a_{10} \to a_9 \to j_2 \to a_{11}$
$a_0 \to a_1 \to a_2 \to f_1 \to a_3 \to a_5 \to a_4 \to a_6 \to j_1 \to a_7 \to a_8 \to f_2 \to a_9 \to a_{10} \to j_2 \to a_{11}$
$a_0 \to a_1 \to a_2 \to f_1 \to a_3 \to a_5 \to a_4 \to a_6 \to j_1 \to a_7 \to a_8 \to f_2 \to a_{10} \to a_9 \to j_2 \to a_{11}$
$a_0 \to a_1 \to a_2 \to f_1 \to a_3 \to a_4 \to a_6 \to a_5 \to j_1 \to a_7 \to a_8 \to f_2 \to a_9 \to a_{10} \to j_2 \to a_{11}$
$a_0 \to a_1 \to a_2 \to f_1 \to a_3 \to a_4 \to a_6 \to a_5 \to j_1 \to a_7 \to a_8 \to f_2 \to a_{10} \to a_9 \to j_2 \to a_{11}$
$a_0 \to a_1 \to a_2 \to f_1 \to a_3 \to a_5 \to a_6 \to a_4 \to j_1 \to a_7 \to a_8 \to f_2 \to a_9 \to a_{10} \to j_2 \to a_{11}$
$a_0 \to a_1 \to a_2 \to f_1 \to a_3 \to a_5 \to a_6 \to a_4 \to j_1 \to a_7 \to a_8 \to f_2 \to a_{10} \to a_9 \to j_2 \to a_{11}$
$a_0 \to a_1 \to a_2 \to f_1 \to a_3 \to a_6 \to a_4 \to a_5 \to j_1 \to a_7 \to a_8 \to f_2 \to a_9 \to a_{10} \to j_2 \to a_{11}$
$a_0 \to a_1 \to a_2 \to f_1 \to a_3 \to a_6 \to a_4 \to a_5 \to j_1 \to a_7 \to a_8 \to f_2 \to a_{10} \to a_9 \to j_2 \to a_{11}$
$a_0 \to a_1 \to a_2 \to f_1 \to a_3 \to a_6 \to a_5 \to a_4 \to j_1 \to a_7 \to a_8 \to f_2 \to a_9 \to a_{10} \to j_2 \to a_{11}$
$a_0 \to a_1 \to a_2 \to f_1 \to a_3 \to a_6 \to a_5 \to a_4 \to j_1 \to a_7 \to a_8 \to f_2 \to a_{10} \to a_9 \to j_2 \to a_{11}$
....
....
....
....
....

4.2.5 Approach to Deriving Test Scenarios for Concurrent Activities inside Fork-Join

Concurrent activities in a system are represented using fork-join constructs. In case of concurrent activities, several activities can be active at the same time. They may share data in any order as the order of accessing is non-deterministic. Hence, such access of shared data may bring in unexpected results [LHS01].

Example: Consider the example of the use case, booking a package tour. The activity diagram of the use case is shown in Figure 4.2. The activities are named a_0, a_1,....., the transitions t_0, t_1, ... , the forks/joins as f_1, f_2,.... and j_1, j_2, ... for ease of use. Note that in this work the details of each activity, i.e. actions are not

[3] A part of the work stated here is reported in 'Automated Scenario Generation based on UML Activity Diagrams', 11th International Conference on Information Technology (ICIT 2008), pp. 209-214, IEEE Computer Society, 2008

Algorithm 1 ScenGen

1: { *Input :*
G : An activity graph obtained from activity diagram, AD.
Output :
Set of Scenarios
Functions called:
fork-join(G') returns all paths within a fork join construct which is a subgraph G'
Initialize :
Visited of each node = 0. Stack S to keep track of visited nodes.
SCEN stores scenarios. Adjaceny - adjacency matrix that stores transtions between nodes present in the activity diagram. }

2: Begin

3: Traverse the graph using depth-first-search(DFS) from the initial nodes to the final node. For each node visited during the traversal, increment the number of visits.

4: If type of node = fork, then , get all nodes until corresponding join, forming a sub graph, and call function fork-join(G') to generate all paths within fork-join construct. Set visited for each node. Return the set of subpaths.

5: If reached final node or number of visits of current node = 3, then store scenario. Backtrack to the node which has atleast a child node with number of visits less thatn two. Pop nodes until this point from S.

6: Perform above steps until all nodes are visited and final node is reached.

7: For each scenario in SCEN, check if the last node is a final node. If yes, print the scenario. Else, get all sub-paths from the last node to the final node and append to the scenario. Print scenarios

8: End

considered to preserve the simplicity of the activity diagram. a_1, a_2, a_7 and a_8 are non concurrent activities whereas a_3, a_4, a_5, a_6, a_9 and a_{10} are concurrent activites. For a large and complex system, concurrent activities lead to path explosion due to a large number of threads that might be in operation at the same time. The order of execution of concurrent activities cannot be determined apriori as it depends on the runtime environment. Hence, the complexity in testing concurrent activities is n! where n is the number of activities within a fork-join construct. The steps involved in generating scenarios from an activity diagram is given in Algorithm 1. Depth first traversal is used to traverse the graph. Applying the algorithm *ScenGen(G)* on the activity graph shown in Figure 4.2, the scenarios obtained are as shown in Table 4.2.

At first, customer details and trip details are obtained. Once the details are obtained, concurrent activities begin. Concurrent activities including booking the flight, hotel, cab and tour are performed. All of the four activities must be performed successfully for a trip to be booked. Also, the order in which the concurrent activities get executed is not predictable and can be in any order. Once, the four activities produce an output, the amount is computed and confirmation of the trip is taken from the customer. Payment is made towards the trip and again concurrent activities of 'generating invoice' and the 'tour folder' is performed. In this example, a simple fork-join construct is used in the activity diagram. A total of 48 scenarios can be generated considering the different ordering in calling concurrent activities. Thus, the number of scenarios generated in case of a concurrent activities is very large and impossible to test within given constraints.

One way to reduce the number of scenarios generated in case of concurrent activities is to discard illegal or irrelevant combinations of activities that may occur due to random selection. Consider the case in Figure 4.2. Suppose that the tour and cab booking activities, being in house can be arranged by the business in one

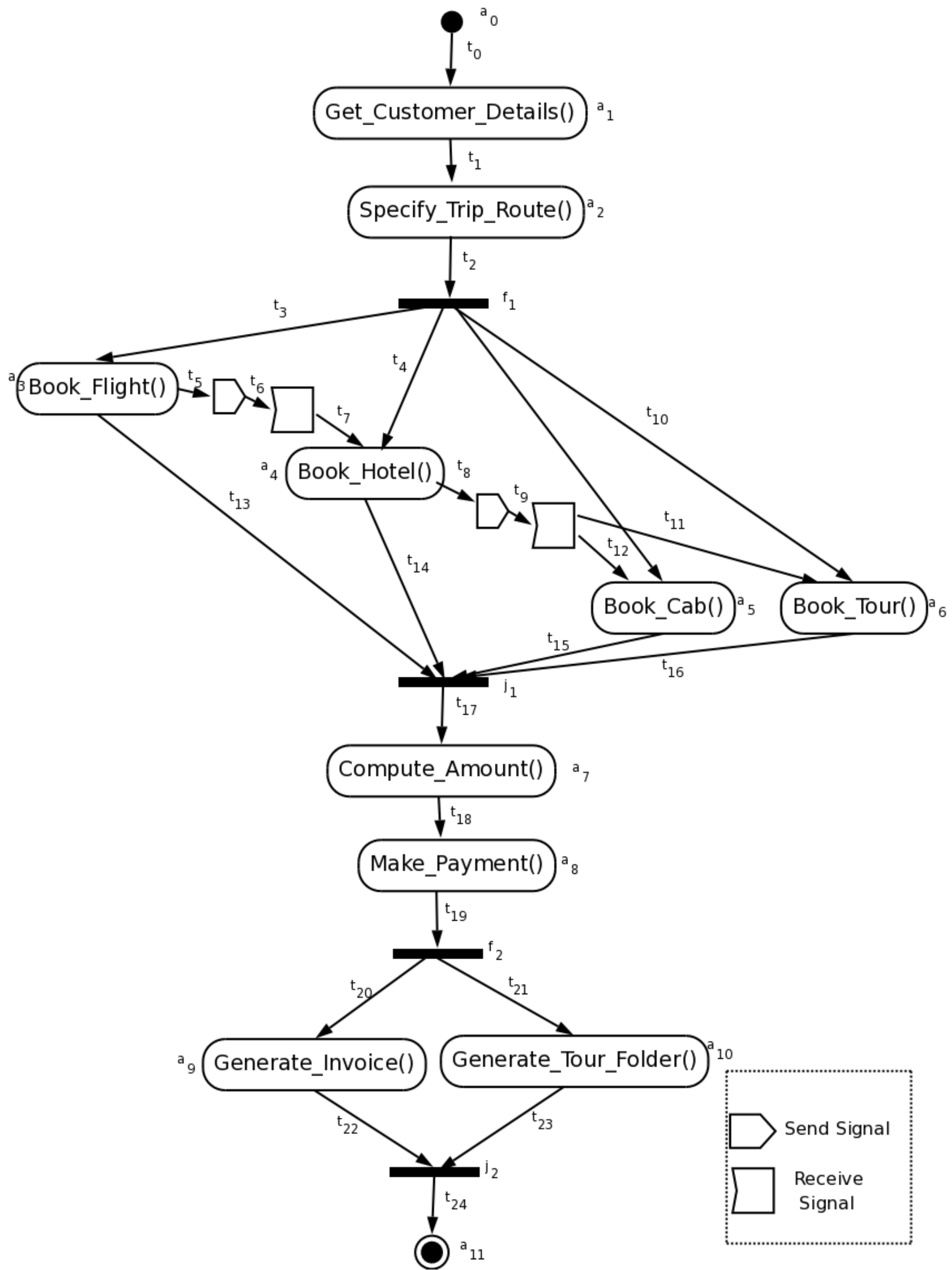


Figure 4.3: Interleaving of activities inside fork-join

of their establishments, dependent on the arrival of the customer and availability of accomodation. Then, this constraint in terms of order of execution could be used effectively in generation of scenarios. Figure 4.2 shows the interleaving of activities within the fork-join construct. The activity 'Book Hotel()' depends on confirmation from the activity 'Book Flight()'. In case the latter is not available, then the execution of the former is not applicable. Similar is the case of dependency between 'Book Hotel()' and 'Book Flight()' with activities, 'Book Cab()' and 'Book Tour()'. Thus, sharing of data among activities brings in dependency that enforces an order among concurrent activities; for dependency, only some sequences of activities are realistic and hence others may be discarded.

Figure 4.3 shows the case of 'Booking a Package Tour' with interleaving among the activities. The activity 'Book Flight()' sends a signal to 'Book Hotel()' once it is completed. This signal fires the activity 'Book Hotel()' to completion. The same is true of 'Book Cab()' and 'Book Tour(). This dependency between activities can be used to generate a subset of scenarios thereby reducing the test effort. The scenarios generated based on the above is shown in Table 4.3.

Table 4.3: Test Scenarios for 'Book Package Tour applying priority-based selection'

$a_0 \rightarrow a_1 \rightarrow a_2 \rightarrow f_1 \rightarrow a_3 \rightarrow a_4 \rightarrow a_5 \rightarrow a_6 \rightarrow j_1 \rightarrow a_7 \rightarrow a_8 \rightarrow f_2 \rightarrow a_9 \rightarrow a_{10} \rightarrow j_2 \rightarrow a_{11}$
$a_0 \rightarrow a_1 \rightarrow a_2 \rightarrow f_1 \rightarrow a_3 \rightarrow a_4 \rightarrow a_5 \rightarrow a_6 \rightarrow j_1 \rightarrow a_7 \rightarrow a_8 \rightarrow f_2 \rightarrow a_{10} \rightarrow a_0 \rightarrow j_2 \rightarrow a_{11}$
$a_0 \rightarrow a_1 \rightarrow a_2 \rightarrow f_1 \rightarrow a_3 \rightarrow a_4 \rightarrow a_6 \rightarrow a_5 \rightarrow j_1 \rightarrow a_7 \rightarrow a_8 \rightarrow f_2 \rightarrow a_9 \rightarrow a_{10} \rightarrow j_2 \rightarrow a_{11}$
$a_0 \rightarrow a_1 \rightarrow a_2 \rightarrow f_1 \rightarrow a_3 \rightarrow a_4 \rightarrow a_6 \rightarrow a_5 \rightarrow j_1 \rightarrow a_7 \rightarrow a_8 \rightarrow f_2 \rightarrow a_{10} \rightarrow a_9 \rightarrow j_2 \rightarrow a_{11}$

In this work, two criterion based on precedence are defined for the generation of scenarios in case of concurrent activities.

Criterion 1: Priority-based selection

Let priority of an activity a_i be denoted as $a_i.p$. If the priority of a_1 is greater than the priority of a_2, then $a_1 \rightarrow a_2$ is a legal scenario.

i.e. $\forall\, a_i.p_i > a_j.p_j \wedge a_i,a_j \in$ A | $a_i \rightarrow a_j \in$ S

i,j $\in$ n

An algorithm *Priority-Based-ScenGen()*(Algorithm 2), is proposed to generate all scenarios satisfying the above criterion. The function is called when a fork node, f_i is encountered. All activities originating from the fork node and till the join node are considered. A queue Q contains all concurrent activities originating from the fork, f_i, in order of priority. An adjacency matrix is used to store details on transitions between activities i.e. element AM[i][j] is set to 1 if there exists a transition from activity a_i to a_j. Priority of each activity is stored in P and L stores the generated scenario. By applying Algorithm 2, a reduced set of test scenarios is obtained. Consider that the priority of activity Book_Flight() is greater than priority of activity Book_Hotel() whose priority is greater than the priority of the activities Book_ Cab() and Book_Tour(); priority of activity Book Cab() and Book Tour() are equal. Then, the order of execution of concurrent activities is altered due to the priority they hold with respect to each other. That is, Book_Flight() having highest priority is executed first followed by activity Book_Tour() followed by Book_Cab() and Book_Tour(). Application of the algorithm to the activity diagram in Figure 4.3 gives the scenarios shown in Table 4.3. Thus, introducing priority for activities within the fork join construct helps reduce the number of scenarios generated. Once all the test scenarios are generated, test cases can be generated to form the test suite.

Criterion 2: Level-based selection

Activities inside a fork-join can be said to be at different levels of execution due to the dependency that exists between them. That is, activities at one level must be completed before moving to activities at the next level. Hence,

Algorithm 2 Priority-Based-ScenGen

```
1: { Input:
   Q : Concurrent activities ordered by priority originating from fork
   AM : Adjacency Matrix where AM[i][j]=1 if there exits a transition from ac-
   tivity a_i to a_j.
   S : Generated Scenario
   P : Priority of each activity
   Output:
   Scenarios
   }
2: while Q not empty do
3:   Get activity a from Q ;
4:   if a not in S then
5:     Add a to S;
6:   end if
7:   if all activities present in S then
8:     Print S;
9:     exit();
10:  end if
11:  for each activity a_i do
12:    if there exists a transition t_i from a to a_i such that P(a) > P(a_i) then
13:      set transition t_i to visited
14:      if a_i has no other activities with higher priority incident on it then
15:        set activity a_i to visited
16:        add a_i to S
17:      end if
18:    end if
19:  end for
20: end while
```

Let L = { l_1, l_2,...l_n } be a finite set of levels;

Let { a_0, a_1, a_2,...a_j } $\in l_1$,

{ a_{j+1}, a_{j+2},.....a_k } $\in l_2$,....,

then, the operation priority of all activities in a particular level is the same.

i.e. a_0.p = a_1.p =... = a_j.p.

Also, l_1.p > l_2.p. i.e. operation priority of activities in level l_1, is greater than operation priority of activities in level l_2.

In case of criterion 2, all activities at level l_i have to be completed before moving to activities at level l_{i+1}. Procedure *LevelBasedGen* gives the procedure for level-based selection. The level of each activity is stored in LVL. In case of nested fork-joins, the algorithm is called recursively.

Algorithm *Level-Based-ScenGen()* (Algorithm 3) gives the steps involved in level based generation of scenarios. Consider a set of activities, where certain activities need to be executed(performed) before other tasks. This defines a precedence order. The precedence constraints form a directed acyclic graph. The approach defines an order to execute activities such that each activity is executed only after the activities incident on it are completed. Application of the algorithm to the activity diagram in Figure 4.3 gives the scenarios shown in Table 4.4. Introducing levels for execution of activities based on dependency among concurrent activities help reduce the number of scenarios generated.

Table 4.4: Test Scenarios for 'Book Package Tour applying level-based selection'

Scenario
$a_0 \rightarrow a_1 \rightarrow a_2 \rightarrow f_1 \rightarrow a_3 \rightarrow a_4 \rightarrow a_5 \rightarrow a_6 \rightarrow j_1 \rightarrow a_7 \rightarrow a_8 \rightarrow f_2 \rightarrow a_9 \rightarrow a_{10} \rightarrow j_2 \rightarrow a_{11}$
$a_0 \rightarrow a_1 \rightarrow a_2 \rightarrow f_1 \rightarrow a_3 \rightarrow a_4 \rightarrow a_6 \rightarrow a_5 \rightarrow j_1 \rightarrow a_7 \rightarrow a_8 \rightarrow f_2 \rightarrow a_9 \rightarrow a_{10} \rightarrow j_2 \rightarrow a_{11}$
$a_0 \rightarrow a_1 \rightarrow a_2 \rightarrow f_1 \rightarrow a_3 \rightarrow a_4 \rightarrow a_5 \rightarrow a_6 \rightarrow j_1 \rightarrow a_7 \rightarrow a_8 \rightarrow f_2 \rightarrow a_{10} \rightarrow a_0 \rightarrow j_2 \rightarrow a_{11}$
$a_0 \rightarrow a_1 \rightarrow a_2 \rightarrow f_1 \rightarrow a_3 \rightarrow a_4 \rightarrow a_6 \rightarrow a_5 \rightarrow j_1 \rightarrow a_7 \rightarrow a_8 \rightarrow f_2 \rightarrow a_{10} \rightarrow a_9 \rightarrow j_2 \rightarrow a_{11}$

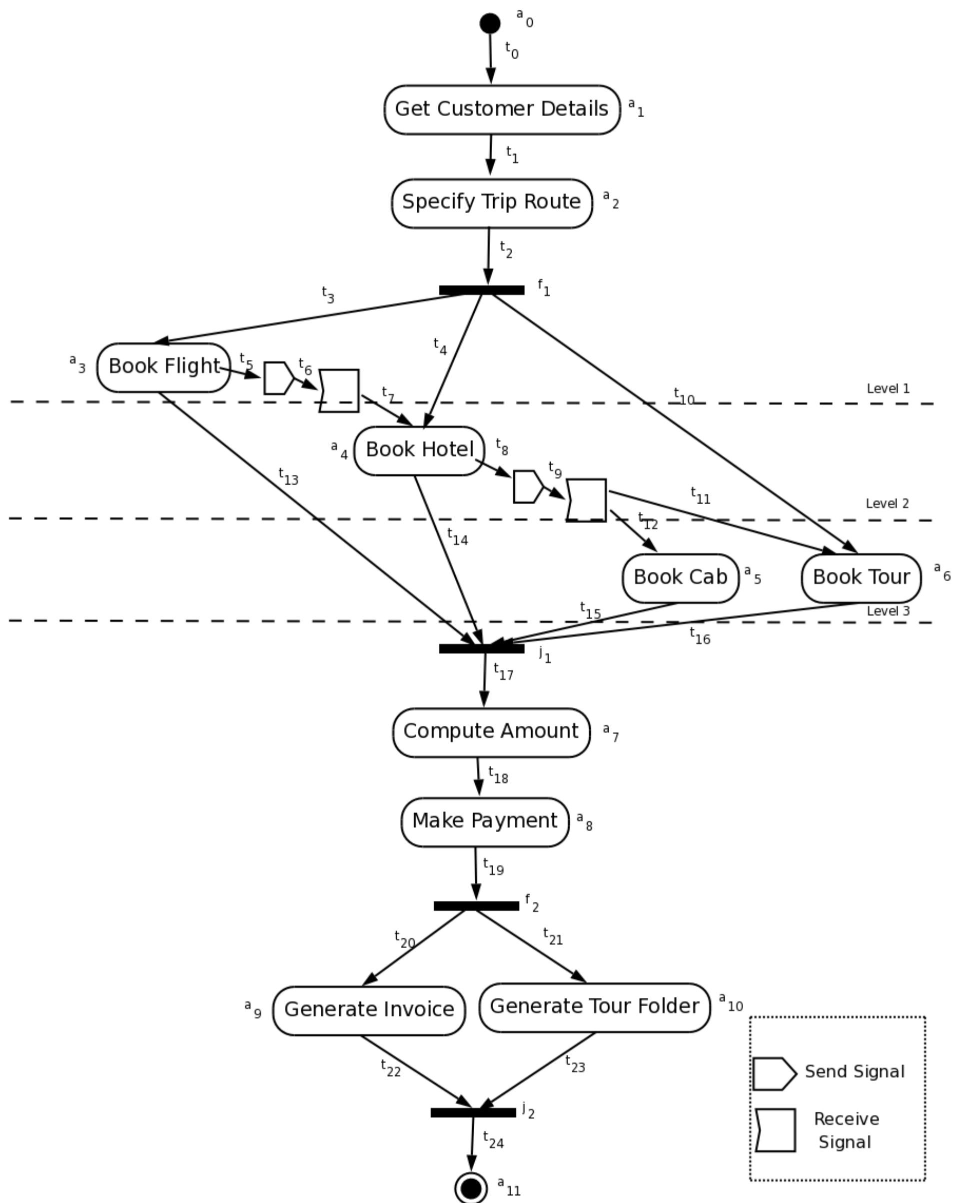


Figure 4.4: Interleaving of activities inside fork-join

Algorithm 3 Level-Based-ScenGen

```
1: {  Input:
   Q : Concurrent activities originating from fork
   AM : Adjacency Matrix where aM[i][j]=1 if there exits a transition from ac-
   tivity a_i to a_j.
   S : Generated Scenario
   Lvl : Level of each activity
   Output:
   Scenarios
   }
2: while Q not empty do
3:    Get activity a from Q ;
4:    Add a to S ;
5:    if all activities present in S then
6:       Print S;
7:       exit();
8:    end if
9:    for each activity a_i do
10:      if there exists a transition t_i from a to a_i such that LVL(a) > LVL(a_i)
         then
11:         set transition t_i to visited
12:         if a_i has no other activities with higher level incident on it then
13:            set activity a_i to visited
14:            add a_i to S
15:         end if
16:      end if
17:   end for
18: end while
```

4.2.6 Test coverage criteria

Test quality is one of the key issues of the testing process. Measurement of test quality can be done by defining the adequacy criteria for testing[Mat07]. Adequacy for activity diagrams are based on covering the elements, namely, the activity states and transitions. Test assessment is a measurement of the goodness of the test set and is carried out based on one or more criteria, such as activity coverage, transition coverage, path coverage, condition coverage and loop coverage[kaw03].

Test coverage criteria is a set of rules that guide to decide appropriate elements to be covered to make test case design adequate[KK09b]. The following coverage criterion was used to consider efficacy of the technique:

a. **Basic path coverage criterion**

A basic path is a sequence of activities(scenario) where an activity in that path occurs exactly once[LJX+04, MXX06]. A basic path considers a loop to be executed at most once. Thus, given a set of scenarios, S, obtained from an activity graph and a set of test cases TC , for each scenario $s_i \in$ S, there must be at least one test case tc $\in$ TC such that when system is executed with the test case tc, s_i is exercised. For example, $a_0 \rightarrow a_1 \rightarrow a_2 \rightarrow f_1 \rightarrow a_3 \rightarrow a_4 \rightarrow a_5 \rightarrow a_6 \rightarrow j_1 \rightarrow a_7 \rightarrow a_8 \rightarrow f_2 \rightarrow a_9 \rightarrow a_{10} \rightarrow j_2 \rightarrow a_{11}$ is a basic path.

b. **Activity coverage**

Activity coverage ensures that each activity in the activity diagram be covered atleast once. Thus, given a set of scenarios, S, obtained from an activity graph and a set of test cases TC , for each activity $a_i \in$ G, there must be at least one test case tc $\in$ TC such that when system is executed with the test case tc, a_i is exercised.

c. **Transition coverage**

Transition coverage ensures that each transition in the activity diagram be covered atleast once. Given a set of scenarios, S, obtained from an activity graph and a set

of test cases TC , for each transition $t_i \in$ G, there must be at least one test case tc $\in$ TC such that when system is executed with the test case tc, t_i is exercised.

4.2.7 Scenario Generation: Summary

The technique handles automated generation of test scenarios for concurrent activities. Concurrent activities, though conceptually concurrent are interleaved. This knowledge has been used to define dependency among the activities inside fork-join pairs. This information helps in generating those test scenarios that are possible, thereby reducing unwanted scenarios. Besides, the dependency information helps in design of specifications too. This approach has the following advantages:

a. Scenarios are generated based on dependency of activities

b. Only valid scenarios are generated thereby reducing the number of scenarios for which test cases have to be generated

c. It meets the activity, transition and path adequacy criteria

Generating scenarios using an automated approach from specification captured by activity diagrams has the advantage of being understandable by all stakeholders thereby providing a common platform. Besides, automated generation of scenarios aids the testing process as scenarios are available for testing early in the lifecycle. This has the added advantage of being a verification mechanism to check for ambiguity and mismatch between requirements too. The scenarios thus generated maybe used for testing directly. However, being ordered randomly, the test suite may not be effective in detecting defects or in achieveing maximum coverage early. Prioritizing scenarios towards an objective like fault detection and coverage makes testing effective, techniques for which are discussed in Section 4.3. Also, exhaustive testing being impossible, there is need to select a subset of test scenarios that best

represents the entire set of scenarios. Techniques for test scenario selection is discussed in Section 4.4.

4.3 Prioritizing Scenarios

4.3.1 Introduction

Testing is a continuous process done across the software development life cycle with the objective of maximizing test performance like early defect detection or maximizing coverage. As software systems evolve, the size of the test suite grows making exhaustive testing infeasible. Constraints of cost, time and effort require that testing is done in an optimized way. To this end, prioritization helps in ordering test cases according to their importance.

Different indices have been used to prioritize scenarios and generally a single index is used for prioritization. Examples of prioritization indices are[TAS06]:

- priority or criticality of requirements
- rate of fault detection
- level of coverage achieved by test case e.g. number of statements or branches
- complexity metrics e.g. cyclomatic complexity, computed for procedures executed by each test case
- historical information e.g. fault proneness

Customers are involved in a project to minimize the risk of misinterpreted and missing requirements[Sri04, SWO05, SW05]. Also, customers knowledge is used in the process of requirements prioritization[TAS06]. From the customer point of view, it is important to exercise as thoroughly as possible those functionalities that carry higher value to the users. From the developer point of view, it is important to exercise as early as possible those functionalities based on the complexity as well as inter dependencies between them. So, prioritization of test scenarios

involves both customers and developers viewpoint about the system under development. Customer prioritization is obtained directly by requiring customers to order requirements(functionality) and scenarios by importance. Developer's viewpoint on priority is based on the knowledge of technical complexity involved both in design and coding. This knowledge is possessed by managers involved in the development process and is used for prioritization. Both these methods rely on knowledge possessed by the stakeholders[PST08].

While customer and developer inputs on priority is an influential factor in the success of a project, it is tedious especially in case of large software systems where the number of requirements and scenarios is large. Hence, there is need to aid the process of prioritization by providing automated techniques. Various work in literature use different techniques for prioritization based on factors like requirement, risk, cost and history. Metrics are used by Moisiadis[Moi00] as a measure to prioritize scenarios belonging to a use case. Actors used in each scenario, objects used in a particular scenario, level of extensions needed to exhaust the alternatives for each scenario and the number of abstract use cases used in each scenario are used as indicators to determine the importance of each scenario belonging to a use case. The advantage of the technique is that it provides an automated measure for prioritization. However, the metrics considered is limited to the use case diagram. Also, the automated technique by itself may not be effective as a sole measure to base prioritization. In this direction, this work introduces techniques that use primitives of both use case and activity diagrams as a measure of complexity to aid prioritization in addition to customer based prioritization. Metrics captured based on primitives is used in automating the process of prioritization. Also, a weighted average of customer priority and priority obtained from technical aspects(primitives of UML diagrams) provides an effective ordering of scenarios for testing.

Section 4.3.2 discusses the need for prioritization. Related work in the area is discussed in Section 4.3.3. The approach followed in this work for prioritizing use cases and scenarios is discussed in Sectin 4.3.4. Customer based prioritizatio is discussed in Section 4.3.5. Prioritization of use cases is discussed in Section 4.3.6 and prioritization of scenarios is discussed in Section 4.3.8.

4.3.2 Need for prioritization

The reasons for prioritization are:

- Prioritizing requirements and scenarios helps in understanding requirements that are important to the customer.
- In case of projects that are constrained by budget, prioritization helps decide requirements that should be considered for implementation and those that should be dropped.
- With projects that are constrained by time, prioritization helps arrange requirements in decreasing order or importance. i.e. test cases with higher priority, according to some criterion are executed earlier than those with lower priority.
- Prioritization helps build projects in iteration by assigning requirements to each iteration. This gives the customer a working product at every delivery.
- Customer conviction and satisfaction in the product increases with important requirements delivered early on.
- Prioritizing requirements help define costs and benefits for both customer and developer as well as define resource allocation.
- Prioritization helps developers concentrate time and effort on requirements of high priority.

4.3.3 Related Work on Prioritization

Test scenarios are developed from: one, the requirements independently, two, building models for testing purposes, and three, using the same analysis and design model used for development. In case of the first method, disadvantage is the effort and time required in developing the scenarios. Though requirements specification and test scenario generation are done in parallel, requirements may be interpreted differently causing mismatch. The advantage of the technique is that missing requirements may be identified during the process of developing test scenarios. In case of the second method, separate models (different from design models) are used to represent the system with focus being on testing. Modelling language used maybe the same as ones capturing requirements, like UML or different like state charts or Labelled Transition Systems(LTS). Advantage of the method is that inconsistencies and misinterpretations of requirements may be detected better due to different people building models for development and testing. The disadvantage in this case is the effort involved in developing two different models. The third approach involves using the same model to capture requirements and for testing. UML used for capturing requirements of the system are used for testing too. The use of the same model for design and testing helps in reducing effort involved in design as well as in removing inconsistencies in design. Disadvantage lies in the fact that missing and misinterpreted requirements may creep into the design and remain unidentified as the same model is used for building test cases. Automated tools for generating scenarios from UML diagrams are available(e.g. Rational, Visual Paradigm) but they do not aid in prioritizing scenarios. Addition of techniques for prioritization makes the process of testing effective.

Moisiadis [Moi00] use relation between use case to prioritize scenarios. Based on UML 1.1, three relations namely communicate, extend and uses are employed to

compute priority based on object usage and actor usage. Kundu and Samanta [KS07] employ use case scenarios for prioritization. Use case scenarios are represented as system sequence diagrams and converted into a graph. Weights are assigned to edges of the graph based on the outgoing edges. The technique is automated except for the additional overhead involved in converting use case scenario into a system sequence diagram. Stallbaum et al [SMP08] introduce risk based prioritization using UML activity diagrams. Risk is calculated by using two factors, damage and probability of facult obtained by augmenting risk information for each activity during risk assessment. Sum of risk values of activities is the risk of the test scenario. Though risk information is augmented once during risk assessment, there is need to do it for all activities across requirements which is an overhead especially in case of large systems. In this work, techniques for prioritizing requirements and scenarios based on primitives of UML diagram(here, use case and activity diagrams) are used to aid in prioritization. Prioritization values thus derived do not require manual intervention can be used along with customer inputs to aid prioritization process.

4.3.4 The Approach

As mentioned previously, requirements are captured using use case diagrams. Each use case represents a functionality of the system. Activity diagrams elaborate on the functionality represented in the use case diagram. For this, a guideline was introduced: Each use case in a use case diagram is elaborated using an activity diagram. Test scenarios are generated for each use case(functionality) using the corresponding activity diagram. The use case diagrams and corresponding scenarios form a hierarchy i.e., a system has one or more actors, where each actor is associated to atleast one use case. A use case is related to another by means of the <include>, <extend> and <generalization> relationship. Also, each use case

being elaborated by an activity diagram has a set of scenarios related to it.

In this section, the proposed approach to prioritize scenarios for testing is discussed. The approach consists of the following steps, with the order of execution shown in Figure 4.5.

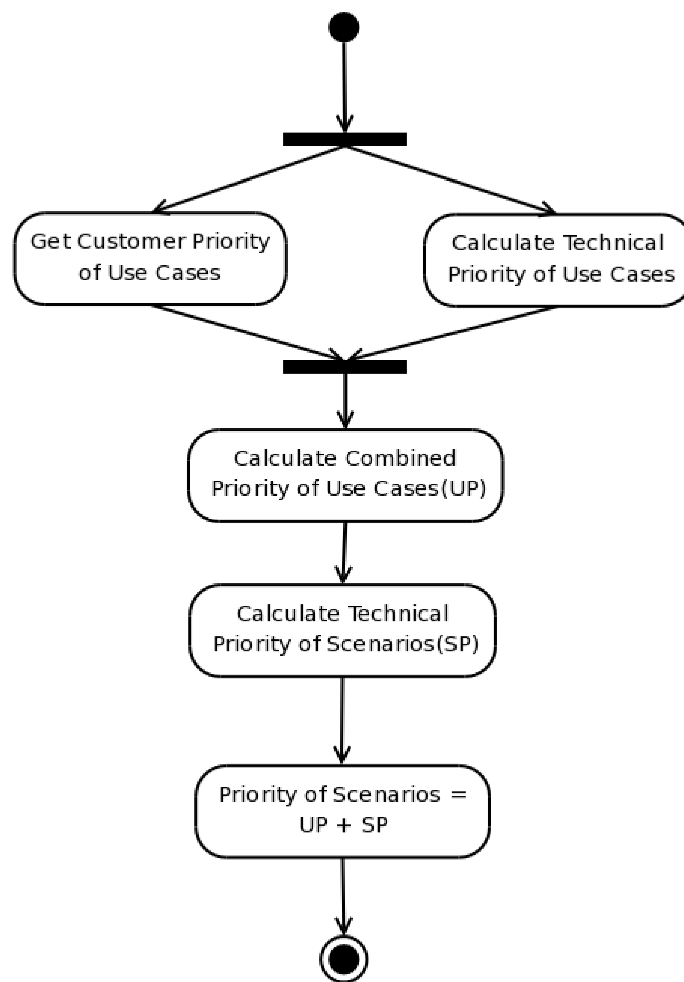


Figure 4.5: Steps in prioritization

a. Obtain customer prioritization of use cases

b. Compute use case priority from use case diagram primitives

c. Calculate combined priority of use case

d. Compute scenario priority from activity diagram primitives

e. Calculate combined priority of scenarios

Each of the activities in the activity diagram are discussed in the following subsections in detail.

4.3.5 Customer Prioritization

Customers are one of the most important stakeholders of a software project. One of the main goals of software development is to deliver a product that is of quality and within set time schedules and cost. In this direction, there is need to understand which requirements are important to the customer and their relative order. This section discusses the need to obtain customer inputs for prioritizing requirements to aid the testing process.

The Standish Group and the British Computer Society through their study concluded that, only one in eight projects can be considered truly successful. 52.7% of completed projects cost over their original estimates whereas 1% of IT projects get cancelled before completion. Table 4.5 based on the Standish Group study[Gro] also shows that customer involvement or lack thereof contributes in a big way to the success or failure of a project.

Customers assign a value to features based on their knowledge about requirements, relative importance and the need i.e. customers prioritize requirements based upon the value that a set of features will bring to the business. Moisiadis[Moi00] in their work state that approximately 45% of the software functions are never used, 19% are rarely used, and only 36% are sometimes or always

Table 4.5: Summary: Factors with rankings for successful, challenged and impaired projects[Gro]

Rank	Factors for Successful Projects	Factors for Challenged Projects	Factors for Impaired Projects
1	User involvement	Lack of user input	Incomplete requirements
2	Executive management support	Incomplete requirements	Lack of user involvement
3	Clear statement of requirements	Changing requirements & specifications	Lack of resources
4	Proper Planning	Lack of executive support	Unrealistic expectations
5	Realistic expectations	Technology incompetence	Lack of executive support
6	Smaller project milestones	Lack of resources	Changing requirements specifications
7	Competent staff	Unrealistic expectations	Lack of planning
8	Ownership	Unclear objectives	Didn't need it any longer
9	Clear vision & objectives	Unrealistic time frames	Lack of IT management
10	Hard-working focused team	New technology	Technology illiteracy

used. Hence, it can be inferred that only a subset of functionalities are used mostly by the customer. Therefore, there is need to identify the subset of requirements most important to the customer. Besides, customers view of software to be built differs from developers view. To overcome this disparity, there is need to involve the stakeholders in ordering requirements.

The objective of involving the customer in prioritizing requirements are:

- Provides better understanding of requirements and clears ambiguity e.g. incomplete requirements.
- Understand customer viewpoint of requirements priorty.
- Customers gain insight on the cost and technical difficulty associated with specific requirements.
- Understanding effects of change.
- Understand interrelationships among different requirements and their alignment with business requirements.
- Increased communication among stakeholders.
- Gain agreement of stakeholders.
- Increase business value to customer by identifying and testing completely the set of requirements of highest value to customer.
- Development being iterative, customer is made aware of the progress of the product.
- Implementing those requirements that are most valuable to the customer, first.

Customer-assigned priority (CP) is a measure of the importance of a requirement to the customer. Weigers[Wie99] defines two requirements prioritization scales, both subjective in nature. Requirements are grouped into three priority categories (three-level scales). In the first, three measures (high, medium and low) are used to rate requirements according to their relative importance. Another way is to prioritize requirements according to granularity. In medium to large projects, the number of functional requirements is large. Hence, there is need to choose an appropriate level of abstraction for prioritization (use case level, feature level, or details functional requirement level). Table 4.6 shows the two three-level scales.

Table 4.6: Scale for requirements prioritization

Names	**Meaning**
High	critical requirement, needed for the next phase
Medium	required, but can be part of a later release if necessary
Low	functional or quality enhancement; added if resources permit
Essential	product not acceptable unless requirements are satisfied
Conditional	enhances product, but product not unacceptable if requirement absent
Optional	functions that may or may not be worthwhile

Analytic Hierarchy Process(AHP) is a technique used to obtain customer priority for requirements and scenarios[YHTS09]. It helps determine the relative merit of a requirement/scenario with reference to the set of requirements/scenarios. Asking the customer alone to prioritize requirements for development is not feasible as they prioritize requirements from their own perspective i.e. how important a requirement is to the business. Cost, technical risk, resources, interdependencies and other trade-offs associated with requirements may need that the ordering of requirements for development is different. Also, for software development, there maybe a need to follow an order i.e. a requirement may need to be built before another, due to its impact on the product architecture. Hence, there is need to involve the user and the development team in the process of prioritization.

In this work, a five point scale to prioritize requirements(very low, low, average, high, very high) is used. Customer rates a requirement on a scale of 1 to 10, 1 being a requirement which has very low priority and 10, a requirement with the highest priority. The priority assigned to requirements(use cases) by the customer is used along with priority calculated based on structural aspects of a use case diagram, which is the scope of the next section.

4.3.6 Prioritizing use cases

Use case diagrams capture the requirements of a software system. Besides representing requirements(as use cases) and the actors related to a requirement, they also capture the dependencies between requirements. In this section, data captured from the primitives of the use case diagrams are used to aid in prioritization [4]. Interactions among the constructs in the diagrams are used to guide prioritization.

4.3.6.1 Constructs and relations in use case diagrams

Constructs used in a use case diagram include: actor, use case, and the relationships between them, namely, association, include, extend and generalization as shown in Figure 4.6. An actor is a user of a system who communicates directly with the system but is not part of the system[BR07]. A use case is a functionality that a system can provide by interacting with actors. A use case may involve one or more actors. Also, a use case represents all relevant behaviour of the functionality, including normal mainline behaviour, variations on normal behaviour, exception conditions, error conditions and cancellation of a request[BR07].

The relationship between constructs are:

[4] A part of the work state here is published in the Proceedings of the Third UKSim European Symposium on Computer Modeling and Simulation, 2009, IEEE Xplore Digital Society

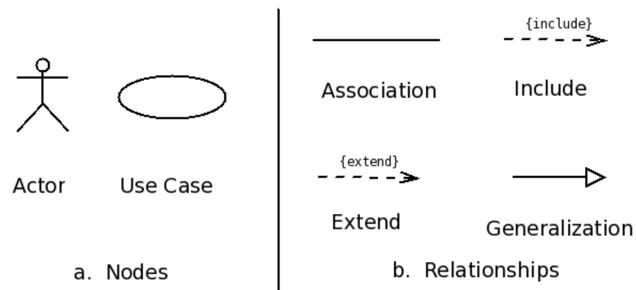


Figure 4.6: Constructs of the Use Case Diagram

- Association: The association relationship is used between an actor and a use case. Every use case should have atleast one actor, and every actor should participate in atleast one use case. Also, a use case may be used by many actors and an actor can participate in many use cases.

- Includes: An include relationship is used between two use cases. It incorporates once use case within the behaviour of another use case. The included use case may or may not be usable on its own.

- Extends: An extend relationship is used between two use cases. The extended use cases adds itself to the base use case. Also, the extended use case cannot appear alone i.e. it is not usable on its own.

- Inherits: The generalization relationship is used between actors(actor - actor) and use cases(use case - use case).

An example of a use case diagram is shown in Figure 4.7.

4.3.6.2 Preprocessing

Functionality of a system can be captured using one or more use case diagrams. Use case diagrams built using a CASE tool are stored as .xmi files. Required elements from the use case diagrams of the System Under Test(SUT) is captured using an EXtensible Stylesheet Language (XSLT) processor to transform XMI

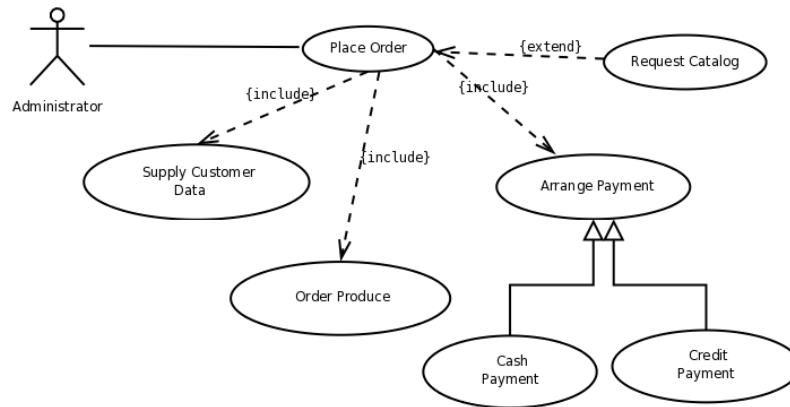


Figure 4.7: Diagram showing various kinds of use case relationships

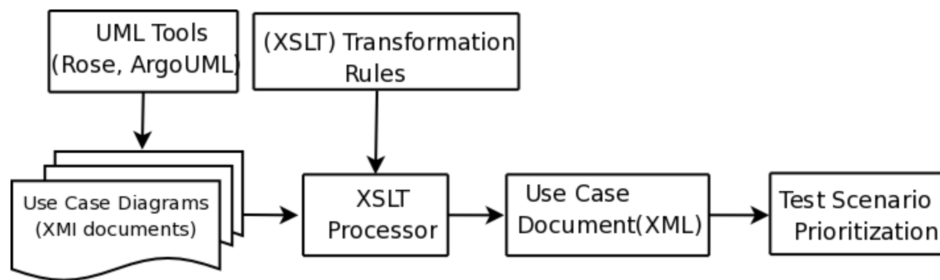


Figure 4.8: Steps in Preprocessing

documents. Figure 4.8 shows the process involved. The data thus captured, is stored in an .xml file, a sample schema of which is given in Listing 4.1.

4.3.6.3 Computing priority from use case diagram (SP - Structural Priority)

The objective of computing priority using the constructs of the use case diagram is to concentrate effort on those functionalities that are most likely to be error prone due to structural complexity. The factors taken into consideration for prioritization of use cases include[5] :

- Number of actors use case interacts with (N_a)
- Number of times use case appears in model (N_t)

[5]Metrics adapted from The Software Design Metrics tool for the UML. Available at http://www.sdmetrics.com

Listing 4.1: Schema of use case document

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="Root">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Actor" type="xs:string" maxOccures="
          unbounded">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="Usecase" type="xs:string"  maxOccurs="
        unbounded">
     </xs:element>
                <xs:sequence minOccures="1">
    <xs:element name="Relation" type="xs:string">
     </xs:element>
    <xs:element name="Usecase" type="xs:string">
     </xs:element>
     </xs:sequence>
          </xs:sequence>
    </xs:complexType>
      </xs:element>
     </xs:complexType>
 </xs:element>
</xs:schema>
```

- Number of use cases this use case includes (N_{in})
- Number of use cases that includes this use case (N_{ind})
- Number of use cases this use case extends (N_{ex})
- Number of use cases that extends this use case (N_{exd})
- Number of use cases inherited by this use case (N_{ind})

The relation between actor and use case are of highest value as they are the points where a user interacts with the system. Hence, they get highest weight. The number of times a use case appears in a model is indicative of the importance of the functionality in the system. Hence, the factor related to appearance of a use case in the model gets next highest priority. Dependency relations are the most common relations. An include relationship denotes the inclusion of behaviour by

another use case. It also indicates potential reuse. An extend dependency indicates a relation where an extending use case continues the behaviour of the base use case. The extending use case accomplishes this by inserting additional sequences into the base use case. This is followed by the inheritance relationship also indicating potential reuse. Hence, $N_a > N_t > N_{in} > N_{ex} > N_{ind} > N_{exd} > N_{ind}$. A weighted sum of the above factors is taken to calculate the priority of each use case. That is, $w_1(N_a) > w_2(N_t) > w_3(N_{in}) > w_4(N_{ex}) > w_5(N_{ind}) > w_6(N_{exd}) > w_7(N_{ind})$. $\Sigma w_i = 1$, i = 1..7. The use case with highest value gets highest priority.

4.3.7 Combined prioritization of use cases

Prioritization of use cases involves taking into consideration customer requirement as well as structural complexity. The block diagram (Figure 4.9) shows the steps involved in use case prioritization. Priority of a use case UC_i is computed as a weighted sum of customer priority and structural priority.

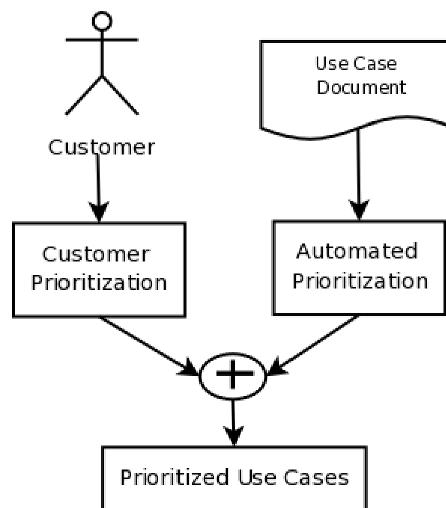


Figure 4.9: Calculating priority of use cases

Priority(UC_i) = $w_1(CP_i) + w_2(SP_i)$

where UC_i is the use case i,

CP is Customer Priority of UC_i,

SP is Structural Priority of UC_i,

w_1 and w_2 are weights | $w_1 + w_2 = 1$.

Prioritization of use cases orders requirements according to their relative priority. The next step involves prioritizing scenarios belonging to each use case.

4.3.8 Prioritization of scenarios

The capacity of any stakeholder to prioritize scenarios is limited to test suites of small size. It is impossible to prioritize scenarios consistently in case of large test suites(i.e. test suites consisting of a large number of test scenarios). To address this problem, there is need to introduce automated techniques for prioritizing scenarios.

In this work, a technique for prioritizing scenarios based on the constructs of an activity diagram is introduced[6]. Each use case is elaborated using one or more activity diagrams. An activity diagram is converted into a tree and weights are given for the nodes and edges of the tree based on criteria discussed in Section 4.3.8.3. The weights are then used to prioritize scenarios. The weight associated to each type of primitive used in defining a scenario are considered to compute the weight of a path. The paths in the activity diagram are prioritized based on path-weights.

4.3.8.1 The Approach

Each use case can be represented using one or more activity diagrams. Consider the example given in Figure 4.10. Activity diagrams represent the scenarios related to a use case. A scenario is a complete "path" in an the activity diagram. Execution of a set of paths in an activity diagram achieves a system function-

[6]A part of the work stated here is published in the Proceedings of the 1st International Conference on Computational Intelligence, Communication Systems and Networks (CICSyN 2009) and appears in IEEE Computer

ality. The main scenario (basic path) is the one beginning from the start node, traversing through all the intermediate nodes without any error, upto the end node. Alternate scenarios (alternate paths) are the cases when there is wrong entry of input or a condition is not satisfied that may happen during execution. An exception scenario is a path where an exception arises signalling an obstacle to executing the scenario. Based on the same, in this work, main, alternate and exception scenarios are defined as follows:

- ***Main Scenario :*** A main scenario is the path towards achieving a goal without any deviation.
- ***Alternate Scenario :*** An alternate scenario is one where one or more actions different from the main scenario is taken to achieve the goal. For example, if a node in a path has been visited more than once, for example, due to a wrong entry, then that path is an alternate scenario for the functionality represented by the use case. The post condition of the alternate scenario is the same as the main scenario.
- ***Exception Scenario :*** An exception scenario is one where there is an unexpected event. The post-condition is different from the post condition of the main scenario.

The objective of prioritizing scenarios extracted from the activity diagram is to identify the relative importance of scenarios. If done early in the development life cycle, it is possible to concentrate effort on designing, coding and testing the software thereby increasing quality of the software developed. Also, it ensures that customer requirements are met. The steps are:

1. Extraction of scenarios from an activity diagram.
2. Prioritizing scenarios
   a. Assign weights to nodes in paths

b. Assign weights to edges in paths

c. Calculate weight of path(scenario)

d. Normalize priority values

d. Categorize and Prioritize scenarios

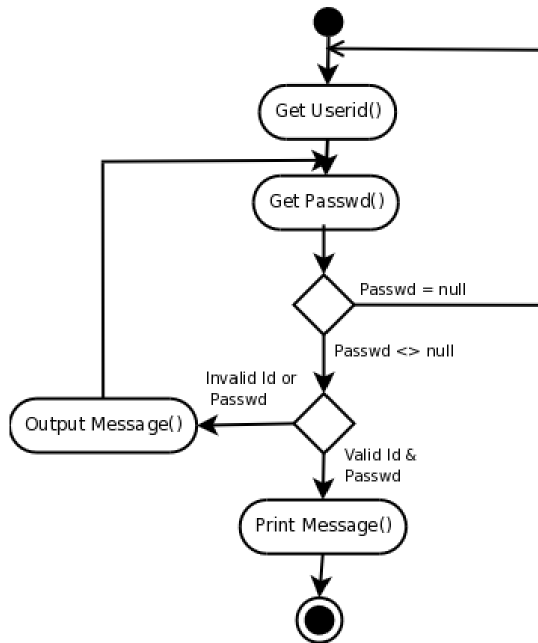


Figure 4.10: Activity Diagram for use case 'Validate User'

4.3.8.2 Converting Activity Diagram into Tree Structure

The first step involves extraction of paths. Path extraction is possible by graph traversal that returns a traversal tree spanning over the graph defined in an activity diagram. An activity diagram is converted into a tree using the following steps:

1. **Activity Diagram - Graph.** Each activity corresponds to a node $n_i \in$ N of tree, T. The label of n_i is the name of the activity. The type of the activity, namely, sequence, fork, join, branch and merge is also stored for each activity. To explain the case, activity diagram related to validating a user(refer Figure 4.10) is taken as an example. The equivalent graph for the activity diagram is shown in Figure 4.11.

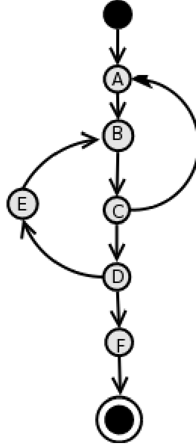


Figure 4.11: Graph representation of Activity Diagram

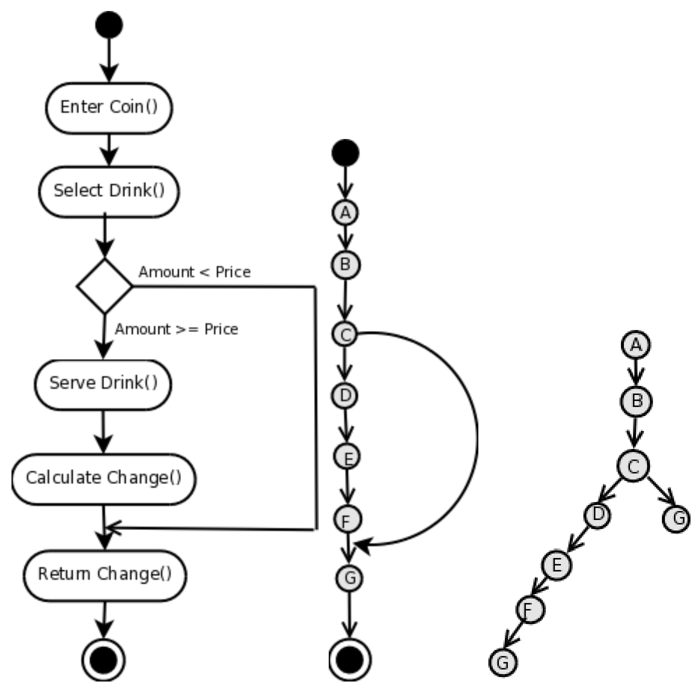
2. **Traversal Tree.** Depth First Traversal is performed on the graph G to explore paths in G. Three cases are considered in traversing an activity diagram, namely, branches, loops and concurrent activities.

Branch. A path starting from the start activity to the final activity constitutes a scenario. The methodology to traverse an activity diagram in case of a branch node is shown in Figure 4.12. For ease of use, the nodes in the graph and tree, have been named alphabetically.

Loop. When traversing an activity diagram, a restriction is imposed that loops be executed atmost twice to avoid the problem of path explosion. A modified depth first search (DFS) algorithm is used for this. Hence, an activity may be visited for a maximum of two times [XLL07, MXX06, KKBK07].

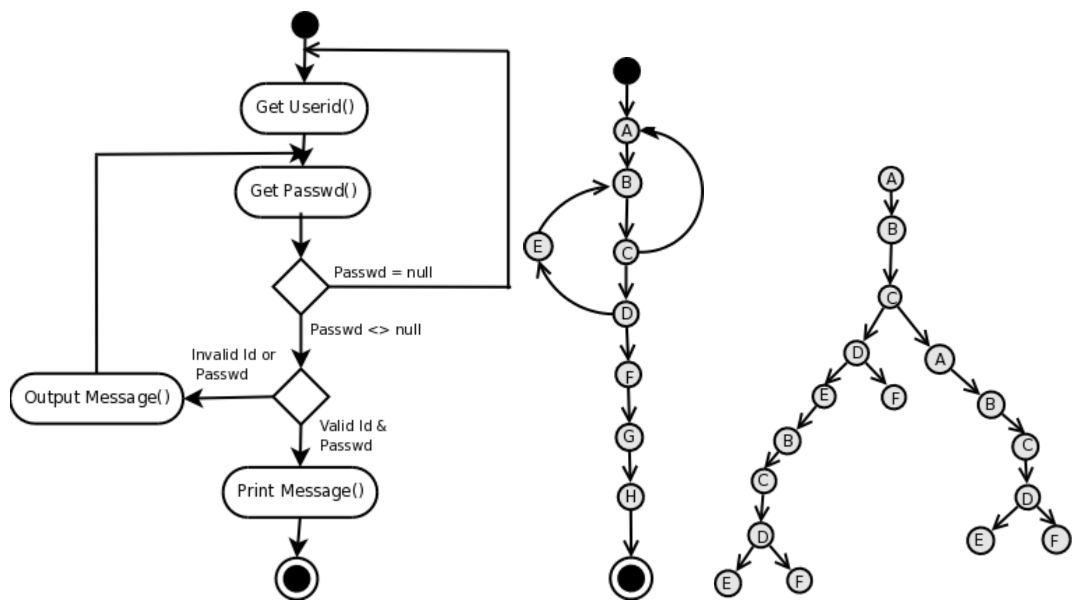
The above constraint is used to generate the tree shown in Figure 4.13(c) from the activity diagram of Figure4.13(a). For ease of use, the nodes in the tree, T have been named alphabetically. It can be seen from the tree that activities b, c, d and e occur for a maximum of two times as they form part of a loop.

Concurrent Activities. Concurrent activities are those that are executed simultaneously(i.e. in parallel). The two horizontal splits indicate the start and



(a) Activity diagram to 'Buy a Drink' (b) Graph representation (c) Corresponding tree

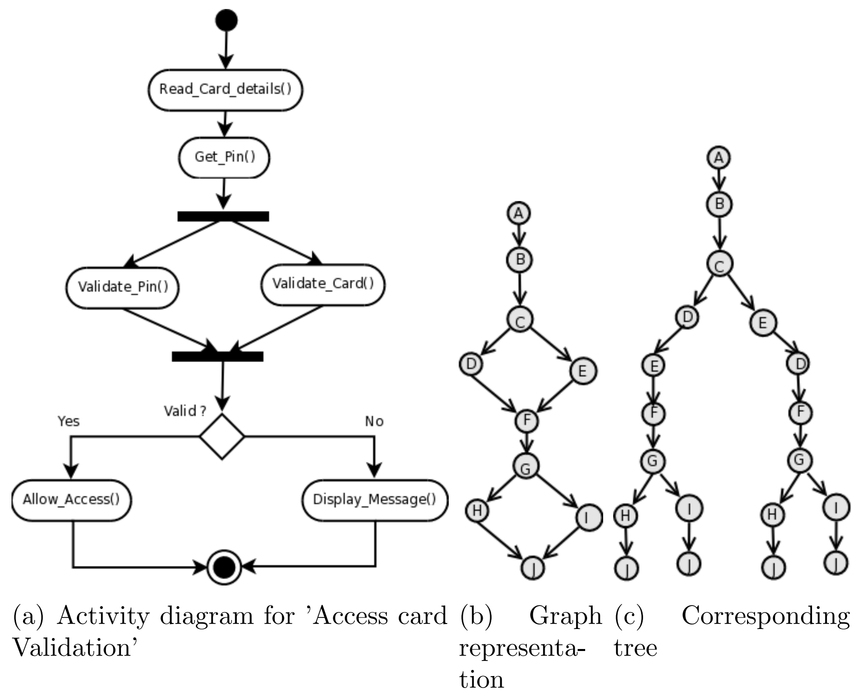
Figure 4.12: Conversion to Tree - Branch



(a) Activity diagram to 'Validate User' (b) Graph representation (c) Corresponding tree

Figure 4.13: Conversion to Tree - Loop

end of concurrent activities. In Figure 4.14(a), *Validate_Pin()* and *Validate_Card()* are concurrent activities which can execute in any order. Both the activities must be completed at the join node for further execution. The above constraint is used to generate the tree shown in Figure 4.14.



(a) Activity diagram for 'Access card Validation' (b) Graph representation (c) Corresponding tree

Figure 4.14: Conversion to Tree - Concurrent Activities

4.3.8.3 Prioritizing scenarios

The tree thus obtained from an activity diagram can be used for prioritizing scenarios. It may be noted that each path from the start node to the end node (root node to leaf node in tree T), corresponds to a scenario of the use case. The steps given below are used to prioritize the scenarios.

1. Assign weights to nodes in T

The different constructs used in an activity diagram are action/activity, branch, merge, fork and join. Weights are assigned to each of these nodes based on the complexity and possibility of occurence of defects. The fork-join construct rep-

resents concurrent activities. The branch-merge nodes represent decisions. The complexity values are given keeping in mind the risk factor involved with reference to the primitives. The fork-join nodes handle concurrency and are hence given highest weight. The branch-merge nodes need to be tested to check the boolean expression for all possible branches to be followed. Hence, second highest weightage is assigned to branch-merge nodes. Action/activity nodes are given a weightage of 1.

$$Wt(n) = \begin{cases} 3 & \text{for fork-join nodes} \\ 2 & \text{for branch-merge nodes} \\ 1 & \text{for action/activity nodes} \\ .5 & \text{for activities inside fork-join} \\ 0 & \text{for start and stop nodes} \end{cases}$$

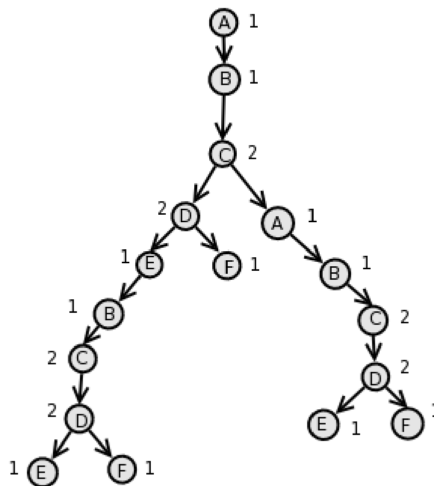


Figure 4.15: Tree T after assigning node weights

The result of traversing the tree using depth-first algorithm and assigning weights to the nodes in 4.12(a) based on the above equation is shown in Figure 4.15.

2. Assign weights to edges in T

$$Wt(e) = (n_i)_{in} \text{ X } (n_j)_{out}$$

where $(n_i)_{in}$ is the number of incoming dependencies of node n_i and $(n_j)_{out}$ is the number of outgoing dependencies of node n_j and e is the edge connecting n_i and n_j. The tree, T, after assigning weights to the edges based on the above equation is shown in Figure 4.16. Again, depth first traversal was used to assign weights to edges in tree, T.

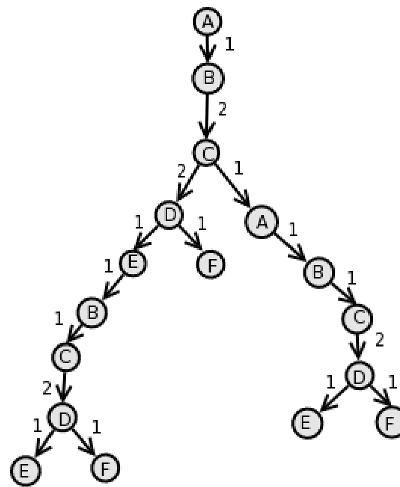


Figure 4.16: Tree T after assigning edge weights

3. Calculate weight of path

The equation given below is used to calculate the weight of the paths. The sum of the weights of both the nodes and edges is the weight of the path.

$$\text{Wt(path)} = \Sigma_{i=1}^{n} \text{ Wt}(N_i) + \Sigma_{j=1}^{m} \text{ Wt}(E_j)$$

Table 4.7 shows the weights of the five paths. The weight of the path is then used for prioritization of scenarios.

Table 4.7: Calculated weights for the five paths

Path	**Weight**
A→B→C→D→E→B→C→D→E	24
A→B→C→D→E→B→C→D→F	24
A→B→C→D→F	13
A→B→C→A→B→C→D→F	20
A→B→C→A→B→C→D→E	20

4. Normalize priority values

As mentioned before, a scale to prioritize scenarios(very low, low, average, high, very high) is used. To convert priority values to a scale of 1 to 10, normalisation is done on priority values obtained from a set of scenarios belonging to an activity diagram.

5. Categorize scenarios

Scenarios are categorized by the actions[JCSdPLK00]. Scenarios are of three types: main scenarios, alternate scenarios and exception scenarios as discussed previously in Section 4.3.8.1.

4.3.8.4 Calculating priority of scenarios

The priority of a scenario is the combined priority of a scenario plus the priority of the corresponding use case. Thus, priority of scenario s_i related to use case uc_j is given as:

$$\text{Priority}(s_i) = w_1(\text{Priority}(uc_j)) + w_2(\text{Priority}(s_i))$$
$$w_1 \text{ and } w_2 \text{ are weights} \mid w_1 + w_2 = 1.$$

This combined priority is used for ordering scenarios by type. Main scenarios are considered first, followed by exception and alternate scenarios. Figure 4.17 shows the prioritized scenarios of the 'Validate User' activity diagram. The order of prioritization P_1, P_2, P_3, P_4, P_5 is based on the definitions stated above. Table 4.8

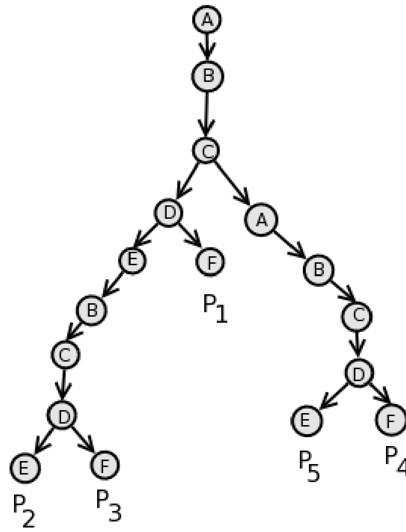


Figure 4.17: Tree T showing prioritized paths

Table 4.8: Prioritized scenarios

Path	**Type**	**Priority**
A→B→C→D→E→B→C→D→E	Alternate	II
A→B→C→D→E→B→C→D→F	Alternate	III
A→B→C→D→F	Main	I
A→B→C→A→B→C→D→F	Alternate	IV
A→B→C→A→B→C→D→E	Alternate	V

shows prioritized scenarios categorized based on the type of scenario. Highest priority is given to to main scenarios and then to alternate and exception scenarios. The path with highest weight gets highest priority.

4.3.9 Prioritization: Summary

In summary, this section presented techniques for prioritizing requirements(use cases) from use case diagrams and scenarios from activity diagrams based on the primitives defined by OMG. Prioritization values thus obtained are used along with prioritization values obtained from customers. A weighted sum of the two is taken to compute priority of use case or scenario. The advantages of the methodology are:

- The technique considers both customers as well as developers viewpoint for prioritization of scenarios.
- Customer knowledge on priority may differ. To incorporate this, weights are given for customer priority as well as technical priority. Weights can be adjusted based on customer knowledge of requirement priority.
- Priority calculated using primitives of use case and activity diagrams is automated and does not require manual intervention.
- This technique may be combined with other techniques to aid prioritization.

In this work, prioritization of scenarios by the customer is not considered. However, in case it is available, a weighted sum similar to the method used for use cases can be applied. Based on priority, scenarios are ordered/ranked for further consideration particularly in case of integration and system testing. As the number of scenarios even for a medium size application could be large, there is a need for selecting scenarios in order of their importance based on some defined context. In the next section, scenario selection is discussed.

4.4 Scenario Selection

4.4.1 Introduction

Specification based testing involves generating test cases from specification, here, UML. The number of automatically generated test scenarios from UML activity diagrams, is large. A test suite is composed of a set of test cases wherein each test case covers a set of actions to be performed by the user(actor) of the system. Given limited resources in comparison to the size and complexity of software, there is need for early and effective feedback with reference to defects in software.

As mentioned earlier, functional requirements are recorded using use case diagrams. Each use case is elaborated using activity diagrams i.e. a use case may have one or more activity diagrams representing main, alternate and exception flows. Automated scenario generation from activity diagrams is done based on methodology followed in Section 4.2. It is considered that loops in activity diagrams are traversed twice. Thus, for each use case, a set of scenarios is obtained whose size can be large dependent on the size and complexity of activity diagrams.

A look at scenarios thus generated reveals that parts of test scenarios are common. A study of similarity among a given set of scenarios is carried out for reducing effort in testing the scenarios and concentrating on scenarios that are different or dissimilar. The objective of the approach is to reduce effort involved in running scenarios that are highly similar by picking a representative of such scenarios and including more dissimilar scenarios with the aim of increasing fault detection and coverage. Test scenario selection thus involves selecting a subset of scenarios that best represent the set of scenarios. A threshold value 't' (percentage of test scenarios to be selected) is obtained from the quality assurance team engineers. It is essential to select scenarios within the threshold, such that the selection ensures qualitative as well as quantitative success in testing a system. In

this section, techniques for automated test selection are proposed.

An automated strategy for test scenario selection based on the use of Levenshtein distance [Lev65] as a measure of dissimilarity between scenarios is presented in Section 4.4.4 . For each pair of scenarios, the Levenshtein distance is calculated. The least value indicates scenario pairs that are least dissimilar and vice versa. Hence, selection is based on the highest dissimilarity between scenarios. A second technique for test case selection presented is based on the Longest Common Subsequence applied to subscenarios(Section 4.4.5). A subscenario is a contiguous chain of activities within a scenario. The technique looks at similarity between scenarios in terms of the length, weight and position of the common subscenarios in the scenario. The above factors are used to calculate the similarity between each possible pair of scenarios. A similarity value indicates the degree of structural similarity i.e. less the value, more similar they are. Again, a heuristic is used to pick one of the two scenarios. Clustering is the grouping of objects(here scenarios) of similar kind into categories. In this work, hierarchical as well as partitional clustering algorithms are used to cluster scenarios according to a distance measure(Levenshtein distance), such that elements in the same cluster are similar based on some criteria(Section 4.4.6).

Generally, selection of one scenario between two is done randomly [CNM07, YH07]. In this work, heuristics other than random choice are introduced to capture the added advantage of using information available on test scenarios i.e. priority, type, knowledge of the quality assurance team (preferred selection) in the selection process thereby producing better results. The test suite thus obtained is used for testing either directly(for random testing) or forms input for prioritization.

Related work in the area of test case selection is discussed in Section 4.4.2. Section 4.4.3 discusses criteria used to measure the effectiveness of selection techniques. Section 4.4.4 presents a technique for clustering based on Levenshtein

distance. Section 4.4.5 presents a similarity measure based on the length and position of a subscenario in a scenario. Selection technique based on Agglomerative Hierarchical Clustering is described in Section 4.4.6.

4.4.2 Related Work on Test Case Selection

Test case selection provides several benefits in automated testing process. Exhaustive testing being impossible, there is need to determine a subset of test cases that ensure test objectives, namely, maximum coverage, and early fault detection. For this, there is need to select an effective subset of the original test suite. Several techniques have been developed which attempt to maintain quality of throughput by reducing the size of the test suite and at the same time increasing efficiency.

Similarity measures for test case selection have been used by [CANM08, CNM07, CLM04]. Test cases are selected based on the distance value computed between two test cases. Techniques used include pairwise comparison in [CNM07] and Euclidean distance in [CLM04]. Cartaxo et al. use Labeled Transition Systems (LTSs) as the model from which they obtain test cases. In [CLOM06], the authors extend the idea of distance to testing object oriented programs by defining a distance for objects, the 'object distance'. The object distance computes distances between arbitrary objects. They describe a model for representing the differences between two objects, and use Levenshtein distance to calculate distance between class names, object names and its contents.

Burguillo et al [BLFR02] in their work consider heuristic driven techniques for test selection. They use four factors, namely, risk, coverage, cost and efficiency, which helps in classification and selection of test cases according to different criteria. Risk based test selection is the focus of work done by Chen et al [CP03b]. Risk is based on the cost of each test case(valued by both customer and vendor) as well as severity. Risk exposure(RE), a product of cost and severity is taken

as the basis for test selection. Scenarios that cover most critical cases are considered first. Cow Suite tool by [BBM02] use UML use case and sequence diagrams to record requirements. Each use case diagram is elaborated using a sequence diagram, to scenarios forming a tree structure. Weights are assigned based on functional importance such that the sum of weights at any level equals to one. The test generation algorithm used by Cow Suite generates all possible test cases. Selection is done based on the weight of the scenarios obtained by product of weights of all nodes in the path leading to particular scenario.

Activity diagrams are used for regression test selection in [CPS02]. Regression test selection techniques involves selecting a subset of test cases to determine if the modified program has the same behaviour as a previous, acceptable version of the program running on T, a set of test cases. A framework for analyzing regression test selection techniques is presented in [RH96]. The framework consists of four categories: inclusiveness, precision, efficiency, and generality. They analyze different techniques on the four factors. In [RH97], the authors construct control flow graphs(CFG) for a procedure or program and its modified version. They use the CFGs to select tests that execute changed code from the original test suite.

In this work, use case diagrams model functionality of a system. Each use case is elaborated using activity diagrams. Scenarios are generated from activity diagrams which form the input of the test selection process. This work also uses distance measure to determine similar scenarios, like work in [CANM08, CNM07, CLM04]. However, in this work different measures are used to select between two scenarios. Information like scenario type(main, alternate, exception) and priority are available as inputs provided by the quality assurance team/customers or through automated techniques presented in previous sections. This information can be used to help in better selection of scenarios.

4.4.3 Selection Criterion

Similarity based test selection is based on distance measure or a similarity measure that gives a value indicating similarity between two scenarios i.e. given two scenarios with a high similarity value, there is need to pick one of the two. Current work in the area use random selection to pick scenarios by selecting one or the other. To improve the criteria for picking one of the two scenarios, other heuristics can be considered based on properties of the scenarios like priority and type. One or a combination of heuristics can be used for picking one of the scenarios. The heuristics used in this work is given below:

Random : One of the two scenarios, is selected randomly.

Priority based : Scenario with higher priority is selected. Priority of scenario is calculated based on one or a combination of the techniques: customer assigned, statement coverage, risk based, random, etc. [SWO05, RUCH01, EMR02]. In case of equal priority, a scenario is selected randomly.

Subsumption : Given two test scenarios, X and Y, X subsumes Y if and only if the scenario Y is contained in X in that order. Hence, X is selected.

Type : Main scenario(flow) gets priority over alternate and exception scenario.

Preferred Scenarios : Preferred scenarios are the subset of all scenarios listed by the quality assurance engineers as those that must compulsorily be tested. All such scenarios are included in the test suite followed by selection using one of the above techniques.

4.4.4 Scenario Selection based on Levenshtein Distance

4.4.4.1 Introduction

Levenshtein distance [Lev65], named after Vladimir Levenshtein, is a metric for measuring the amount of difference between two sequences (i.e., the so called *edit distance*). The Levenshtein distance between two strings is given by the minimum number of operations needed to transform one string into the other, where an operation is an insertion, deletion, or substitution of a single character. A modification of the Levenshtein distance calculates genetic distance. *Genetic distance* between two words is taken as the edit distance divided by the number of characters of the longer of the two. The genetic distance is thus, any value between 0 and 1. The algorithm has a complexity of Θ(mn), where m and n are the lengths of the strings.

For example, consider two strings,

T E S T I N G and

T E S T E D

Edit distance : 3 (Substitute 'I' by 'E'; substitute 'N' by 'D'; and delete 'G')

Genetic distance : 0.43(Edit distance 3 divided by 7, length of longer string)

The algorithm to calculate the Levenshtein distance between two strings is given in Algorithm leven(s1,s2).

4.4.4.2 Application to Test Case Selection

For each use case, uc_i a set of scenarios, S, deduced from all activity diagrams elaborating the use case is obtained. Each of these scenarios is specified as a string of characters where each character represents an activity. The Levenshtein distance

Algorithm 4 leven(s1,s2) //calculates Levenshtein distance between two scenarios

```
1:  {  Input :
    s1,s2 : scenarios of size m and n.
    Output :
    Distance value between scenarios
    Initialize :
    d = a matrix of size m × n, set to 0
2:  for i = 0 to m do
3:     d[i,0] = i
4:  end for
5:  for j = 0 to n do
6:     d[0,j] = j
7:  end for }
8:  for j = 1 to n do
9:     for i = 1 to m do
10:       if (s1[i] = s2[j]) then
11:          d[i,j] = d[i-1,j-1]
12:       else
13:          d[i, j] := minimum(d[i-1, j] + 1, d[i, j-1] + 1, d[i-1, j-1] + 1)
14:       end if
15:    end for
16: end for
17: if (m > n) then
18:    return d[m,n]/m
19: else
20:    return d[m,n]/n
21: end if
```

(genetic distance) is used to calculate the dissimilarity between two scenarios[7]. The genetic distance between each pair of scenarios pertaining to a use case is stored in a matrix M = n * n where n is the number of scenarios. The minimum value in the matrix belongs to the scenarios, say, s_1 and s_2 i.e. having minimum dissimilarity. Selection criterion described in Section 4.4.3 is used to select/pick one scenario between the two.

4.4.4.3 Algorithm

Algorithm *Select(S, P, Tq)* shows the steps involved in selecting test cases using Levenshtein genetic distance. S, the set of scenarios and P, the percentage of test cases to be selected is given as input. Tq denotes the selection technique to be adopted. D is the matrix, of size n × n, containing the Levenshtein distance calculated between scenarios. T is the threshold calculated based on the percentage of test cases to be selected i.e. the number of test cases to be selected. First, the Levenshtein genetic distance for the scenarios is calculated. Then, selection of scenarios is done based on the technique (random, priority, type) selected. In case two scenarios are equal, i.e. two scenarios have same priority, then random selection is done.

4.4.4.4 Example

Consider as an example, Figure 4.18 showing the graph derived from an activity diagram. Activities are represented as nodes and transitions as edges. Using the test scenario generation algorithm, all paths in the graph are deduced. The

[7]A part of the work stated here is published in the Proceedings of the 6th International Conference on Distributed Computing and Internet Technologies (ICDCIT 2010) and appears in Springer-Verlag

Algorithm 5 Select(S,P,T_q) // selection of a subset of scenarios

1: { <u>*Input :*</u>
S : the set of scenarios.
P : percentage of test cases to be selected.
T_q *: the selection technique adopted.*
n : size of S, the set of scenarios
T : threshold for test case selection (n times P/100)
<u>*Output :*</u>
Selected subset of Scenarios
<u>*Functions called:*</u>
D(min) : returns the scenarios having minimum value in D
SelectScenario(T_q,r,c) : returns scenario selected according to technique T_q
levenshtein(S) : returns a matrix containing distance values between scenarios
<u>*Initialize :*</u>
count = 0
D = matrix of size n × n, containing Levenshtein distance between scenarios
TS = test suite after selection
}
2: D = levenshtein(S)
3: **while** (D <> 0) **do**
4: // while matrix D is not null
5: **if** (count < T) **then**
6: // if number of scenarios selected is less than threshold, T
7: (r,c) = min(D) // r and c are the scenarios having minimum value in D
8: result = Selectscenario(T_q,r,c) // result contains the scenario selected w.r.t Tq
9: Add scenario 'result' to TS
10: Remove scenario 'result' from D
11: count = count + 1
12: **else**
13: exit
14: **end if**
15: **end while**

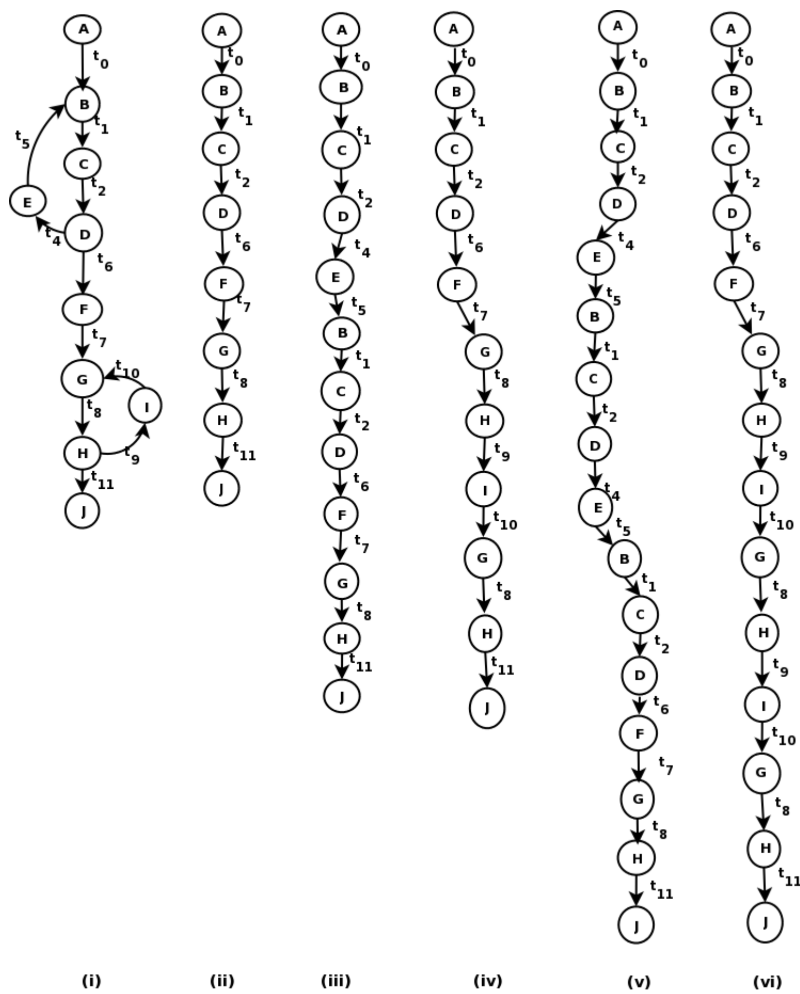


Figure 4.18: Set of Scenarios (i)Graph (ii) 5 out of 9 scenarios generated

activity diagram consists of nine scenarios(paths from the start node to the end node) five of which is shown in Figure 4.18(ii). Here, loops are executed atmost twice. For brevity, only activity names are considered. Table 4.9 shows the set of scenarios generated.

Table 4.9: Set of scenarios generated from graph in Figure 4.18

[1] A → B → C → D → F → G → H → J
[2] A → B → C → D → F → G → H → I → G → H → J
[3] A → B → C → D → E → B → C → D → F → G → H → J
[4] A → B → C → D → F → G → H → I → G → H → I → G → H → J
[5] A → B → C → D → E → B → C → D → E → B → C → D → F → G → H → J
[6] A → B → C → D → E → B → C → D → E → B → C → D → F → G → H → I → G → H → J
[7] A → B → C → D → E → B → C → D → F → G → H → I → G → H → J
[8] A → B → C → D → E → B → C → D → F → G → H → I → G → H → I → G → H → J
[9] A → B → C → D → E → B → C → D → E → B → C → D → F → G → H → I → G → H → I → G → H → J

Table 4.10: Levenshtein distance calculated between scenarios in Table 4.9

	1	2	3	4	5	6	7	8	9
1		.27	.33	.43	.5	.58	.47	.56	.63
2			.42	.21	.57	.42	.27	.39	.5
3				.5	.25	.37	.2	.33	.45
4					.56	.37	.33	.22	.36
5						.16	.31	.39	.27
6							.21	.26	**.14**
7								.17	.32
8									.32
9									

The matrix D with the pairwise Levenshtein distance(genetic distance) calculations for the scenarios listed are shown in Table 4.10. Values indicate level of dissimilarity between the scenarios. Consider that the desired path coverage percentage is 50%. Then, it is required to select four test cases out of a total set of nine scenarios. Hence, threshold value = 4.

The scenarios with minimum dissimilarity is between scenarios '[6]' and '[9]' having Levenshtein distance = 0.14. Assuming that 'Random' is the selection criterion, scenario '[9]' is selected. Scenario '[9]' is removed from the matrix. Table 4.11 shows matrix D after elimination of scenario '[9]'. The next minimum value is 0.16 between scenarios '[5]' and '[6]'. This procedure continues until the

threshold value is reached. The test suite with 50% coverage consists of scenarios [9],[6],[8] and [7].

Table 4.11: Distance table after selection of scenario 9

	1	2	3	4	5	6	7	8
1		.27	.33	.43	.5	.58	.47	.56
2			.42	.21	.57	.42	.27	.39
3				.5	.25	.37	.2	.33
4					.56	.37	.33	.22
5						.16	.31	.39
6							.21	.26
7								.17
8								

4.4.5 Scenario selection based on Common Substrings

A subscenario is a contiguous chain of activities within a scenario. In this work, the idea of substrings is used to introduce a new similarity measure that considers the common subscenarios, their lengths, and weight with respect to the position in the scenario[8]. These factors are used to calculate the similarity between each possible pair of scenarios. A similarity value indicates the degree of structural similarity i.e. more the value, more similar they are. The pairs are arranged in ascending order of their similarity value. A bottom up approach is followed in selecting pairs and selection of pairs continue until the desired test coverage criteria is met. From each pair, one path is selected as test scenario.

4.4.5.1 The Approach

The common substring between two strings is the set of contiguous set of characters in a string. The same idea is adapted for selection of test scenarios. In this work, a common subscenario is the longest contiguous chain of activities that exists in

[8]A part of the work stated here is published in the Proceedings of the 4th International Conference on Industrial and Information Systems, 2009 and appears in IEEE Xplore Digital Library

both scenarios. Common subscenario can be viewed in two terms: it can be said that longer the subscenario, better is the match between two scenarios i.e. they are similar; secondly, if the sum of the lengths of all common subscenario is close to the length of larger scenario, again, it can be said that the two scenarios are highly similar.

Based on the above, a new measure is introduced that takes into consideration common subscenario in a scenario, their length as well as relative position in the scenario. It is assumed that length of the common subscenario must be a value between 3..n considering that a subscenario is of the form
$< activity >< activityedge >< activity >$
where an activity edge is an open arrowhead line connecting two activity nodes. For brevity,only activity names are consider here.

The intention is: one, by considering common subscenarios, similarities between scenarios is taken into account; two, length of the subscenario determines number of contiguous activities that are similar; and three, different weights can be given based on the position of the subscenario. Activity diagrams elaborate a functionality. As can be observed, activity diagrams can be sliced based on the importance or risk value of the activities. Quality assurance engineers define the slice and assign weights to the slices such that

$$W = \{w_1, w_2, ..., w_n\} \mid \Sigma w_i = 1.$$

Figure 1 shows an activity diagram sliced into three. The heuristic used is that slice 2 containing computational aspects be given more weightage than slice 3 where results are generated, which is higher than slice 1 wherein inputs are obtained.

On basis of the above, a measure is defined that reflects length and relative importance of the subscenario to the size of the scenario.

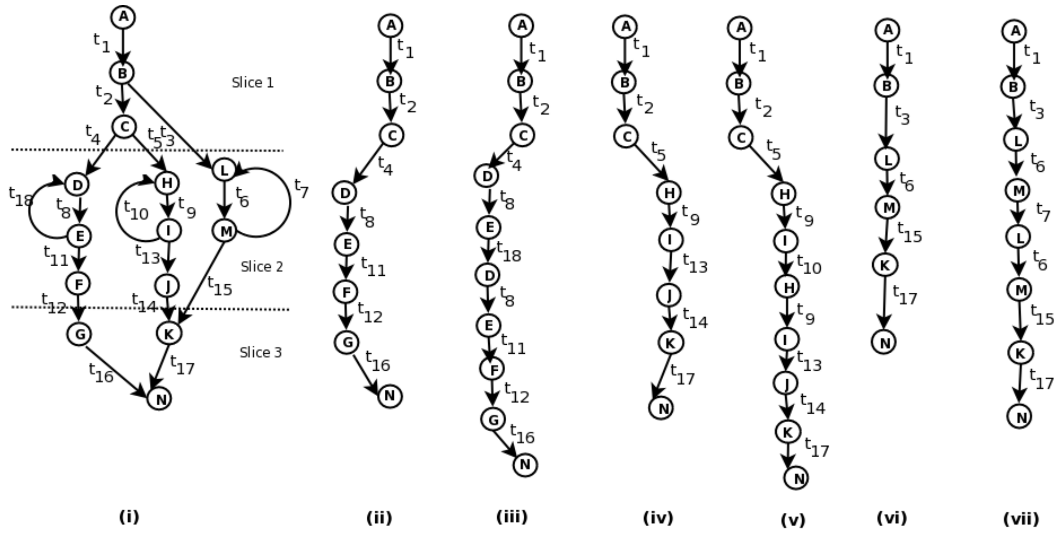


Figure 4.19: Activity diagram showing a. (i) slices b. (ii) - (vii) 6 scenarios in (a)

If s1 and s2 are two scenarios, then,

Sb = $\{sb_1,\ sb_2,....sb_n\}$ is the set of subscenarios, common between s1 and s2;

W = $\{w_1,\ w_2,....w_n\}$ is the relative weights of the subscenario in Sb with respect to position in scenario. w_i takes values between 0 and 1. If weights are different, then minimum of the weights is subtracted with the displacement value(i.e. if subscenarios are similar but at differing positions, then the weight is lesser than if the subscenarios were at the same positions in the scenarios). If both the scenarios are equal, then weight=1. If both scenarios do not have any common subscenarios, then the weight=0.

length(sb_i**)** returns the length of the subscenario.

len = length(s1), if length(s1) > length(s2); else, length(s2).

Then,

$$Similarity(s1, s2) = \frac{\Sigma(w_i \times (length(sb_i)))}{len} \quad (4.1)$$

For example, consider two scenarios(Figure 4.19),

s1 = "A B C D E F G N" and

s2 = "A B C H I J K N".

The activity diagram pertaining to scenarios are sliced into three, and assigned

weights, 0.3, 0.5 and 0.2.

Then, Sb = {"A B C"}

W = {0.3}

$$Similarity(s1, s2) = \frac{0.3 \times 3}{8} = .11$$

The value 0.11 is the similarity value between the two scenarios.

4.4.5.2 Example

Consider that 50% scenarios from Figure 4.19 need to be selected. Values calculated using similarity measure is shown in Table 4.12.

Table 4.12: Distance values between scenarios using similarity measure for Figure 4.19

	1	2	3	4	5	6
1		**.21**	.11	.09	.07	.07
2			.09	.09	.06	.06
3				.21	.12	.12
4					.10	.10
5						.20
6						

Picking the largest value, 0.21, the highest similarity, one scenario is selected randomly between 1 and 2, say 1, and added to the test suite, TS. Then, scenario '1' is deleted from the matrix(Table 4.13), and the same procedure continues until 50% of scenarios is selected. TS = 1,3,5.

Table 4.13: Distance matrix after deletion of scenario 1

	2	3	4	5	6
2		.09	.09	.06	.06
3			**.21**	.12	.12
4				.10	.10
5					.20
6					

4.4.5.3 Algorithm

Algorithm *Select(S, P, Tq)* shows the steps involved in selecting test cases using similarity measure. SC, the set of scenarios and P, the percentage of test cases to be selected is given as input. D is the matrix, of size n × n, containing the distance calculated between scenarios. T is the threshold calculated based on percentage of test cases to be selected i.e. number of test cases to be selected. First, distance between scenarios is calculated. Then, selection of scenarios is done randomly.

Algorithm 6 $Select(S, P, Tq)$ //calculates distance between two scenarios

```
1: { Input :
2: S : the set of scenarios
3: P : the percentage of test cases to be selected
4: n : size of S, the set of scenarios
5: T : Threshold for test case selection (n X P/100)
   Output :
   Distance value between scenarios
   Initialize :
6: Initialize: count = 0
7: D = matrix of size n × n, containing similarity values between scenarios
   Functions :
8: max(D) is a function that returns the scenarios having maximum value in D
9: Selectscenario(r,c) is a function that randomly selects one of the scenarios r,c
10: Similarity(S) is a function that returns the distance calculated according to
   equation 1
   }
11: D = Similarity(S)
12: while (D <> 0) do
13:   // while matrix D is not null
14:   if (count < T) then
15:     // if number of scenarios selected is less than threshold, T
16:     r,c = max(D)
17:     // r and c are the scenarios having maximum value in D
18:     result = Selectscenario(Tq,r,c)
19:     Add scenario 'result' to TS
20:     Remove scenario 'result' from D
21:   else
22:     exit
23:   end if
24: end while
```

4.4.6 Clustering based Test Selection

4.4.6.1 Introduction

Clustering is the grouping of objects(here scenarios) of similar kind i.e. it partitions objects into mutually exclusive groups. Thus, each group(cluster) consists of objects that are similar among themselves and dissimilar to objects of other groups. Cluster analysis identifies and classifies objects on the basis of the similarity of the characteristics they possess minimizing intra-group variance and maximizing inter-group variance as shown in Figure 4.20. Clustering therefore results in a number of heterogeneous groups with similar(homogeneous) content.

The objective of clustering are:

- discover types from given data
- obtain a sample of several types instead of a big sample set

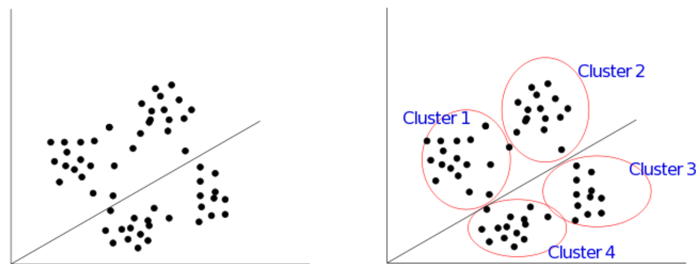


Figure 4.20: Diagrammatic representation of clustering

Similarity criterion is the distance between two or more objects belonging to the same cluster. Techniques that use distance as a means for grouping is called distance based clustering.

Hamming distance is used as the basis for clustering by Yoo et al [YHTS09]. Dynamic execution trace of each test case is used to calculate similarity between features tested. The limitation is that the method requires an execution trace to

be generated from code for clustering. In this work, clustering is used to group scenarios based on similarity obtained from scenarios. Similarity between scenarios is determined in terms of activities and transitions between them using Levenshtein distance. A look at Figure 4.19 shows activities and transitions that are common between scenarios. As mentioned before, exhaustive testing being impossible due to constraints of resources and time, there is need to determine an effective set of scenarios. The commonality in activities and transitions across scenarios(similarity) is used in this work to cluster scenarios. Clustering produces groups of scenarios such that members of a group are similar. One or more scenarios from each group can be used for testing.

In this work, Agglomerative Hierarchical Clustering(AHC) is used for clustering scenarios generated from activity diagrams. Given a set of clusters, and a percentage of scenarios to be selected, scenarios are randomly picked from each cluster. The set of scenarios thus obtained forms a test suite(random ordering). Also, the set of scenarios may be ordered(prioritized) according to some criterion(priority,type) and then executed.

4.4.6.2 Selection Method

Agglomerative Hierarchical Clustering(AHC) technique is used for clustering scenarios.

Agglomerative Hierarchical Clustering

Agglomerative Hierarchical Clustering is a bottom-up clustering method. An AHC clustering procedure produces 'n' single scenario clusters P_n, P_{n-1}....., P_1. P_n consists of n single object 'clusters' which in this case are individual scenarios. At each stage, two clusters which are closest(most similar) are joined to form a

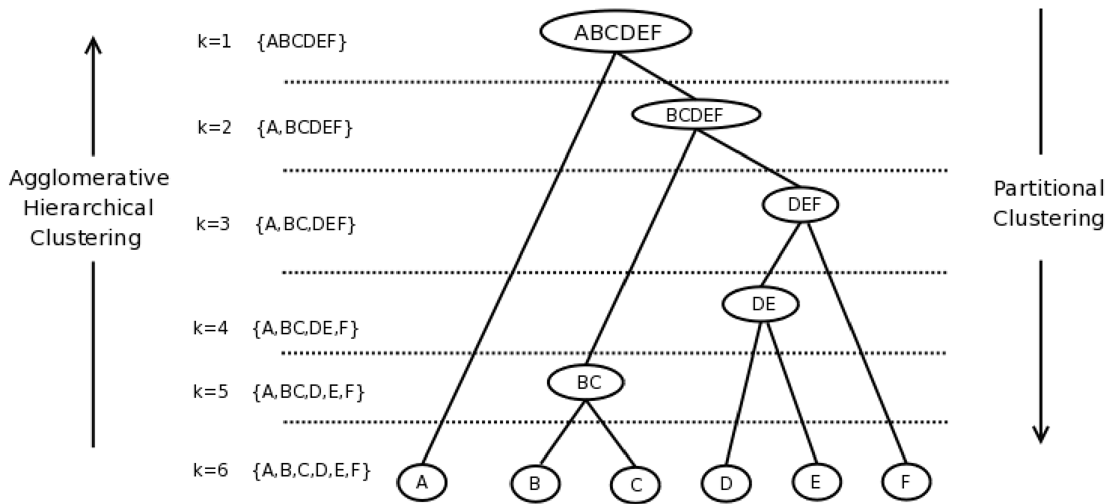


Figure 4.21: Dendrogram obtained by Clustering

new cluster. Finally, P_1 consists of a single group consisting of all n objects(here, scenarios).

Figure 4.21 shows the clusters formed by AHC, algorithm for which is given below. The dendrogram is a tree structure that represents clusters. Also, by cutting the tree at different heights, it is possible to generate k clusters for any k in [1, n].

Algorithm 7 gives the steps involved in Agglomerative Hierarchical Clustering technique. The resulting dendrogram, D, is a tree structure. Similarity value between scenarios is calculated using a similarity/dissimilarity metric, namely, Levenshtein distance. The similarity matrix gives the set of scenarios s_1, s_2 that are similar which is used in the algorithm to determine the clusters.

To obtain a test suite, 'k', the level indicating number of clusters is obtained. All clusters at level 'k' with scenarios belonging to each cluster is returned. To select a percentage of scenarios from the clusters for the test suite, scenarios within a cluster are selected randomly. Again, it would be disadvantageous to run all scenarios belonging to a cluster before executing scenarios from the next as similar scenarios belonging to a cluster get executed before execution of the next cluster.

Algorithm 7 $AHC(S, n)$ // Clustering of a set of scenarios, of size n

1: { *Input :*
S : A set of scenarios
Output :
A dendrogram, D, giving all clusters.
Initialize :
Cl : Cluster }
2: Form n clusters with each scenario.
3: Add the clusters to 'Cl'.
4: Calculate similarity value between scenarios using Levenshtein distance, leading to matrix, M.
5: **while** (Cardinality(Cl) > 1) **do**
6: Find pair of scenarios, s_i, s_j with minimum distance from M.
7: Merge the pair to form a new cluster, C_n.
8: Remove the scenarios from M.
9: Remove the scenario pair from Cl.
10: Add C_n to Cl.
11: Insert C_n as parent of scenarios s_1, s_2 into D.
12: **end while**

This would lead to similar faults getting detected. To avoid such a case, scenarios are picked for execution in one of the following three ways:

- The set of scenarios obtained from the clusters after selection are ordered again, according to some criterion.
- One scenario from each cluster is picked for execution in order till scenarios get exhausted.
- Random selection of set of scenarios obtained from the clusters.

4.4.6.3 Example

Consider the distance matrix for the activity diagram in Figure 4.19. Values calculated using similarity metric is shown in Table 4.14.

Each scenario is taken as a cluster. In each step of the iteration, the closest pair of clusters is taken into consideration. In this case, the closest cluster is between cluster 3 and 4 with shortest distance of 0.2. Therefore, cluster 3 and 4 are grouped

Table 4.14: Distance calculated between scenarios using metric for Figure 4.19

	1	2	3	4	5	6
1	0	.30	.63	.70	.71	.75
2	.30	0	.60	.60	.70	.70
3	.63	.60	0	**.20**	.50	.50
4	.70	.60	.20	0	.60	.60
5	.71	.7	.5	.6	0	.25
6	.75	.7	.5	.6	.25	0

into cluster (3, 4). Then the distance matrix is updated (see Table 4.15). Distance between ungrouped clusters do not change in the distance matrix. To calculate the distance between newly formed clusters and other clusters, single linkage rule is used, where the minimum distance between original cluster and the combined clusters are taken.

Table 4.15: Distance matrix after clustering scenarios 3/4

0	1	2	3/4	5	6
1	0	.30	.63	.71	.75
2	.30	0	.60	.70	.70
3/4	.63	.60	0	.50	.50
5	.71	.70	.50	0	.25
6	.75	.70	.50	**.25**	0

The process continues until there is only one cluster. Dendrogram, D, for the above is given in Figure 4.22. Given a set of clusters, and a percentage of

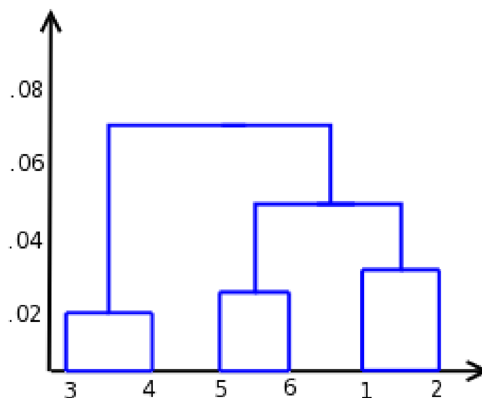


Figure 4.22: Dendrogram obtained by Clustering - AHC

scenarios to be selected, scenarios are randomly picked from each cluster. The set of scenarios thus obtained forms a test suite(random ordering). Also, the set of scenarios may be ordered(prioritized) according to some criterion(priority,type) and then executed.

4.4.7 Scenario Selection: Summary

To summarize, in this section, techniques for test scenario selection is presented. One class of technique use distance measure, one based on Levenshtein distance and another on the Longest Common Subsequence for selection of scenarios. The second class, uses clustering based on the distance measure as a way of selecting scenarios to form a test suite.

Levenshtein distance is used to calculate the similarity between two scenarios based on the number of insertions, deletions and substitutions required to convert one scenario to another. The technique is simple. However, the technique looks at the activities and transitions individually rather than as a subscenario. A second distance measure incorporates a subscenario as being the unit of similarity and adapts the Longest Common Subsequence algorithm. A third technique based on clustering is presented. The Agglomerative Hierarchical Clustering technique is used to cluster scenarios based on a similarity measure, here distance(Levenshtein distance). Given the percentage of scenarios to be considered and the number of clusters, scenarios are selected from the clusters in random.

Thus, a set of scenarios, which is a subset of the original set maybe selected to form a test suite.

4.5 Summary

Techniques for generation, prioritization and selection of scenarios to form a test suite is presented. Automated generation of scenarios from UML use case and activity diagrams leads to large number of scenarios. Given constraints of resource and time in testing the software, there is need for automated techniques for generation, prioritization and selection of scenarios.

Concurrent activities in an activity diagram are represented using fork-join constructs. As the order of execution of concurrent activities cannot be predetermined, scenario generation explores all possibilities. However, it is impossible to test exhaustively. To overcome this, domain dependency existing between concurrent activities is used by assigning priority to activities or categorizing them into levels. The advantage of the approach is that only valid scenarios are generated thereby reducing considerably the number of scenarios generated for testing.

The ordering of scenarios generated through automation is random, and such an order does not ensure effectiveness in terms of objectives like fault detection and coverage. Besides customer inputs on priority, techniques to prioritize requirements and scenarios based on the primitives of the use case and activity diagrams are introduced in this work. A weighted sum of customer inputs as well as priority based on primitives is used to prioritize requirements and scenarios. Prioritization ensures early feedback to developers for bug fixing and further development.

Prioritization orders scenarios according to some criterion. However, testing exhaustively is not possible. Hence, there is need for selecting a subset of scenarios that best represents the complete set of scenarios. In this work, distance measures are used as basis for selecting a subset of scenarios based on their similarity. Another technique used is to cluster scenarios and select a representative subset from the clusters.

Different criterion are used besides random for both prioritization and selec-

tion. They include priority of scenario and type of scenario(main,alternate and exception) used for selection. The ordering and selection of scenarios are based on two criterion, namely, rate of fault detection and percentage of coverage. Average Percentage of Faults Detected(APFD) metric is used to calculate the rate of fault detection. Coverage is measured in terms of activities, transitions, basic path and additional path coverage criterion.

Chapter 5

Ontology-based Scenario Management

Management of test scenarios involves storing and retrieving scenarios fulfilling a criteria like selecting certain chunk of scenarios pertaining to a given use case. In order to achieve this, knowledge on domain entities and their relations are to be maintained. This knowledge helps in categorizing scenarios as per the approaches used for testing. Ontologies provide a mechanism to share and reason on knowledge that is captured. The objective of the work described in this chapter is to use ontologies to aid test management.

5.1 Introduction

One of the challenges faced in software engineering is building software with quality at reduced cost, time and effort [BCO09]. This requires a clear understanding of the elements that build a software and the relations among them. In this direction ontologies can aid in reusing and performing inference on knowledge gathered from requirements captured using UML diagrams. Ontologies also enable communication between different people involved in building a software and hence can be used as a common mechanism for requirements analysis.

Based on the work in previous chapters, use cases and scenarios have been

obtained. Now, during testing, test cases related to use cases and scenarios are to be picked as per a testing criteria like required percentage of test coverage. For the purpose, an ontology based approach for storage and retrieval of use cases and their corresponding scenarios is proposed. Ontology helps to retain domain specific relations among use cases and scenarios; it enables to reason over ontology facilitating retrieval of scenarios from repository for a given criteria. A metamodel for use case and scenario management is proposed so that a reasoner can be applied while querying. For the purpose, Protege, an open source ontology editor has been used. Further, the ontology metamodel is extended for test process management.

The work reported in this chapter is organized as follows: Section 5.1.1 provides motivating examples for using ontology. Section 5.1.2 discusses the use of an ontology as a repository for test management. The use of ontologies in different areas and particularly in testing is discussed in Section 5.2. Section 5.3 enumerates the steps involved in building an ontology. Application of the steps to build an ontology for testing based on the testing concepts is also discussed. Section 5.3.1 describes the process of implementing using Protege, an open source ontology editor. Extension of ontology using concepts extracted from SWEBOK[1] and using the primitives of relevant UML diagrams is discussed in Section 5.5.

5.1.1 Motivating Examples

This section discusses examples that motivate work on test management using an ontology.

Example 1:

Fig. 5.1 shows the relationship between use case, scenario and classes. A requirement shown in a use case diagram and detailed by scenarios is implemented by

[1]Software Engineering Book of Knowledge provides a consistent view of software engineering, its boundary and contents. Available at www.swebok.org/

classes. For querying on test process with respect to a requirement, there has to be traceability for the relations existing among the artifacts. e.g. 'project management' has scenario 'wage calculation' implemented by classes, say, 'pay-calculate', 'award-calculate', 'utility-award', etc. The relations among these artifacts can be shown using an ontology. Using an E-R diagram for the same example does not show the impact on testing. That is, we can say that for every use case that is to be tested, corresponding scenarios must be tested. Also, unit testing of all classes that make up the scenario must be done. This requires a transitive relationship to be defined such that for each use case, all corresponding scenarios as well as related classes have to be tested. Usually class repositories are stored in an RDBMS, but still has limitations.

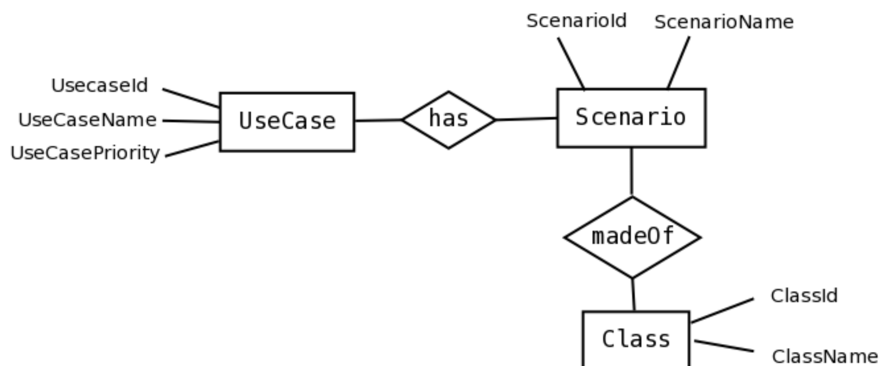


Figure 5.1: Entity Relationship diagram for the relation 'Use case has scenarios, and each scenario is made of classes'

Example 2:

The timetable scheduling problem in a college environment is another case where representing all relationships and constraints cannot be done explicitly in an RDBMS. The time table scheduling problem involves, allotting hours for a particular class belonging to a course. Entities of the system include instructor, class, course, classroom, classlength and subject. One of the challenges in the construction of the timetable is to ensure that constraints on faculty, courses, classrooms, time-slots, or subjects are not violated. Constraints include:

a. A class is taught by one instructor only.

b. An instructor cannot teach more than one class at a point of time.

c. A class may use more than one classroom. However, a class may use only one classroom at a time.

d. No classes share a classroom at the same time.

e. A class cannot have more that one course at a time.

f. A class has a scheduled length of one/two hours.

Though the basic relationships of subject, course, teacher, classroom, period can be represented, the other constraints and preferred options cannot be explicitly represented in an RDBMS. As shown in the above examples, all constraints on the system cannot be captured using RDBMS. However, ontologies help capture constraints related to concepts in the system.

5.1.2 Ontology for Repository and Reasoning

The common practice is to use a database to store the requirements of a system and query the same. However, it is not possible to capture all constraints related to a system using the representation provided by relational databases. Examples showing situations where a relational database is insufficient to represent the conceptual relations are discussed in Section 5.1.1.

For the following disabilities [MW] found with RDBMS, ontology is considered as useful repository.

- Lack of hierarchy. Relational models have no notion of a concept/query hierarchy.

- Representing m:n relationships. Relation between entities are represented using 1:n relationship in RDBMS. m:n relationships are represented as a set

of m, 1:n relationship.

- Access to data. There is need to have access to the database to query the database.

- Knowledge of a query language. User must have knowledge of a query language supported by the specific database.

Representing requirements using ontologies is found more useful than RDBMS for being expressive and flexible to manage. Querying on requirements stored in a relational repository requires prior knowledge on entity relations(i.e. domain structure) whereas in case of ontology, repository queries can be exploratory for traversing links among concepts that make the ontology. Different areas where ontologies have been used is discussed in the next section.

The use of analyzing the requirements of a domain through an ontology of software testing is to allow reasoning about the information represented [AvH04]. For instance :

a. Consistency. Suppose it is declared a and b are instances of the concept U. Also, it is defined that a and b are different individuals and the relation 'includes' is irreflexive and asymmetric. Then, if it is defined that a includes b and b includes a, there is inconsistency.. This error can be detected on reasoning of test ontology.

b. Inferred relationships. If concept X is equivalent to concept Y, and concept Y is equivalent to concept Z, then X is equivalent to Z.

c. Membership. If a is an instance of concept U and U is a subconcept of Y, then it can be inferred that a is an instance of Y. For example, consider two concepts, *'Student'* and *'MCA Student'* where *'MCA Student'* is a subconcept

of *'Student'*. Also consider that *'y'* is an instance of *'MCA Student'*. Then, it can be inferred that *'y'* is a *'Student'*.

d. Classification. Suppose it is declared that certain property-value pairs are a sufficient condition for membership in a concept A. Then, if an individual x satisfies the condition, it can be said that x is an instance of A.

5.2 State of the Art

A survey of current work on ontologies for software reveals that ontologies have been built with varied objectives in mind. One of the objectives of building ontologies is to facilitate collaboration of remote teams in multi-site distributed software development. SWEBOK has been used as the point of reference to build ontologies of software engineering [WC05, WCD05]. The ability of ontologies to query necessary and relevant information regarding the domain concerned is exploited. Dillon et al [DCW08] have focused on the same issue by developing a software engineering ontology that defines common sharable software engineering knowledge as well as information about particular projects. The software engineering ontology consists of five sub-ontologies for software requirements, design, construction, testing and tools and methods. The software testing ontology in particular, consists of subontologies, namely: test issues sub-ontology, test targets sub-ontology, test objectives sub-ontology, test techniques sub-ontology and test activities sub-ontology. Both [WC05, DCW08] provide the advantage of development of a consistent understanding of the meaning of issues(terminology and agreements) related to a project by different people distributed geographically across locations.

A second objective for building ontologies is to access ontology mediated information [DCW08]. Rules can be written about relationships between concepts

in an ontology and the same can be used for query processing. The advantage over databases is that new facts can be inferred or reasoned with asserted facts. An example of this objective is the Protein Ontology built to integrate protein knowledge and provide a structured and unified vocabulary to represent concepts related to protein synthesis.

Thirdly, ontologies are used in the area of semantic web services [DCW08]. Several issues need to be addressed in the area of web services that include, selection of architecture, discovery of service, selection of service and composition and coordination of services to meet requirements. Web services are semantically annotated to assist in the process of discovering and selection for which a combination of ontologies and Web 2.0 is used.

A fourth area in which ontologies are used is multi-agent systems [DCW08, NPT08]. Multi agent systems involve multiple autonomous agents that collaborate with one another to fulfil goals of the system. Agents have a knowledge base that provides some intelligence. Also, the system is distributed and decentralized where agents are geographically distributed and communicate mainly through message passing. To maintain coherence and consistency of knowledge among agents, an ontology is used as a common knowledge base that is shared by all agents. This facilitates communication and coordination among agents.

Other ontologies that have been built include the disease ontology, manufacturing ontology and different financial system ontologies [DCW08]. A software test ontology(SwTO), that deals with the software testing domain has been built by [BCO09]. The ontology is used along with a test sequence generator to generate test sequences to test the operating system domain, Linux. Zhu et al [ZH05] in their work present an ontology of software testing. They discuss use of an ontology in a multi-agent software environment context to support the evolutionary development and maintenance of web-based applications. Software agents use the

ontology as the content language to register into a system and for test engineers and agents to make test requests and report results. The paper describes how the concepts of the ontology and the relations between them are defined in UML.

Thus, ontologies have found use in different areas like software engineering, semantic web services and multi-agent systems. The next section details the steps involved in building an ontology with focus on testing.

5.3 The Proposed Approach

One of the well-known methods to build an ontology is the Methontology strategy [MF97]. A generally applicable method to construct domain knowledge model and validate the same is proposed. The ontology development process is composed of the following steps: planning, understanding, knowledge elicitation, conceptualization, formalization, integration, implementation, evaluation, documentation and maintenance. Noy et al [NM01] follow an iterative design process to build an ontology. A set of steps to build an ontology is proposed. This work adapts ideas from both works.

5.3.1 Building an Ontology for testing

In this section, the steps discussed in Section 5.3 are applied to build an ontology for testing.

5.3.2 Design of Ontology

5.3.2.1 Defining concepts

Steps described in building an ontology are applied in the context of software testing and are explained below:

1. **Determine domain and scope of ontology :** The objective of this work is to provide a framework for testing, specifically specification based testing wherein specification is captured using UML. Representation and management of use cases and scenarios are in focus while building an ontology. The objective is to use this ontology for enumerating test scenarios to form a test suite for testing.

   In the ontology, concepts related to requirements and scenarios like, use cases, related users(actors), scenarios related to each use case are included. In this work, the objective of building an ontology is to provide a means to manage scenarios. Some of the questions that the ontology should answer includes:

   - Which requirements involve use of activity 'x'?
   - What are the scenarios required to test use case 'UC'?
   - What are the use cases that include use case 'UC' ?
   - What are the use cases that extend use case 'UC' ?
   - List all scenarios of the use case 'UC' that include use case 'UC1' having priority 'p'.

2. **Defining concepts in the ontology :** To define an ontology for testing, a list of terms and the related properties are listed. For example, important terms related to this work inlude: *use case*, *actor*, *scenario*, *activity diagram*, *priority*; different types of priority include *customer assigned priority* and *structure based priority.*

3. **Create a concept hierarchy :** First, terms that independently describe objects are considered. The terms form concepts in the ontology. For example, use case, actor, scenario, priority form concepts in the ontology. The

next step involves organizing concepts into hierarchical taxonomy. For example, *customer assigned priority* and *structure based priority* are subclasses of the class *priority.* i.e. *customer assigned priority* is a type of priority. Therefore, customer priority is a subconcept of the *priority* concept.

4. **Defining properties and constraints :** Concepts themselves are incapable of answering questions such as those enumerated in Step 1. There is need to describe internal structure of concepts. For example, consider the concept *use case.* Properties related to the concept includes *haspriority*, *hasscenario*, *hasactor.* For each property, the concept it describes must be determined. The properties are attached to concepts.

5. **Creating instances :** Instances are individuals belonging to a concept.

5.3.2.2 Defining relations between concepts

For each concept, conditions are defined as to how the concepts interact for realizing an objective. An example of sufficient and necessary condition for some of the concepts is elaborated below:

Use case: Condition

i. *UseCase* is a sub concept of *System.*
ii. Individuals of *UseCase* relate itself to individuals of the *Actor* concept through the *hasActor* property.
iii. A *UseCase* must be related to at least one actor in the *Actor* concept through the *hasActor* property.

Use case: Properties

i. *hasActor* is an inverse object property of *isActorOf*, whose domain is *UseCase* and range is *Actor.*
ii. *hasUseCasePrecondition* is a functional object property, whose domain is UseCase and range is UCPrecondition.
iii. *hasUseCasePostcondition* is a functional object property, whose domain is UseCase and range is UCPostcondition.

iv. *hasUseCasePriority* is a functional object property, whose domain is UseCase and range is UCPriority.

v. *hasUseCaseName* is an object property.

Actor : Condition

i. *Actor* is a sub concept of *System*.

ii. Individuals of *Actor* relate itself to individuals of *UseCase* concept through *hasUseCase* property.

iii. An *Actor* must be related to at least one *UseCase* in the *UseCase* concept through the *hasUseCase* property.

Actor: Properties

i. *associate* is a functional object property, whose domain is *Actor* and range is *Usecase*.

ii. *hasPriority* is a functional object property, whose domain is Actor and range is ActorPriority.

Scenario : Condition

i. *Scenario* is a sub concept of *System*.

ii. Individuals of *Scenario* relate itself to individuals of *UseCase* concept through *hasScenario* property.

iii. A *Scenario* must be related to at least one *UseCase* in the *UseCase* concept through the *hasScenario* property.

Scenario: Properties

i. *hasScenario* is an inverse object property of *isScenarioOf*, whose domain is *UseCase* and range is *Scenario*.

ii. hasPrecondition is a functional object property, whose domain is Scenario and range is Precondition.

iii. hasPostcondition is a functional object property, whose domain is Scenario and range is Postcondition.

iv. hasPriority is a functional object property, whose domain is Scenario and range is Priority.

Activity : Condition

i. *Activity* is a sub class of *System*.

ii. Individuals of *Activity* relate itself to individuals of *Scenario* concept through *hasActivity* property.

iii. An *Activity* must be related to at least one *Scenario* under a *UseCase* concept through the *hasActivity* property.

Activity: Properties

i. *hasActivity* is an inverse object property of *isActivityOf*, whose domain is *Scenario* and range is *Activity*.

ii. hasActivityPrecondition is a functional object property, whose domain is Activity and range is Precondition.

iii. hasPostcondition is a functional object property, whose domain is Scenario and range is Postcondition.

iv. hasPriority is a functional object property, whose domain is Scenario and range is Priority.

The ontology built following the steps above is shown in Figure 5.2 below:

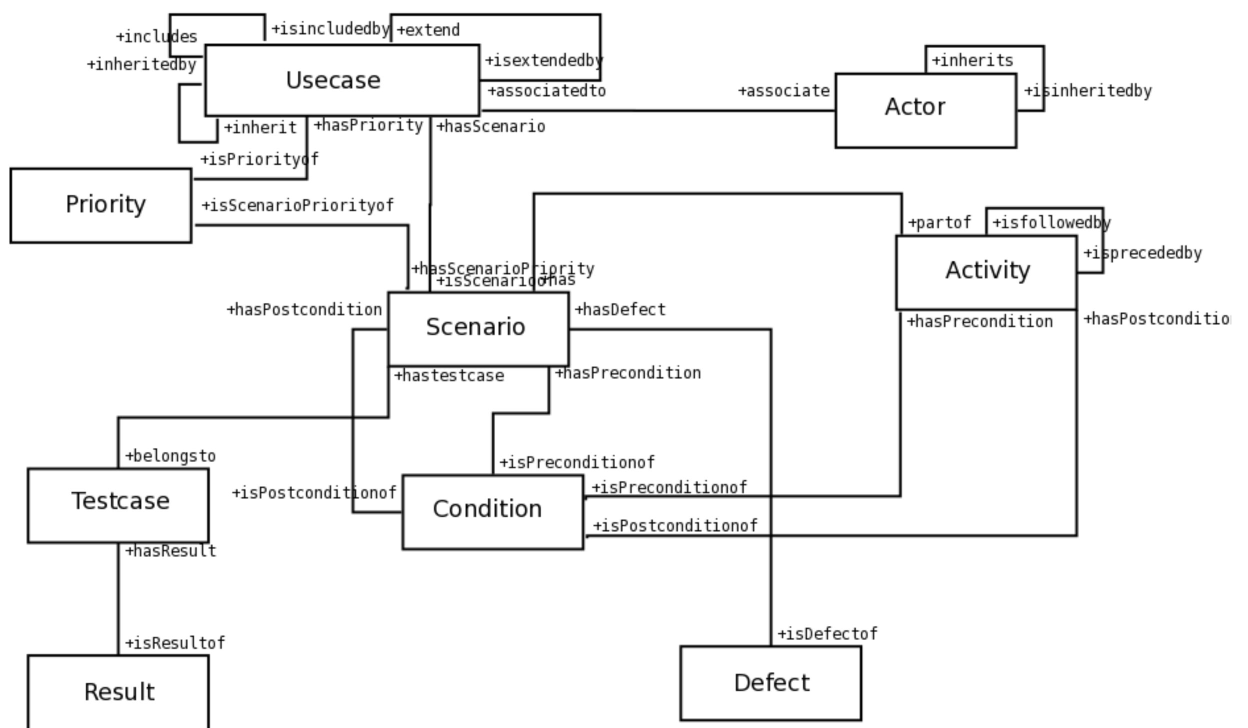


Figure 5.2: Class diagram showing relation among concepts of the ontology

5.3.3 Implementation using OWL

OWL has been chosen to implement the ontology due to its knowledge representation capabilities(concepts, individuals, properties, relationships and axioms) and the possibility to reason about the concepts and individuals[MAQ]. Other ontology languages that can be used include RDF, DAML and DAML+OIL.

Given the use case and activity diagrams, an XSLT(eXtensible Stylesheet Language Transformation) document is written that defines the transformation rules to covert an XML file to OWL file[DuC]. An XSLT processor aids in performing the transformation based on the rules. Following are the transformation steps from XML to OWL using XSLT as shown in Figure 5.3.

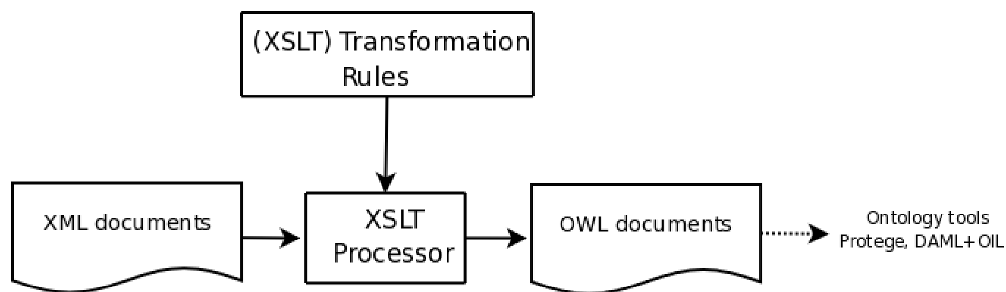


Figure 5.3: The transformation process

- Input. The use case and activity diagrams serve as input to the transformation process. In this work, scenarios have been generated from activity diagrams[2].

- Run the XSLT processor. The XSLT file provides the template for transformation. The input file is read and if it finds the predefined pattern, it replaces the pattern with another according to the rules.

- Output of the process is OWL specification. Ontology tools like Protege[3]

[2]It is to be noted here that scenario generation algorithm in this work generates scenarios for system testing

[3]Protege is a free, open source ontology editor and knowledge-base framework. Available at http://protege.stanford.edu

can be used to verify syntax and validate semantics captured.

Steps described in building an ontology(Section 5.3.1) applied in the context of software testing are explained below:

1. **Defining concepts in the ontology :** To define an ontology for testing,

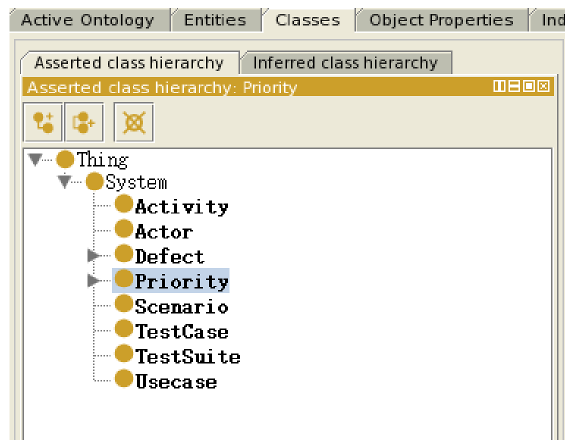


Figure 5.4: Concepts in the ontology

   a list of terms and the related properties are listed. For example, important terms related to this work inlude: *use case*, *actor*, *scenario*, *activity diagram*, *priority*; different types of priority include *customer assigned priority* and *structure based priority*. Figure 5.4 shows some of the classes defined.

2. **Create a concept hierarchy :** First, terms that independently describe objects are considered. The terms form concepts in the ontology. For example, use case, actor, scenario, priority form concepts in the ontology. The next step involves organizing concepts into hierarchical taxonomy. For example, *customer assigned priority* and *structure based priority* are subclasses of the class *priority*. i.e. *customer assigned priority* is a type of priority. Therefore, customer priority is a subconcept of the *priority* concept.

3. **Defining relations between concepts** Concepts and relations between the concepts are shown in Figure 5.6. For each concept, conditions are

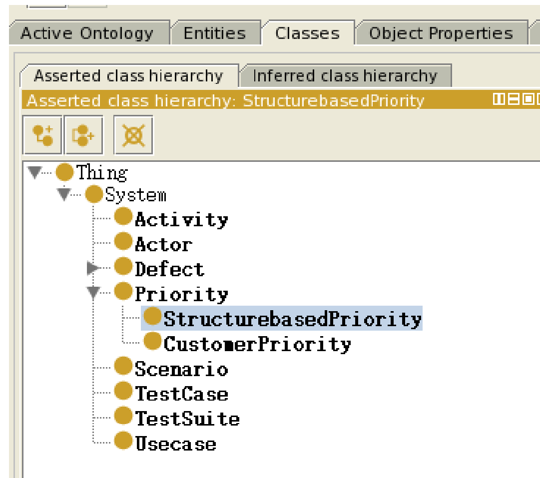


Figure 5.5: Creating a concept hierarchy

defined as to how the concepts interact for realizing an objective.

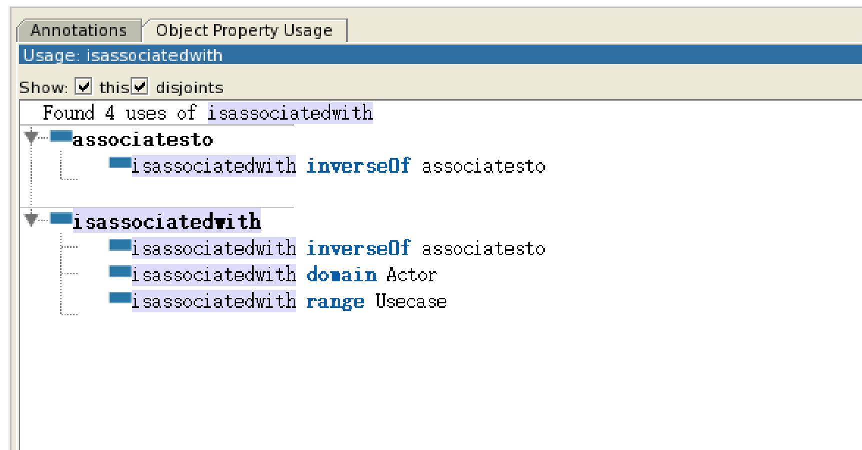


Figure 5.6: Class diagram showing relation among concepts of the ontology

4. **Defining properties and constraints :** Concepts themselves are incapable of answering questions such as those enumerated in Step 1. There is need to describe internal structure of concepts. For example, consider the concept *use case*. Properties related to the concept includes *haspriority*, *hasscenario*, *hasactor*. For each property, the concept it describes must be determined. The properties are attached to concepts.

5. **Creating instances :** Instances are individuals belonging to a concept.

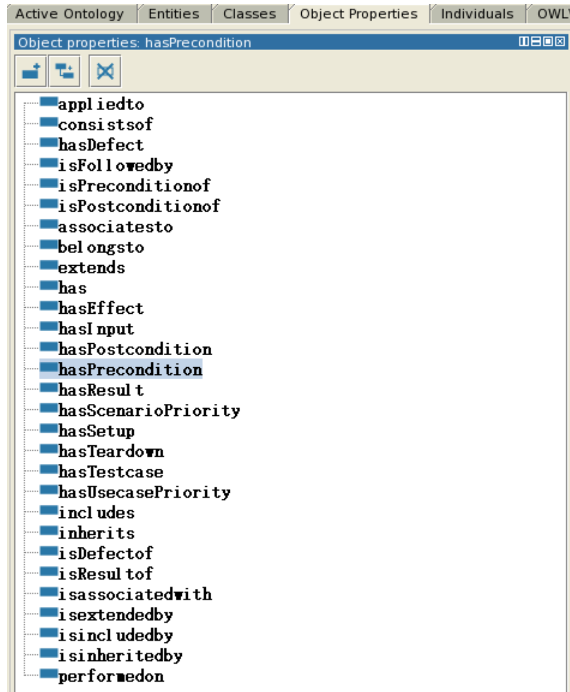


Figure 5.7: Properties of concepts

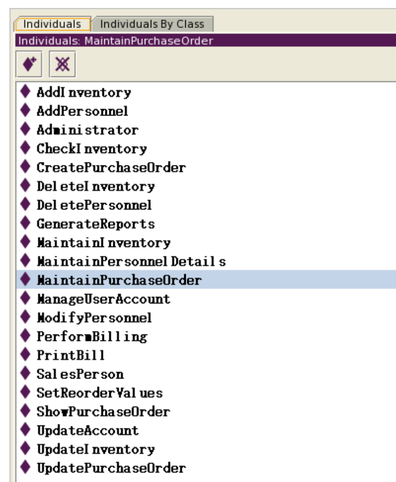


Figure 5.8: Instances belong to concepts

5.4 Querying the Ontology

As shown in Figure 5.10 the Protege OWL editor has support to edit and execute rules through the Query window using Pellet/FACT++ as reasoner. A user expresses requirements as a query and submits it to the Protege tool to obtain results. Queries include determining the number of tests that passed based on some criteria, tests based on a criteria that failed as well as displaying results. A sample of the queries that can be given to the ontology include:

a. What are the use cases the include use case 'x' ?

b. List all actors related to use cases and scenarios having priority 'y'.

c. List all scenarios for use case 'uc'. (This lists all scenarios for use case 'uc' as well as for use cases that are included by use case 'uc').

d. List all scenarios used by an actor.

Three examples querying the ontology is shown. In the first example(Figure 5.9 & Figure 5.10), use cases, 'PerformBilling', 'CreatePurchaseOrder' and 'MaintainInventory' include use case 'CheckInventory'.

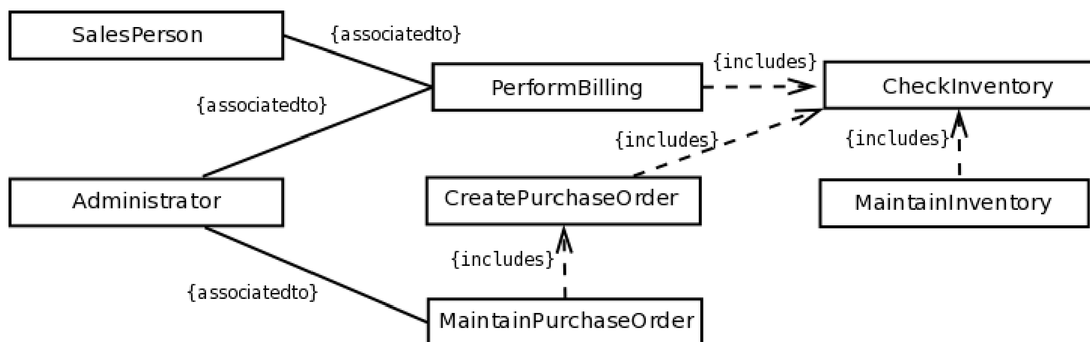


Figure 5.9: Querying the ontology - an example

The constraint applied on relation 'include' is that it is transitive. Hence, the query 'Usecase and includes value CheckInventory' gives the above three use

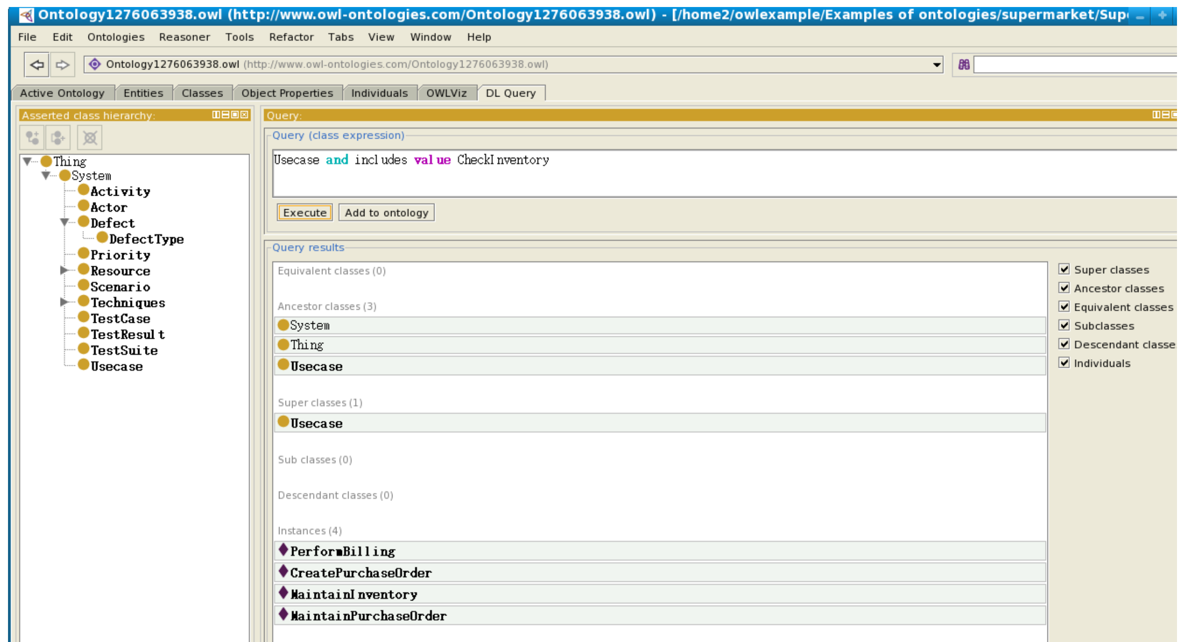


Figure 5.10: Querying an ontology - List all use cases that include a particular use case

cases as well as the use case 'MaintainPurchaseOrder' which includes use case 'CreatePurchaseOrder'. This inference is useful in testing the use case 'Maintain Purchase Order' where the scenarios belonging to 'Check Inventory' also have to be tested.

A second example requires that all actors related to both use cases 'Perform-Billing' and 'Manage User Account' be given. Fig. 5.11 show the results of the query.

A third example requires that all actors associated to 'PerformBilling' be listed. Fig. 5.12 show the results of the query.

5.5 Extension of Ontology

For test management, not only domain, but also the process of testing should be specified comprehensively so that stakeholders in testing can participate effectively. Besides the concepts obtained as shown in Figure 5.2, testing concepts outlined in

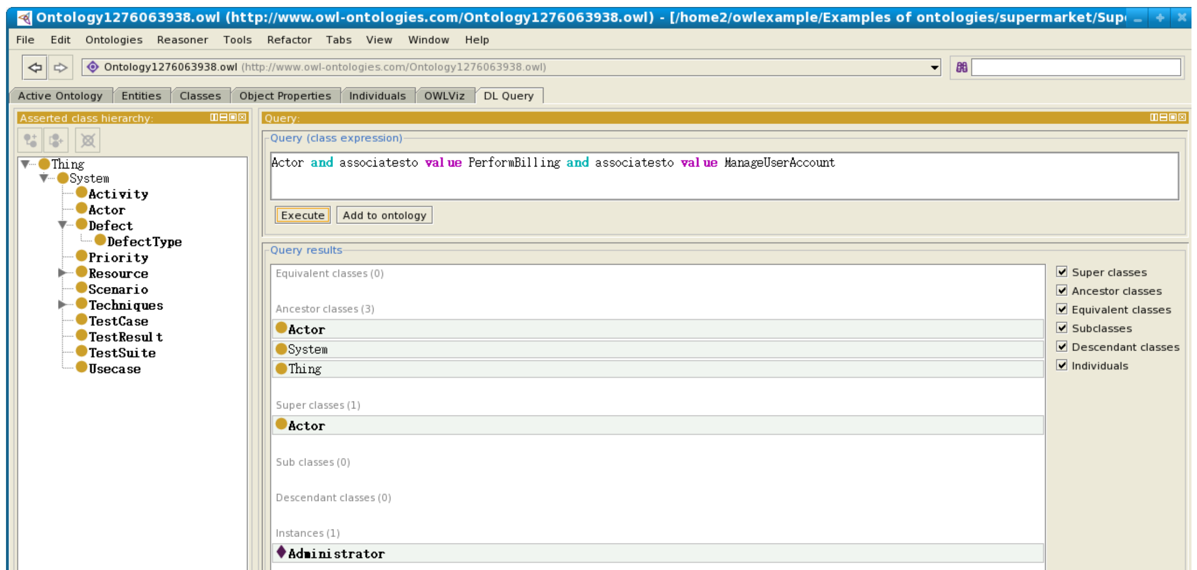


Figure 5.11: Querying an ontology - List all actors related to use cases 'Perform-Billing' and 'Manage User Account'

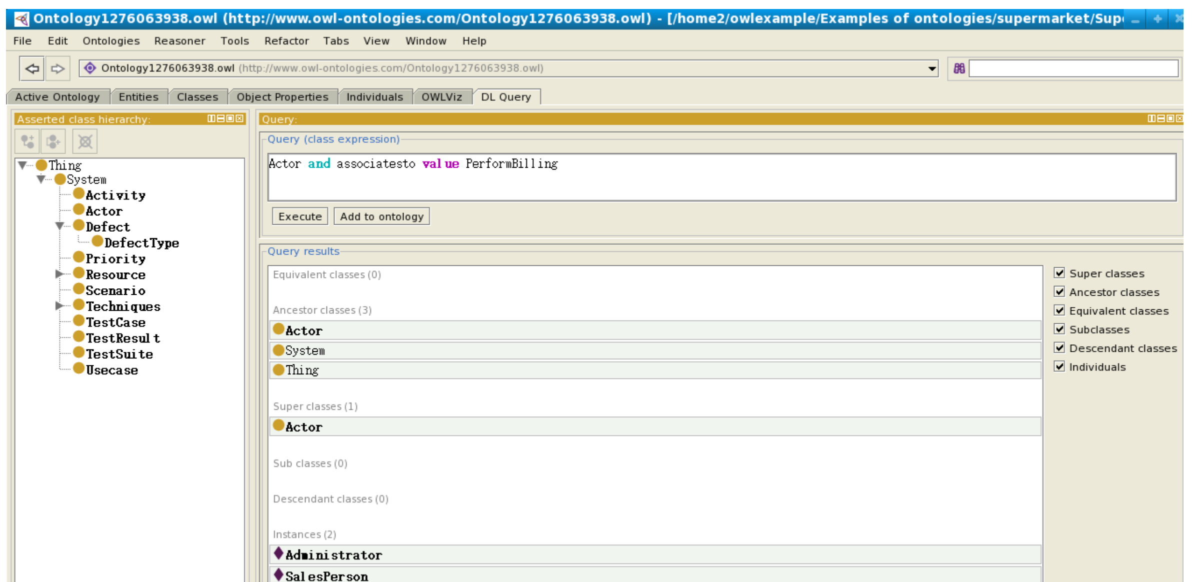


Figure 5.12: Querying an ontology - List all actors associated to a particular use case

the SWEBOK are considered. Concepts obtained from SWEBOK include TestPlan, Result, TestType, UnitTest, RegressionTest, SystemTest, IntegrationTest, Resource, Hardware, Software, People, Time, DefectType, TestSuite, TestResult and TestData. The concepts applied to build an ontology for testing is shown in Figure 5.13. The advantage of the ontology is in managing the process of testing by querying, reasoning and inferring on the concepts.

A few examples of queries possible on the ontology are:

- What are the resources required in a particular TestPlan ?
- List all classes related to testing a scenarios.
- What are the scenarios involved in RegressionTest ?
- Give the scenarios selected using a particular testing technique.
- What is the people resource required for a particular testplan ?

5.6 Summary

Scenario management involves selecting a set of scenarios for testing in order to meet project management criteria. Knowledge about entities in the domain and interactions between them provide significant input for scenario selection. This chapter presented an approach to query knowledge captured related to a domain for the purpose of selecting scenarios for testing. To effectively carry out the querying an ontology of software testing using related UML artifacts are built. Actors, related functionalities(use cases), scenarios, activities, priority of use cases and scenario have been considered. Protege, an OWL editor is used for querying and reasoning as shown in Section 5.4. Ontologies also help different users of a system like architects, developers and test engineers have a common reference point. For this, concepts from SWEBOK have been adopted and added to the

Figure 5.13: Class diagram showing relation among concepts of the ontology

ontology. Thus, an ontology aids in better understanding of requirements and goals of the system.

Chapter 6

Results and Prototype Tool

A prototype tool that implements the concepts discussed in the previous chapters was built and the concepts were tested with case studies. The tool consists of automated scenario generation from UML diagrams, prioritization and selection of scenarios. The results obtained using the tool on the case studies are presented in this chapter.

6.1 Introduction

The ideas presented in Chapter 4, Sections 4.2, 4.3 and 4.4 are tested on case studies and results are presented in this chapter. The experiments attempt to answer the following:

- Are all scenarios preferred by the customer generated using the scenario generation algorithm proposed in the work(ScenGen, Priority-Based-ScenGen and Level-Based-ScenGen in Chapter 4, section 4.2)
- Compare performance of scenario prioritization obtained due to combination of weights assigned by the user and calculated using structure based complexity to optimal, coverage based and random prioritization.

- Performance of selection techniques taking into consideration the percentage of scenarios selected vs. percentage of defects detected.

A testing tool called TestGen (Test Generation) was built to support the techniques presented in Chapter 4. The tool implements techniques to generate scenarios from UML activity diagrams, as well as techniques for test scenario prioritization and selection.

In this chapter, a prototype of a testing tool and results produced is presented. Section 6.2 gives a quick overview of the tool and the functionality of the tool is given in Section 6.3. Section 6.4 presents the case studies used to study the effectiveness of the proposed techniques. The results of applying prioritization and selection techniques is also discussed. To end, some conclusions and final remarks are made in Sections 6.5 and 6.6.

6.2 Architecture of TestGen

A tool, TestGen, was built to implement the techniques proposed in this work. The tool consists of two parts, the preprocessing part(scenario generation) and test case generation. This work focusses on the former and the latter is left for future work. The architecture of TestGen is presented in Figure 6.1. Specification of the system in the form of UML use case and activity diagrams is the input to the tool. The TestGen tool, extracts required use case and activity diagram primitives from the diagram files. First, consistency of specification is checked by the *'Consistency Checker'*. Once consistency among diagrams is ensured, the specification can be used for scenario generation. UML activity diagrams are used to generate scenarios.

The next step involves prioritization of use cases and scenarios. Customer inputs on priority of use cases is obtained. Also, structure based priority of use cases

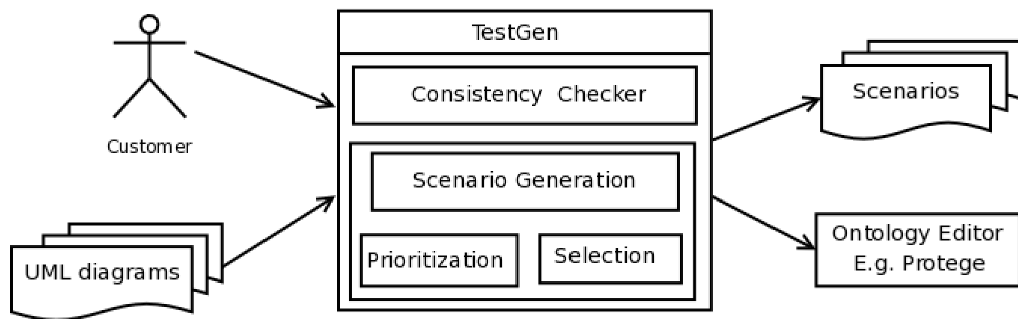


Figure 6.1: Architecture of the testing tool

and scenarios is calculated using an automated technique based on the primitives of the use case and activity diagram respectively. Scenario selection is essential when there is need to pick a subset of scenarios for testing while test engineers work with time limitation in contrast to high demand on quality. Techniques include selection based on distance measures calculated between scenarios(Levenshtein distance, Common subscenario approach, Clustering). The use case diagram of the tool shows the actors and the use cases they interact with(Figure 6.2).

6.3 Functionality supported

The TestGen tool support the following functionality :

- Extracting necessary details from UML use case and activity diagrams.
- Customer assignment of priorities for use cases.
- Automated computation of use case priorities according to primitives.
- Automated computation of scenario priority by primitives.
- Approach for selecting a subset of scenarios using distance measures(Levenshtein based, common substring based).
- Clustering based approach to scenario selection.

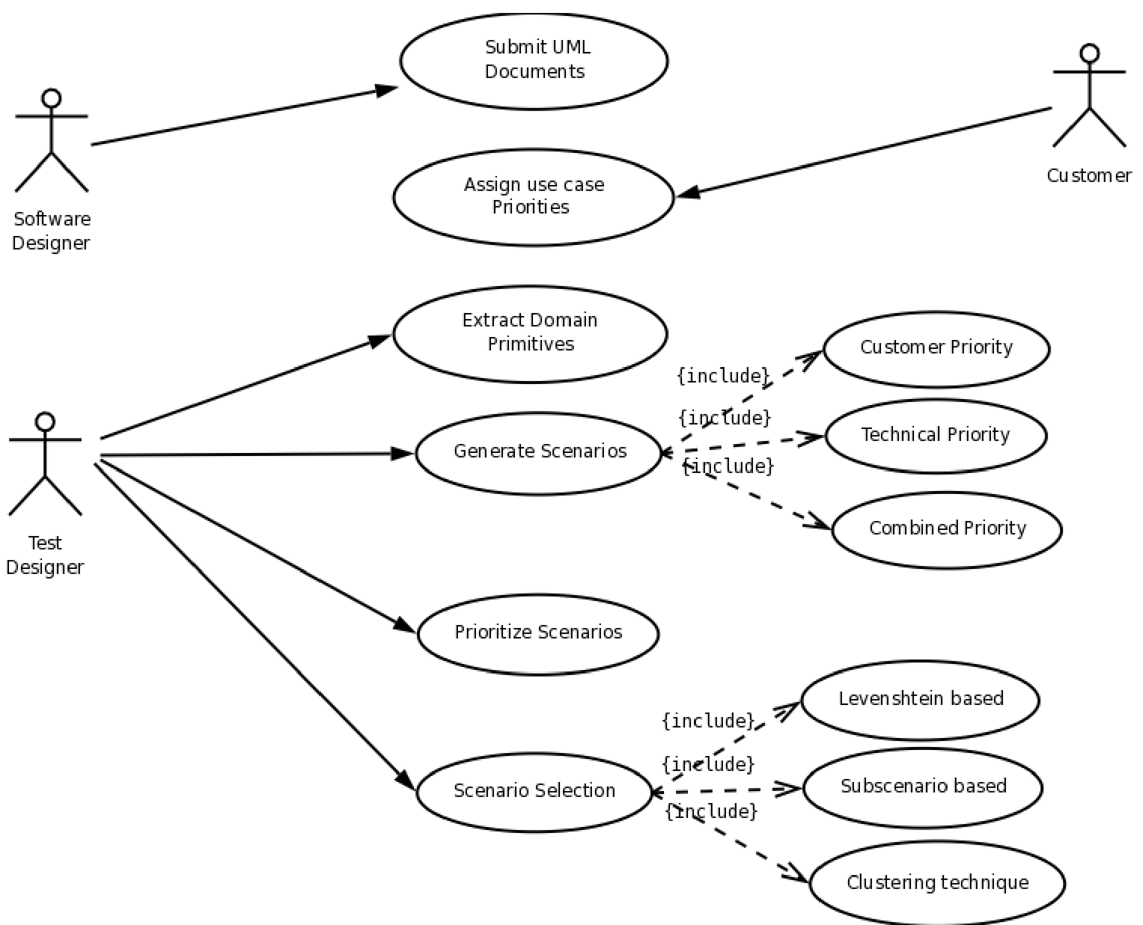


Figure 6.2: Use case diagram for the testing tool

The tool requires as input the folder containing all UML diagrams related to the Software Under Test(SUT). This folder contains the use case diagrams and activity diagrams related to each use case diagram(Figure 6.3).

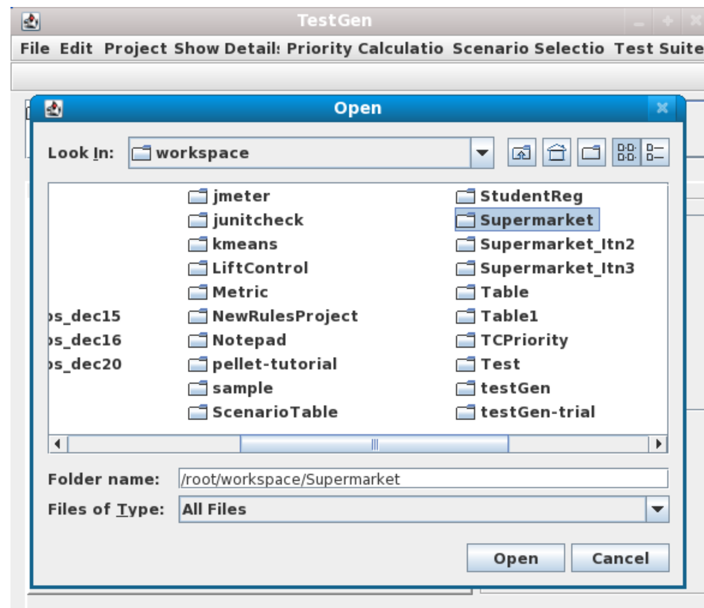


Figure 6.3: Testing tool: TestGen - Selecting the Software Under Test

The use cases are extracted from the 'UseCase' folder. Then, for each use case the corresponding activity diagrams are selected and linked(Figure 6.4).

Figure 6.5 shows a screen shot of the tool, TestGen. There are five main windows: the system window, the model window, the details window, the operations window and the scenario window.

- ***System Window.*** The system window is used to represent the actors and interacting use cases in a hierarchy(tree structure). At the first level is the root node, namely, Root. At level one, are the actors of the system. Related to each actor are the use cases they interact with. Further, inter use case relationships <include>, <extend> and <generalization> relation are shown. The objective of the system tree window is for easy understanding of the relationship between the actors and the functionality of the system.

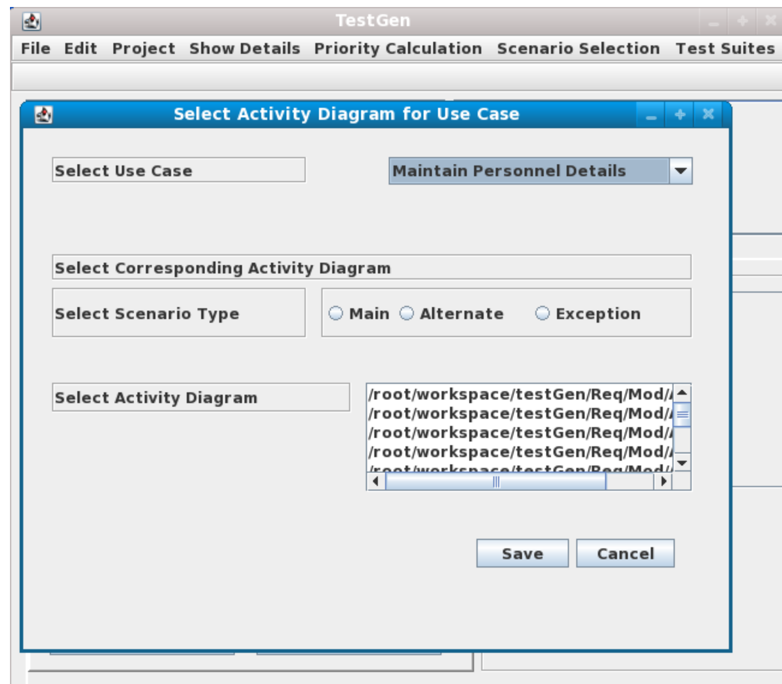


Figure 6.4: Testing tool: TestGen - Linking Activity Diagram

- ***Model Window.*** The model window shows the activity diagram(s) related to each use case.

- ***Details Window.*** On request on each item in System window, the 'Details Window' shows the information on the entity e.g. for a use case entity the details are the name of the entity, the type, related actors and priority information.

- ***Operations Window.*** Operations window gives a set of operations that can be applied on use cases. It includes, select, project, union, intersect, difference and alike. These relational operators can be applied to use cases.

- ***Scenario Window.*** Each use case consists of a set of scenarios. The scenario window lists all scenarios related to a use case.

Scenarios generated for each use case from corresponding activity diagrams are stored in .xml file.

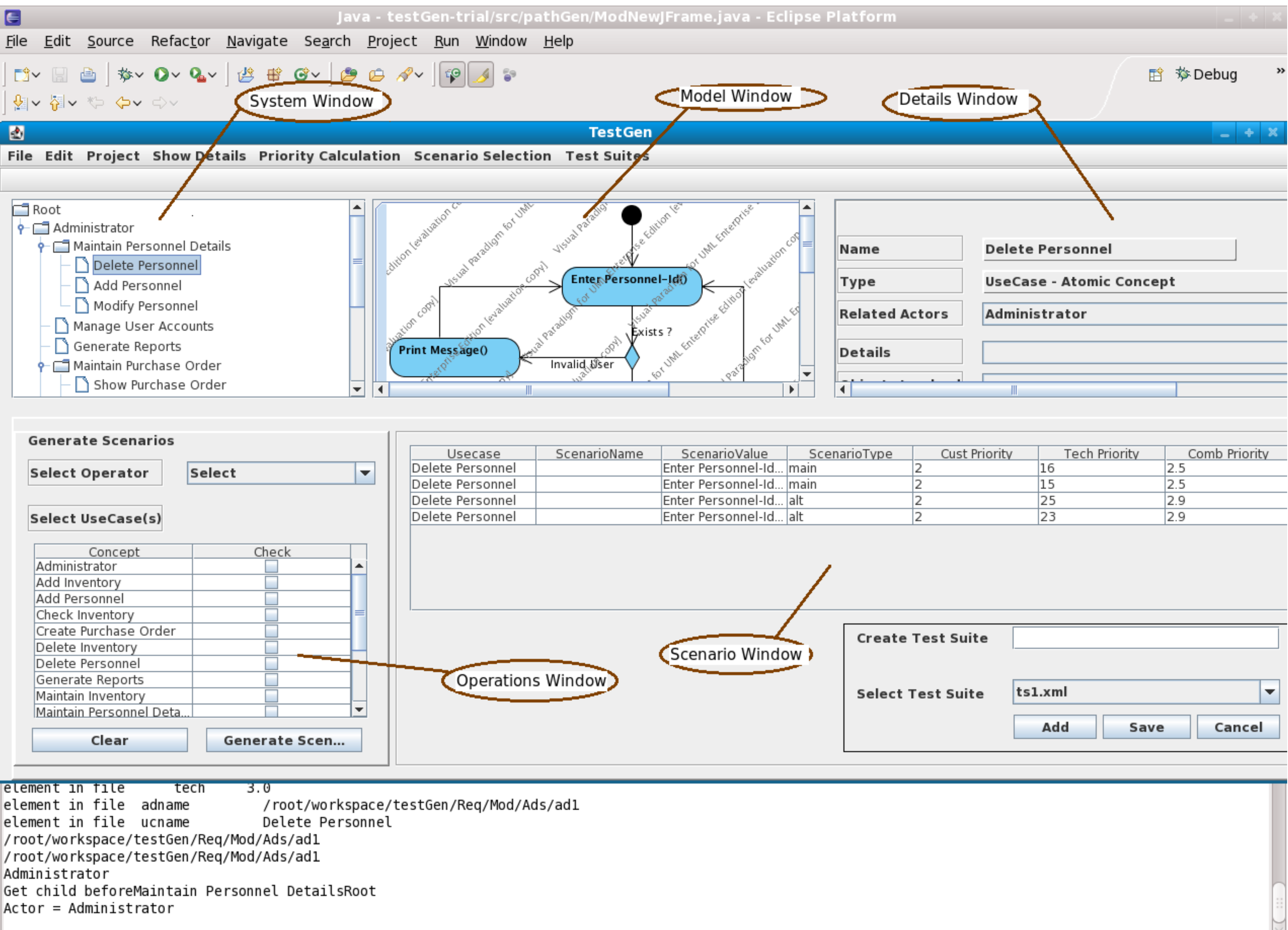


Figure 6.5: Testing tool: TestGen

6.3.1 Modules of the System

The proposed system integrates the following modules:

- ***Preprocessing.*** This module prepares data from the UML documents. Specification captured using UML use case and activity diagrams are stored in .xmi format. In the preprocessing stage, first the use case diagrams are used to extract details about the actors, use cases and the relationship between actors and use cases as well as between use cases. The output is a .xml containing actors, use cases and the relationship between them. Second, activity diagrams are used to extract activities and the relationship between activities. Thus, the output of this process is the recording of each activity with its type(e.g. activity, start/stop activity, decision, fork/join) as well as the activity interaction pairs.

- ***Scenario Generation.*** The input to this step is the .xml file containing details about activities and their relations with reference to each use case. Scenarios are generated using modified Depth First Traversal. Scenarios for each use case diagram is generated from the corresponding activity diagrams as discussed in Chapter 4, Section 4.2.5.

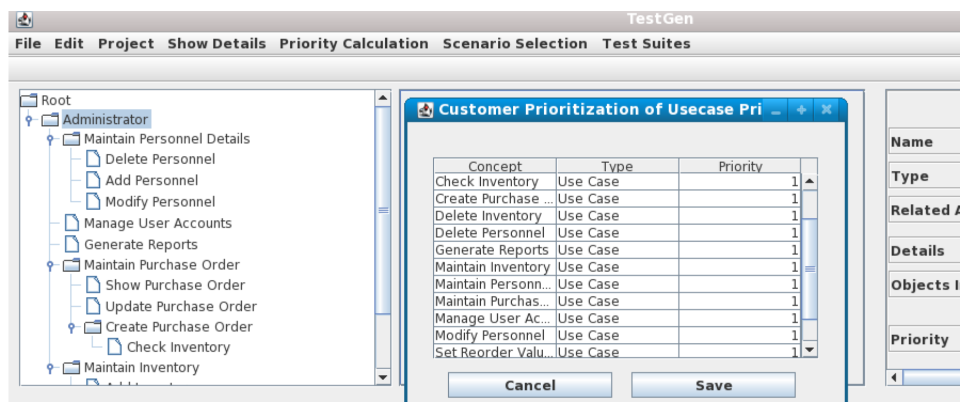


Figure 6.6: Obtaining priority from the Customer(default value assigned to 1, lowest priority)

- ***Prioritization.*** Priority of each scenario is calculated thus:

  1. Customer inputs on priority is obtained for each use case. The customer gives priority to use cases on a scale of 1-10. Example is shown in Figure 6.6.

  2. Structure based priority is calculated using the primitives of the use case and activity diagram. The results of structure based priority is shown in Figure 6.7. The details on calculation of structure based priority is already discussed in Chapter 4, Section 4.3.8.

  3. Combined priority of use cases and activities is used to prioritize scenarios belonging to a test suite. The relative weights for consideration of customer and structure based priority are given by the test engineer and the combined priority calculated.

- ***Selection.*** Involves selecting a subset of scenarios to form a test suite. Dissimilar scenarios are selected based on distance measures to form a representative set of scenarios using one of three techniques as mentioned in Section 4.4.

- ***Ontology Generation.*** The information captured about the domain, namely, actors, use cases, scenarios and activities, are used to generate an ontology. The .owl file generated can be used with an ontology editor like Protege(discussed in Chapter 5).

6.4 Case Study

In this section, the results obtained on case studies used to validate the proposed methodology empirically are discussed. Two case studies have been considered, namely, the CoffeeMaker system, and a Supermarket Automation System(SAS).

Computing Technical Priority

Usecase Data

Usecase	Actor-UC inter...	UC appearanc...	UC-UC includ...	UC-UC include...	UC-UC extend...	UC-UC extend...	UC-UC inherit...
Add Inventory	0	1	0	1	0	0	0
Add Personnel	0	1	0	1	0	0	0
Check Invent...	0	2	0	2	0	0	0
Create Purch...	0	1	1	1	0	0	0
Delete Invent...	0	1	0	1	0	0	0
Delete Perso...	0	1	0	1	0	0	0
Generate Re...	1	0	0	0	0	0	0
Maintain Inve...	1	0	4	0	0	0	0
Maintain Pers...	1	0	3	0	0	0	0
Maintain Purc...	1	0	3	0	0	0	0
Manage User...	1	0	0	0	0	0	0
Modify Perso...	0	1	0	1	0	0	0
Set Reorder ...	0	1	0	1	0	0	0
Show Purcha...	0	1	0	1	0	0	0

Assign Weights [Scale 0-1, Sum(weights) = 1]

Number	Interaction Type	Description	Weight
I	Actor-UC interaction	Number of actors usecase int...	0.2
II	UC appearance	Number of times use case ap...	0.2
III	UC-UC includes	Number of use cases this use...	0.1
IV	UC-UC included	Number of use cases include...	0.1
V	UC-UC extends	Number of use cases this use...	0.1
VI	UC-UC extended	Number of use cases extend...	0.1
VII	UC-UC inheritance	Number of use cases inherite...	0.1

Cancel | Save Wt

Actor Data

Name	Type	Actor-UC Inter
Administrator	actors	5

Assign weights[Scale 0-1, Sum(weights) = 1]

Weight - Customer Priority 0.5

Weight - Technical Priority 0.5

Ok

Computed Priority

Primitive	Type	Priority
Check Inventory	Use Case	9
Create Purchase O...	Use Case	5.83
Delete Inventory	Use Case	4.5
Delete Personnel	Use Case	4.5
Generate Reports	Use Case	5.67
Maintain Inventory	Use Case	8
Maintain Personnel...	Use Case	6.67

Enter Filename comp-priority | Save

Figure 6.7: Computing Structure based Priority

The goal of the case studies is to present results obtained using test generation, prioritization and selection techniques proposed in this work.

Faults

Faults detected during testing are associated with either real defects or defects injected manually into the software [MCHI97]. In case of manually injected faults, care was taken to try to simulate as closely as possible typical programming errors. Table 6.1 shows a sample of faults, with a short description for each of them.

Table 6.1: Sample of Faults

Fault	Type	Description
F1	Injected	Error in reading file
F2	Injected	Error in writing file
F3	Real	Incorrect path
F4	Injected	Missing Functionality
F5	Real	Missing error handling
F6	Injected	Wrong expresssion
F7	Injected	Wrong condition
F8	Injected	Interface specification
F9	Injected	Wrong algorithm
F10	Real	Incorrect Message

6.4.1 Results of the Case Study

This section describes the experimental results obtained when the prototype tool, TestGen, was used to test two case studies, namely, CoffeeMaker and the Supermarket Automation System. First, scenarios were generated. Then customer inputs on priority of use cases was obtained. Further, priority values based on structural primitives were calculated. The combined priority of the use case and scenario is the priority of the scenario. The ordered scenarios were used to obtain results of testing. Test selection was done using the three techniques and the selected scenarios were executed. Testing results and evaluation of the same is presented in this section.

Test Generation

Objective : Each use case has at least one activity diagram elaborating it. Scenarios need to be generated for each use case. A collection of such scenarios form the entire set of scenarios related to the system.

Input and Output : The input to the test generation process is activity diagrams related to each use case diagram stored as .xmi files. The output is scenarios related to each use case.

Strategy followed : First, for each use case, related activity diagrams were designed. In case of concurrent activities, where possible, level/priority details were annotated. A modified DFS algorithm discussed in Chapter 4 is used to generate scenarios.

Checking Test Adequacy:

Test adequacy criterion specifies the requirements of a particular test. To measure the quality of test scenarios produced, the following test adequacy criterion[CMK08] is used:

- Activity Coverage requires that all the activities in the activity diagram be covered. Activity coverage is calculated as the ratio between the covered activities and all the activities in the activity diagram.
- Transition Coverage requires that all the transitions in the activity diagram be covered. Transition coverage is calculated as the ratio between the checked transitions and all the transitions in the activity diagram.
- Path Coverage requires that all the paths in the activity diagram be covered. Path coverage is the ratio between the traversed paths and all the paths in the activity diagram.

Prioritization

Objective : Given a collection of scenarios, there is need for prioritization to widen coverage and detect defects early.

Input and Output : The input to the prioritization process is scenarios generated from all activity diagrams.

Strategy followed : Customer priority of requirements(here use cases) was obtained. The customer ranked use cases on a scale of 1-10. Further, priority based on structural primitives of use case and activity diagram was calculated (Figure 6.7). A combined priority for each use case was calculated based on weights given to customer priority and structure based priority. The prioritized list of scenarios thus obtained was tested on the system with each scenario having a minimum of at least one test case.

Evaluation Measure:

Results of prioritization are compared against optimal, coverage and random ordering to study effectiveness using the APFD metric discussed in Chapter 2, Section 2.6.2.

Selection

Objective : Given a collection of scenarios, a subset of scenarios need to be selected for testing to meet constraints of cost and time.

Input and Output : The input to the selection process is scenarios generated from all activity diagrams.

Strategy followed : Scenario selection involves selecting a subset of scenarios for testing. Results are compared against random selection. In this work, picking one of two scenarios with minimum distance is done based on one of the following: priority(structure based and combined priority), type and random.

Evaluation Measure:

The Average Percentage of Faults Detected(APFD) metric discussed in Section 2.6.2, is used for evaluating performance of the technique.

6.4.1.1 Case Study - I: CoffeeMaker

The first case study is a CoffeeMaker example. The requirements of the system is as follows: The coffee maker remains in wait state when not in use. There are a total of six options:

1. Add a Recipe
2. Delete a Recipe
3. Edit a Recipe
4. Add Inventory
5. Check Inventory
6. Purchase Beverage

A maximum of three recipes, each being unique, can be added to the CoffeeMaker. A recipe consists of a name, price, units of coffee, units of milk, units of sugar and units of chocolate. The price of an item is an integer. Recipes may be added, edited and deleted. Each recipe being unique, it is not possible to add a duplicate recipe. Also, it is possible to delete only an existing recipe from the list of recipes. The recipe to be deleted is chosen by name. A recipe can be edited by changing the price, units of milk, sugar, coffee or chocolate. Again, the recipe to be edited is chosen by name. After completion of the action, the CoffeeMaker returns to waiting state.

Inventory can be added to the system. Inventory includes units of coffee, milk, sugar and chocolate, units of measurement being integers. On addition of inventory, message is printed and the CoffeeMaker returns to waiting state.

To buy a beverage, a user needs to deposit enough change and select the item required. In case the change is insufficient, the user is prompted for more change. The user may enter change, or select the cancel button, in which case, the change is returned. In case of enough change and the inventory being insufficient, the CoffeeMaker prints a message and prompts for selection of another item. Cancellation of the order returns the change. In case of sufficient change and sufficient inventory, coffee is prepared and dispersed. Also, change is returned to the user, if any. The use case diagram of the CoffeeMaker is shown in Figure 6.8. The coffeemaker case study consists of 900 lines of code.

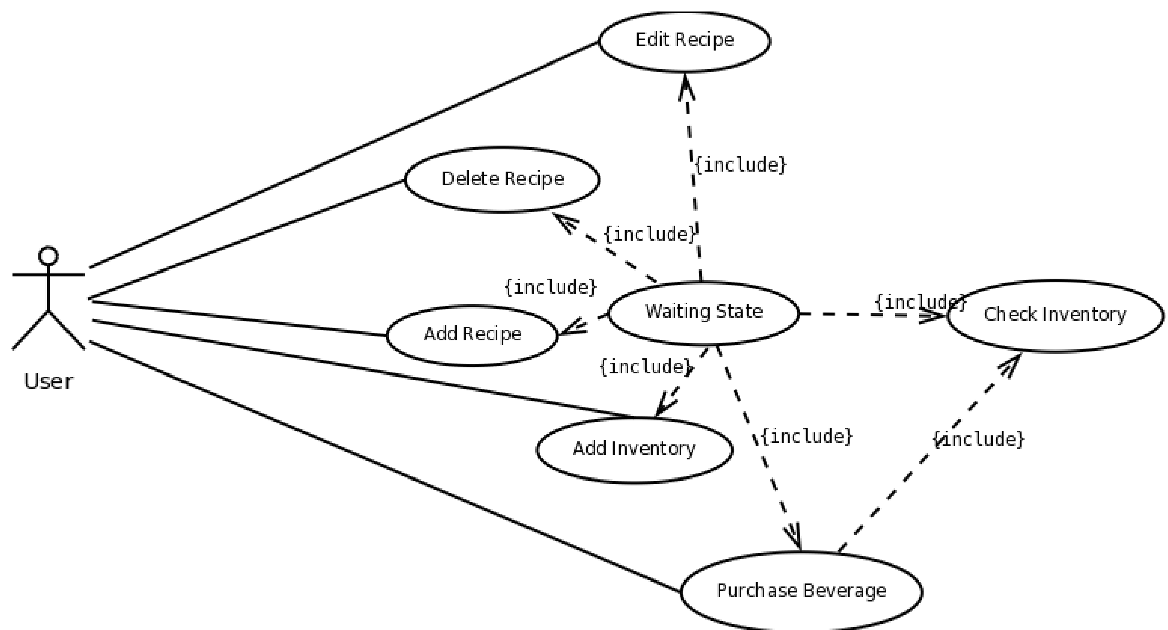


Figure 6.8: Use case diagram for the 'CoffeeMaker'

Results

Scenario Generation

The UML specification consists of 6 use cases. Each use case has one activity diagram related to it. The system consists of a total of 50 activities and 73 transitions. A total of 23 scenarios were generated for the CoffeeMaker system. All scenarios preferred by the customer was obtained.

Prioritization

The use cases were prioritized taking a weighted sum of customer as well as structure based priority. Equal weight has been given for both customer and structure based priority in this work. The results of prioritization applied to the CoffeeMaker case study were calculated using the metric, APFD, discussed previously in Chapter 2, Section 2.6.2.

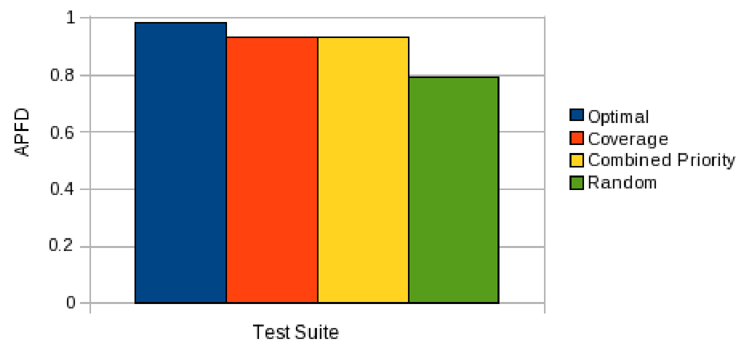


Figure 6.9: Comparison of APFD values obtained for CoffeeMaker System

Figure 6.9 shows the APFD values obtained for the technique, compared to optimal, statement coverage and random prioritization. The same results are summarized in Table 6.2. The results of the technique proposed in this work using customer inputs as well as structural inputs of both use case and activity diagrams are equal to coverage based prioritization. Random prioritization performs substantially worse with APFD value being 10% lower. The automated technique is thus an effective technique in prioritizing scenarios. Also, coverage information

Table 6.2: Results in terms of APFD values for CoffeeMaker System

Prioritization Technique	APFD
Optimal	.98
Combined Priority	.93
Coverage	.93
Random	.79

is not available in the case of a new set of scenarios to be tested. Thus, besides requiring minimal effort, the automated prioritization technique can be used along with other prioritization techniques to aid prioritization of scenarios obtained from UML use case and activity diagrams.

Scenario Selection

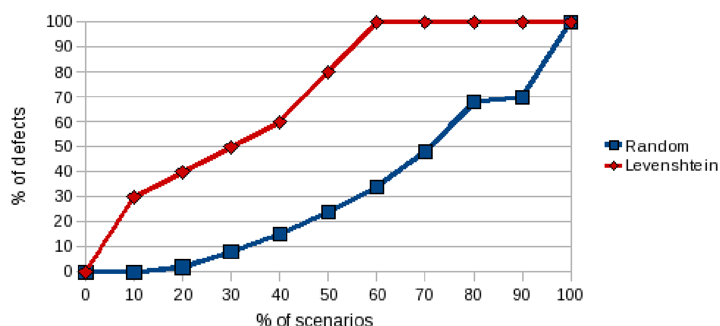
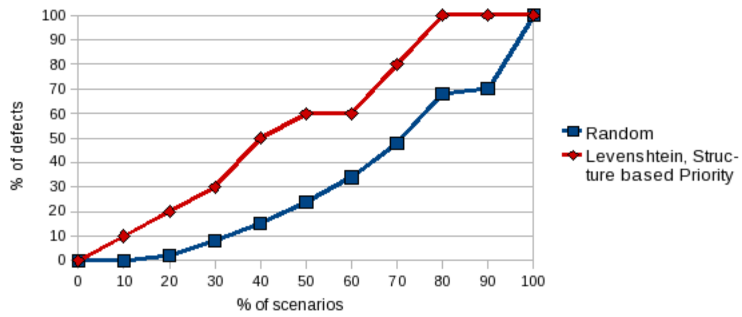
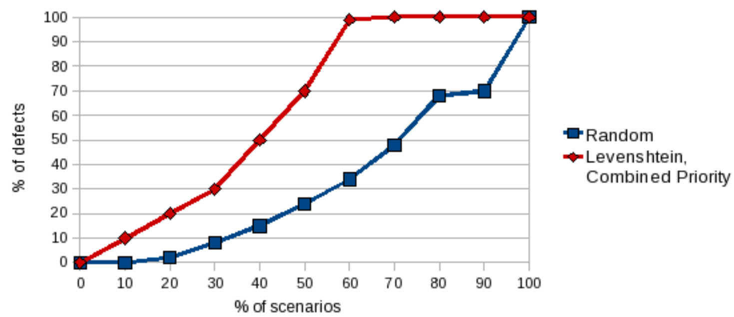


Figure 6.10: Defect detection using Levenshtein distance as distance measure for selection

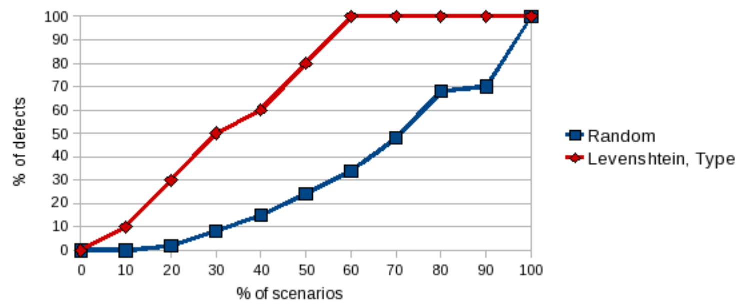
In the first case, levenshtein distance was used as a distance measure to determine similarity among scenarios. The curve shows the relations between percentage of scenarios tested to percentage of defects detected. The upper curve shows the result for Levenshtein based technique as described in Section 4.4.4. The lower curve shows the result for random selection of scenarios. The aim is to detect as many defects early and by testing less number of scenarios. Figure 6.10 shows the results of scenario selection using levenshtein distance. In case of Levenshtein based selection, 60% of scenarios selected give 100% defect detection when com-



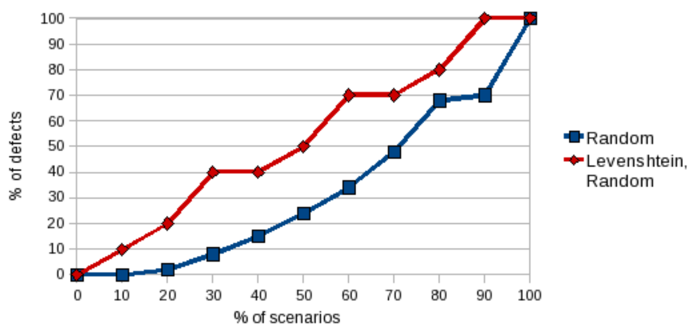
(a) Levenshtein, Structure based Priority Vs Random



(b) Levenshtein, Combined Priority Vs Random



(c) Levenshtein, Type Vs Random



(d) Levenshtein, Random Vs Random

Figure 6.11: Defect detection using Levenshtein distance as distance metric for selection-applying different methods for picking one of two scenarios

pared to random selection which requires 100% testing to detect all defects. The percentage of Levenshtein distance spaced defect discovery is persistently more compared to random. Further, it is noted that 60% of scenario testing captures 100% defects indicating Levenshtein distance is powerful in discovering defects. Hence, Levenshtein based selection is recommended for selection of scenarios.

Different techniques are used to pick one of the two similar scenarios (computed with Levenshtein distance), like, structure based priority, combined priority, type of scenario and random selection as shown in Figure 6.11. The results of the techniques are compared with random selection. Results show that structure based priority, combined priority and type data may be used effectively to better results of selection. Random picking of one between two scenarios may not always give best results.

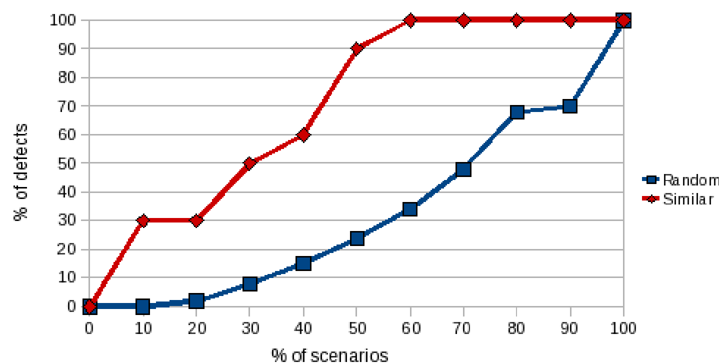
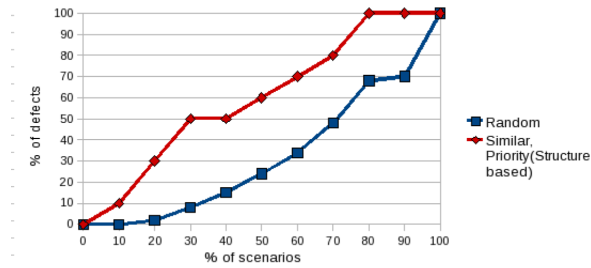
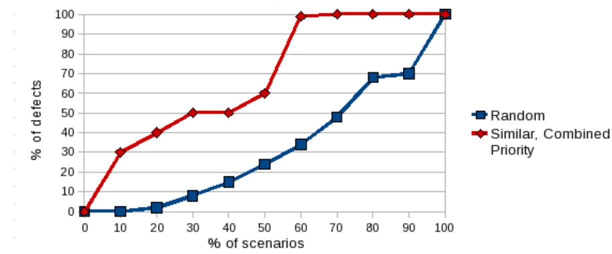


Figure 6.12: Defect detection using similarity measure based on common subscenarios as distance measure for selection

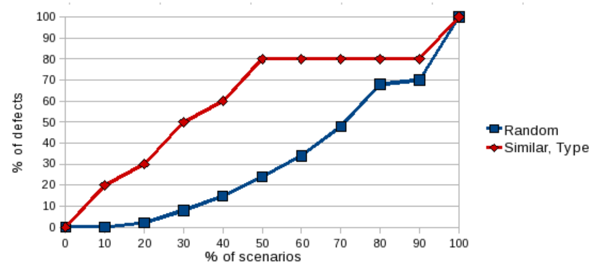
In the second case, metric based on common subscenarios was used to determine similarity among scenarios. Figure 6.12 shows the results of scenario selection. Upper curve shows the rate of defect detection in case of using the similarity measure whereas the lower curve shows the result of random selection. With increasing percentage of scenarios, the number of defects detected increases. 60% of scenarios selected using the similarity measure detects 100% defects. Here also, the similarity measure spaced defect discovery is persistently more compared



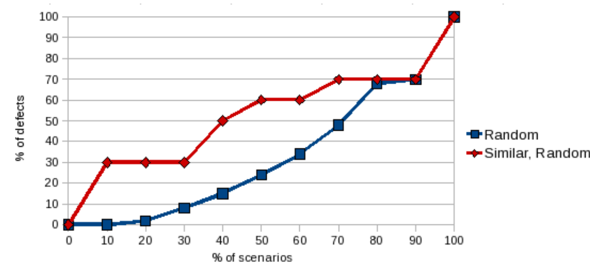
(a) Levenshtein, Structure based Priority Vs Random



(b) Levenshtein, Combined Priority Vs Random



(c) Levenshtein, Type Vs Random



(d) Levenshtein, Random Vs Random

Figure 6.13: Selecting one of two similar scenario based on similarity measure

to random selection of scenarios.

Again, different techniques have been used to pick one of the two similar scenarios. Results show the difference in defect detection rate of the technique for picking one of two scenarios between priority based, type and random. It is observed that random selection performs worse than other techniques(Figure ??).

The third technique used for selection of scenarios is clustering. Here, percentage of scenarios are plotted in the x-axis against the percentage of defects detected in the y-axis. One scenario is selected as representative of the cluster randomly. Figure 6.14 shows the results of selecting scenarios for testing. Clustering based selection is better than random selection as shown.

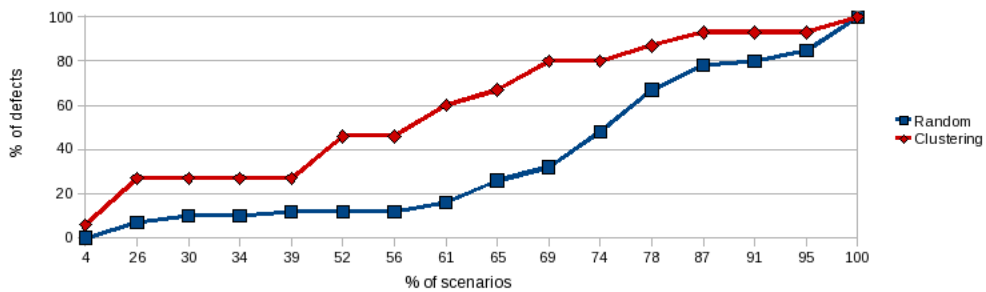


Figure 6.14: Clustering based selection of scenarios

6.4.1.2 Case Study - II: Supermarket Automation System

The second case study considered is a Supermarket Automation System(SAS). SAS deals with the automation of several activities involved in the management of a supermarket, ranging from business to market activities. Areas of automation include: billing, product purchase, personnel management, inventory management, feedback collection and report generation. SAS consists of 19 use cases, with a minimum of one activity diagram for each use case.

There are two types of users, namely, the administrator and the sales people. The administrator manages staff details, creates user access for sales people, keeps

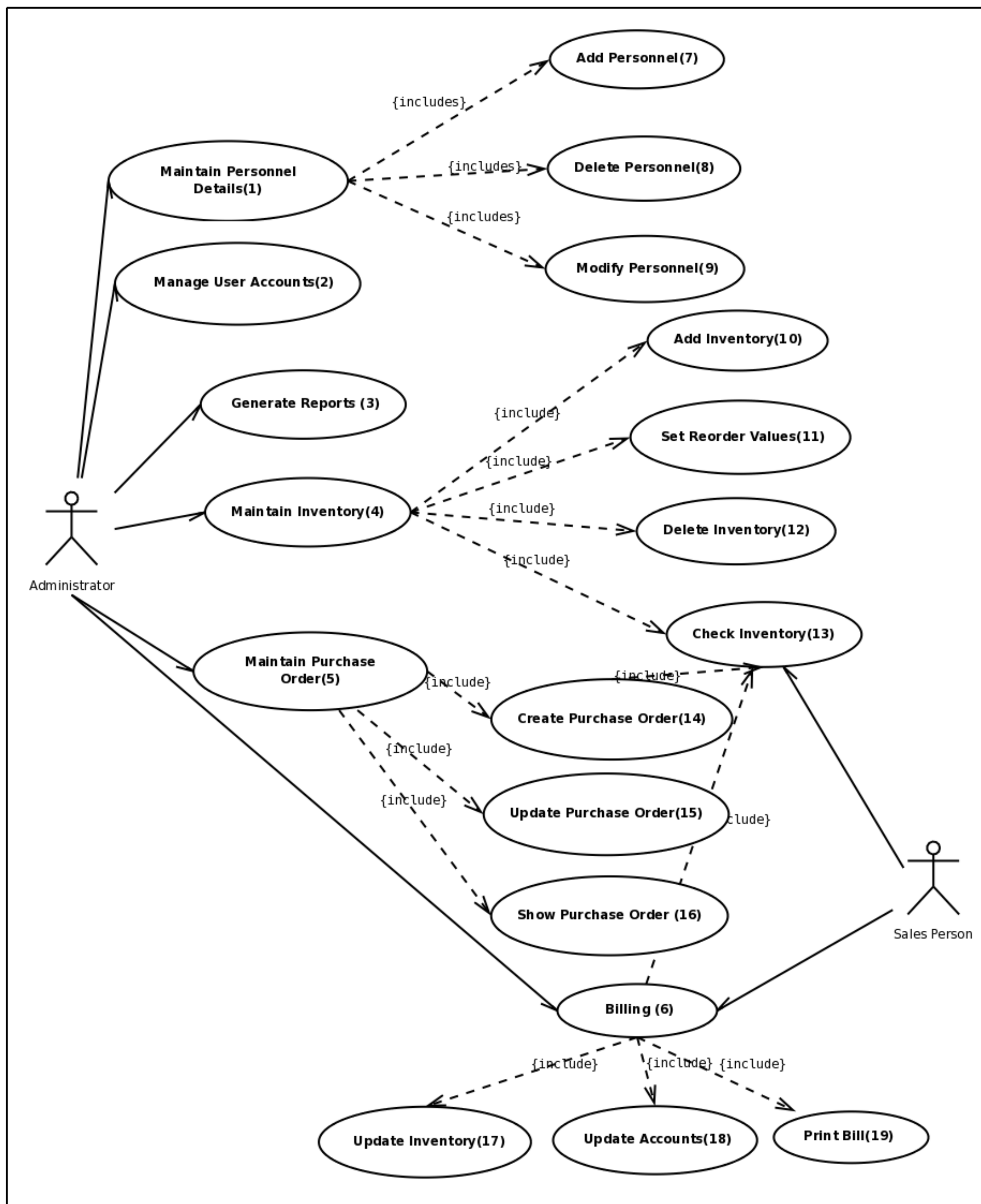


Figure 6.15: Use case diagram for 'Supermarket Automation System(SAS)'

track of inventory as well as generate various reports. The sales persons can only perform the action 'Billing'. The functionality of the system include: adding, editing and deleting products and suppliers, setting reorder limits, intimation in case a product reaches the reorder point, generating purchase orders, managing user accounts, maintaining personnel details, perform billing of products and generating various reports. The use case diagram(Figure 6.15) shows the requirements of the system.

6.4.1.3 Results

Scenario generation

Using the test generation algorithm, a total of 102 scenarios were obtained. The case study consists of 10 classes and 114 activities. Each test scenario has a minimum of one test case. All scenarios preferred by the customer was obtained.

Prioritization

Use cases were prioritized taking a weighted sum of customer as well as structure based priority. Equal weight has been given for both customer and structure based priority in this work. The results of prioritization applied to the SAS case study were calculated using the metric, APFD.

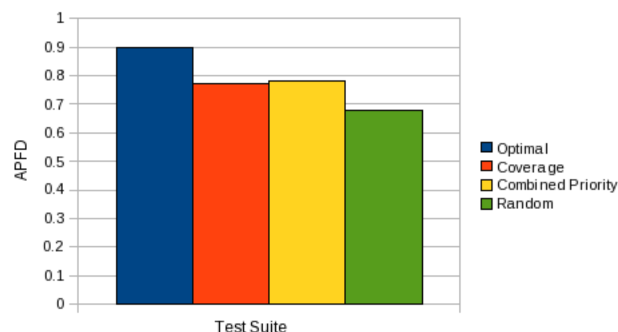


Figure 6.16: Comparison of APFD values obtained for SAS

Figure 6.16 shows the APFD values obtained for the technique, compared to optimal, statement coverage and random prioritization. The same results are

Table 6.3: Results in terms of APFD values for SAS

Prioritization Technique	APFD
Optimal	.80
Combined Priority	.78
Coverage	.77
Random	.68

summarized in Table 6.3. The results of the technique proposed in this work using customer inputs as well as structural inputs of both use case and activity diagrams are better than coverage based prioritization. Random prioritization performs substantially worse with APFD value being 10% lower. The automated technique is thus an effective technique in prioritizing scenarios. Besides requiring minimal effort, the automated prioritization technique can be used along with other prioritization techniques to aid prioritization of scenarios obtained from UML use case and activity diagrams.

<u>Scenario Selection</u>

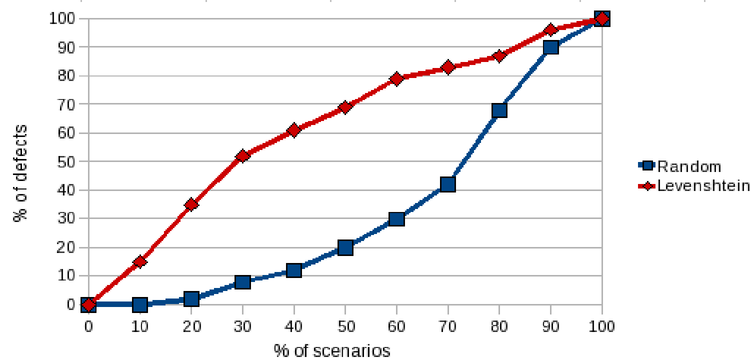
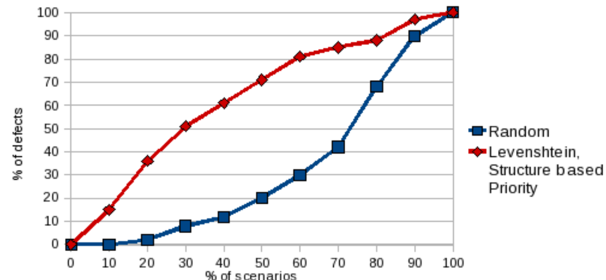
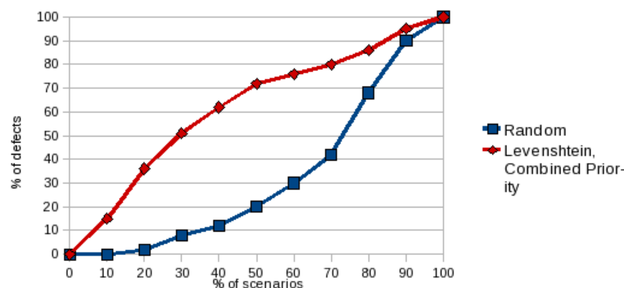


Figure 6.17: Defect detection using Levenshtein distance as distance measure for selection

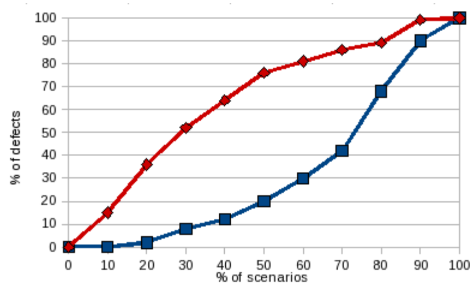
In the first technique, levenshtein distance was used as a distance measure to determine similarity among scenarios. Figure 6.17 shows the results of scenario selection using levenshtein distance. Upper curve shows the rate of defect detection in case of using the similarity metric whereas the lower curve shows the result of



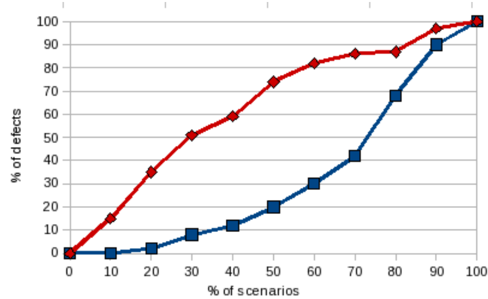
(a) Levenshtein, Structure based Priority Vs Random



(b) Levenshtein, Combined Priority Vs Random



(c) Levenshtein, Type Vs Random



(d) Levenshtein, Random Vs Random

Figure 6.18: Selecting one of two similar scenario based on Levenshtein distance

random selection. With increasing percentage of scenarios, the number of defects detected increases, with levenshtein based selection curve showing higher rate of defect detection.

Again, different techniques are used to pick one of the two similar scenarios, like, structure based priority, combined priority, type of scenario and levenshtein based random selection as shown in Figure 6.18. The results of the techniques are compared with random selection. Results show that similarity selection based on levenshtein distance as a similarity measure is better than random selection of scenarios. The defect discovery rate is persistently more when compared to random selection.

In the second case, similarity measure based on common substrings was used to determine similarity among scenarios. Figure 6.19 shows the results of scenario selection. The upper curve shows the result for similarity measure based selection whereas the lower curve shows the result for random selection of scenarios. The aim here is to detect as many defects as early as possible by testing least number of scenarios. The percentage of similarity measure based defect discovery is persistently more compared to random.

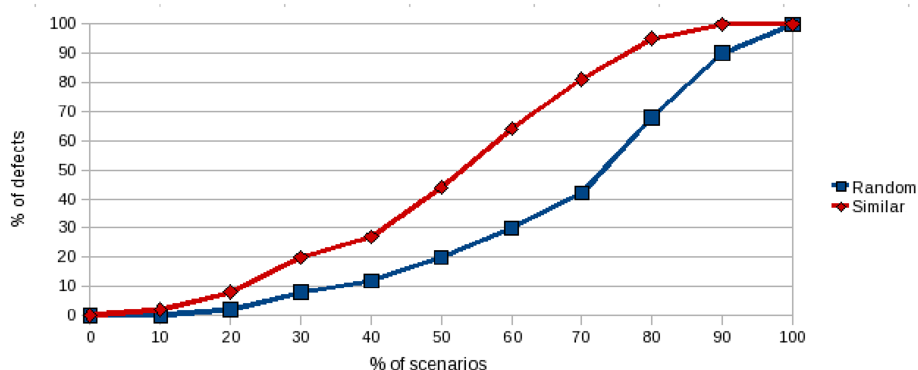
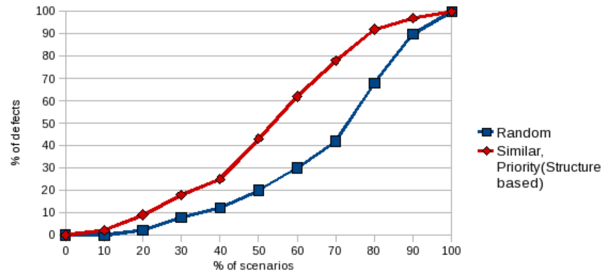
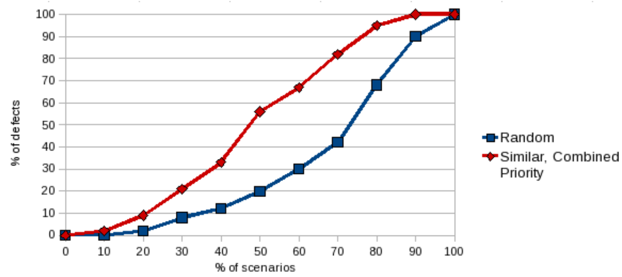


Figure 6.19: Defect detection using similarity measure based on common subscenarios as distance measure for selection

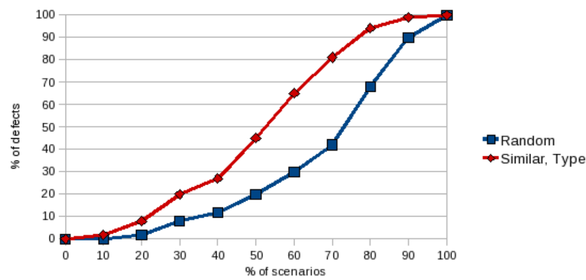
Different techniques used to pick one of two similar scenarios shows better results when based on structure based priority, combined priority and type of sce-



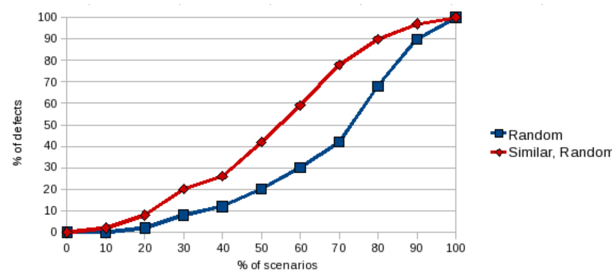
(a) Levenshtein, Structure based Priority Vs Random



(b) Levenshtein, Combined Priority Vs Random



(c) Levenshtein, Type Vs Random



(d) Levenshtein, Random Vs Random

Figure 6.20: Defect detection using similarity measure based on common subscenarios as distance measure for selection-applying different methods for picking one of two scenarios

nario as compared to random selection(Figure 6.20). This indicates the advantage of using one of the measure to pick scenarios instead of random.

The third technique used for selection of scenarios is clustering. Figure 6.21 shows the results of selecting scenarios for testing. Here, again a representative scenario is selected from each cluster. In this case, results are found to be close to random selection. This is due to the fact that selecting one scenario from a cluster is equivalent to picking a random scenario. Hence, this technique is best suited for regression testing where once an error is detected in a scenario, then similar scenarios can be determined and selected for further testing.

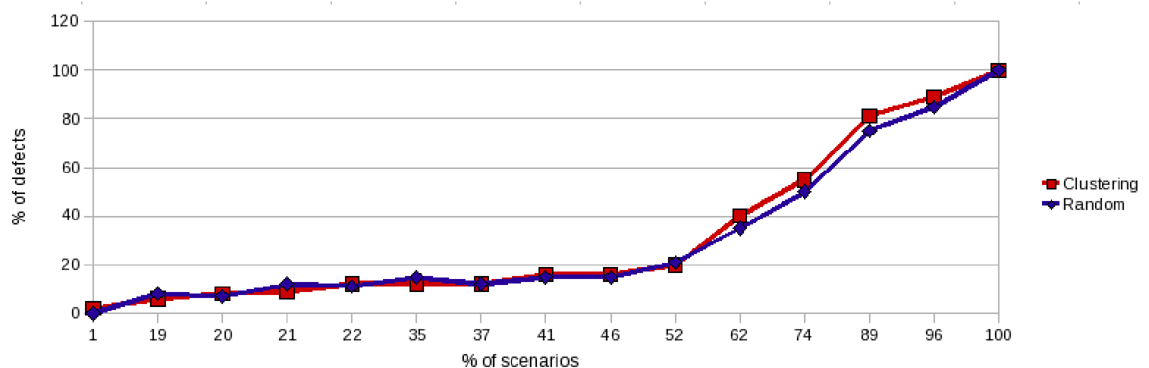


Figure 6.21: Clustering based selection of scenarios

6.4.1.4 Threats to validity

As with case studies, a number of threats affect the validity of the results. The main threats can be classified as:

- Internal validity threats, concerning factors that may have affected the experimental measures. The limited number of subjects to get input in case of customer priority, the software system chosen for the case study and the learning effect may all have influenced the results.

- Construct validity threats, concerning the relationship between theory and observation. The APFD metric used to estimate the effectiveness of different

prioritization techniques, is known to have limitations. In very special cases it may rank equally test case orderings that are not equivalent from the human point of view. However, the APFD metric has been widely used in related research and it represents a reasonable approximation of the desired measure.

- Conclusion validity threats, concerning the relationship between the treatment and the outcome. Since a case study was used, no statistical test could be applied to the results. Hence, the conclusions are supported by qualitative analysis, not by statistical tests, as always happening with case studies.

- External validity threats, concerning the generalization of the findings. Since only one data point is available, generalization of the results obtained in a case study is always hard. In addition to that, generalization to different kinds of subjects (e.g., professional programmers working in an industrial environment) and to different kinds of software (e.g., industrial systems) is difficult. The subjects had a profile which makes them not so different from professional programmers, so it is expected that some generality of the results hold.

6.5 Discussion

The first experiment, Section 6.4, indicates that the scenario generation algorithm generates all the user preferred scenarios for testing. Prioritization of scenarios using customer priority as well as priority obtained from the primitives of use case and activity diagrams aid in early fault detection. In case of scenario selection, where a subset of scenarios are selected for testing, the results show that using distance measures is an effective way of test selection. Besides, the technique

used for picking one of the two selected scenarios is effective when compared to random. Clustering scenarios is another approach to selecting scenarios. The results of the clustering approach indicate that grouping scenarios aids in selecting a representative set for testing.

The results of the second case study, showed that prioritization of scenarios based on primitives is effective for early detection of defects. Also, the technique can be used with customer inputs on priority to test based on needs of the customer. In case of selection, the techniques aid in selecting a representative set of scenarios. The parameter used to pick one of two similar scenarios aids the defect detection rate and hence, one of the techniques can be used based on information available.

In clustering based selection, the results were compared to random selection. Through the experiment, it was understood that where defects are distributed evenly in the application, clustering and selecting scenarios from each cluster for testing is similar to random testing. Hence, the results validate the performance. Clustering scenarios and testing from each cluster is expected to be productive when the defects will be from the same cluster. Such a case is expected during regression testing where the occurrence of a defect in a scenario indicates the possible occurrence of defects in similar scenarios. Here, related scenarios have to be tested. Another possibility is when the defects are localized to a requirement e.g. defects due to wrong design. In such a case, defects will be centered around that requirement, and hence a particular cluster. The results of testing based on cluster validate the same. Hence, there is need to study measures for clustering further to improve on the results obtained.

The advantage of the prioritization and selection techniques is that other than obtaining customer priority for prioritization, the techniques require no human intervention and is completely automated. Also, the technique can be used in

tandem with other techniques available in literature for prioritization and selection of scenarios to aid in effective testing of software.

6.6 Summary

This chapter presented the results of two case studies that have been conducted to study the proposed scenario generation, prioritization and selection techniques. The results obtained from the experiments indicate that the techniques aid in automated testing. The following observations were made:

- Prioritization of scenarios using customer priority as well as priority obtained from the primitives of use case and activity diagrams aid in early fault detection.
- In case of scenario selection, distance measure is effective to select a subset of scenarios for testing.
- The advantage of prioritization and selection techniques presented in this work require minimal human intervention and is completely automated.
- Besides, the techniques can be used along with other techniques for prioritization and selection.

Chapter 7

Conclusion

The overall objective of this thesis is to prescribe certain preprocessing techniques that faciliate requirement based software testing. The aim is to improve the efficacy of test scenario generation, prioritization and selection. Test scenario management using an ontology is also discussed. This chapter, presents a comprehensive summary of the work carried out in this thesis. A proposition on areas for future work concludes the chapter.

7.0.1 Summary

Given the size of software systems, there is need to automate the process of testing. Further, for reduction of test effort, certain preprocessing like test scenario generation, prioritization and selection is to be carried out. The Unified Modeling Language(UML) is being increasingly used in the industry to gather requirements and design of a system. Testing using specification (here, UML) is an important topic of research due to the advantage involved in using specification captured during the requirements gathering and analysis phase for testing. It helps by aiding different stakeholders in understanding different aspects of the system as well as helps to reduce defects in design and implementation as the same UML diagrams

can be used across the software development life cycle.

This work has contributed to research in the following aspects. First, a method for inter-model consistency checking among the use case, activity, sequence, class and state diagram is proposed. Inconsistency creeps in among the UML models as different associates are involved in different phases of the SDLC and bring in their own perspectives. These are to be viewed together to ensure that the inconsistencies do not propagate to design errors. Consistency is ensured by framing well-formedness rules and applying these on the UML diagrams that specify a software.

Second, a method for scenario generation is introduced, specifically in case of concurrent activities. The different scenarios of an application are to be analyzed in order to identify test scenarios. The order in execution of activities is represented in activity diagrams. Concurrent activities are represented using fork-join constructs. The number of scenarios to be tested in presence of a simple fork-join construct could be large. This work exploits the concept of dependency that exists between concurrent activities to aid in reducing the number the scenarios generated. Dependency is used by assigning priorities and levels at which activities are executed. The proposed method helps avoid repeating scenarios thereby reducing the cost and effort involved in testing.

Third, a method for prioritizing scenarios is introduced. The objective is to order scenarios in a manner that aims at detecting defects early and maximizing coverage. Prioritization is done by taking a weighted sum of both customer inputs as well as structural aspects of the software. Customer priority is obtained by involving the customer(providing a prioritization of the features as per the expected usage requirements). Structural priority is calculated by considering the primitives of both use case and activity diagrams. Weights are given to the constructs and priority calculated. The priority of a use case is due to combined

priority of customer inputs and structural priority. Scenarios are prioritized based on structural complexity only.

Given constraints of cost and time, it is difficult to test all scenarios. Hence, test scenario selection, aims at selecting a subset of scenarios for testing based on similarity that exists among scenarios. Three techniques, based on similarity measures have been proposed. In the first, Levenshtein distance is used to calculate the similarity between scenarios and used to obtain a subset of scenarios. The second method takes the weight, length and position of the subscenarios into consideration to calculate similarity. In this work, it is presumed that two scenarios are more similar if the relative position of the subsceanrios in the two scenarios is the same. The third method for selection is based on clustering. Clustering provides the advantage of grouping similar scenarios for selection.

An approach for test scenario management is proposed. The increasing size of software systems makes it necessary to manage the test scenarios in an efficient way. In this regard, ontologies help to analyze the relationship between requirements captured using UML diagrams. Scenarios generated from corresponding activity diagrams along with activities, are linked to use cases. Further, software testing concepts adopted from SWEBOK, are used to build an ontology for test management.

Tools implementing the proposed methods are developed and case studies are carried out. The results obtained due to case studies are encouraging: First, the efficacy of rule based consistency checking is demonstrated. The approach followed for scenario generation shows that the use of dependency between activities is an effective way to eliminate unwanted scenarios, particularly in case of concurrent activities. Two ways of doing this(i.e. priority and level of activities) has been proposed, which are effective in reducing the number of scenarios.

The results obtained for prioritization show that automated prioritization tech-

niques to order use case scenarios is effective and can be used in combination with customer assigned priority. The technique achieves a high rate of fault detection. Similarly, the techniques used for test scenario selection show that distance measures can be effectively used as a basis for testing a subset of the entire set of scenarios. Finally, the result obtained in the case studies demonstrate and validate use of the proposed prioritization and selection techniques in testing.

With regards to the ontology for test scenario management, the benefits it provides are two fold. First, it provides a way to infer and reason on use cases and scenarios for testing. This helps in finding undefined relations between use cases thereby helping in test management. Secondly, it provides the users a common language to communicate and query to understand test needs.

7.0.2 Future Work

There could be several natural extensions to this work. In this work, the focus has been on functional requirements only for prioritization of scenarios. An obvious extension could be prioritization considering other functional and non-functional requirements of a system. Also, it is important to consider risk factors while providing weights to activities for prioritization. A new metric to quantify test coverage following the proposed measure needs to be investigated. Also, there is need to study different measures to use for clustering based selection and analyze its suitability. Further, it will be interesting to investigate usage of ontology for all the steps like scenario generation, prioritization and selection dealt here.

7.0.3 Publications

Parts of the present work has evolved into the following conference and journal papers :

1. **'Ensuring Consistency in Relational Repository of UML Models'**,

10th International Conference on Information Technology (ICIT 2007),pp. 217-222, IEEE Computer Society, December 2007.

2. **'Automated Scenario Generation based on UML Activity Diagrams'**, 11th International Conference on Information Technology (ICIT 2008), pp. 209-214, IEEE Computer Society, December 2008.

3. **'Prioritization of Scenarios Based on UML Activity Diagrams'**, 1st International Conference on Computational Intelligence, Communication Systems and Networks(CICSYN 2009), pp. 271-276, IEEE Computer Society, July 2009.

4. **'Prioritizing Use Cases to aid Ordering of Scenarios'**, 3rd European Symposium on Computer Modelling and Simulation, pp. 136-141, IEEE Computer Society, November 2009.

5. **'Using Similarity Measures for Test Scenario Selection'**, 4th International Conference on Industrial & Information Systems, pp.386 - 391, IEEE Computer Society, December 2009.

6. **'Automated Test Scenario Selection based on Levenshtein Distance'**, 6th International Conference on Distributed Computing and Internet Technology, Lecture Notes in Computer Science, pp.255-266, Springer, February 2010.

7. **'Clustering Test Cases to Achieve Effective Test Selection'**, Amrita ACM-W Celebration of Women in Computing, First Conference of Women in Computing in India. (accepted)

8. **'Incorporating UML Primitives in Test Case Prioritization'**, International Journal of Simulation Systems, Science & Technology. (communicated)

9. **'Developing an Ontology for Test Scenario Management'**, 7th International Conference on Distributed Computing and Internet Technology.(communicated)

Bibliography

[Ala04] Alain Abran and James W. Moore(Exec. Eds.),Pierre Bourque and Robert Dupuis(Eds.). Guide to the Software Engineering Body of Knowledge. Technical report, IEEE Computer Society, 2004.

[AO00] A. Abdurazik and J. Offutt. Using uml collaboration diagrams for static checking and test generation. In *Third International Conference on the Unified Modeling Language*, page 383395, 2000.

[aPV04] Roberto Navigli amd Paolo Velardi. Learning domain ontologies for document warehouses and dedicated web sites. *Computational Linguistics*, 30(2):152–179, 2004.

[ASK04] K.K. Aggrawal, Yogesh Singh, and A. Kaur. Code coverage based technique for prioritizing test cases for regression testing. *SIGSOFT Software Engineering Notes*, 29(5):1–4, 2004.

[AvH04] Grigoris Antoniou and Frank van Harmelen. *A Semantic Web Primer*. The MIT Press, 2004.

[BBG05] Sami Beydeda, Matthias Book, and Volker Gruhn. *Model-Driven Software Development*. Springer, 2005.

[BBM02] Francesca Basanieri, Antonia Bertolino, and Eda Marchetti. The Cow_Suite Approach to Planning and Deriving Test Suites in UML Projects. In *UML '02: Proceedings of the 5th International Conference on The Unified Modeling Language*, pages 383–397. Springer-Verlag, 2002.

[BCO09] Daniella Bezerra, Afonso Costa, and Karla Okada. $swto^I$ (Software Test Ontology Integrated) and its Application in Linux Test. In *Proceedings of Ontose 2009*, pages 25–36, 2009.

[BDM02] Simona Bernardi, Susanna Donatelli, and José Merseguer. From UML Sequence Diagrams and Statecharts to Analysable Petrinet Models. In *WOSP '02: Proceedings of the 3rd international workshop on Software and performance*, pages 35–45, New York, NY, USA, 2002. ACM.

[BDZ03] Matthias Beyer, Winfried Dulz, and Fenhua Zhen. Automated ttcn-3 test case generation by means of uml sequence diagrams and markov chains. *Asian Test Symposium*, 0:102, 2003.

[Bec02] *Test Driven Development: By Example.* Addison-Wesley Longman Publishing Co., 2002.

[BEG09] Fevzi Belli, Mubariz Eminov, and Nida Gokce. Model-based test prioritizing a comparative soft-computing approach and case studies. In *KI 2009: Advances in Artificial Intelligence, Lecture Notes in Computer Science*, volume 5803/2009, pages 427–434, 2009.

[Bei02] Boris Beizer. *Software Testing Techniques.* Dreamtech Press, 2002.

[BHHS06] Saartje Brockmans, Peter Haase, Pascal Hitzler, and Rudi Studer. A Metamodel and UML Profile for Rule-Extended OWL DL Ontologies. In *ESWC 2006*, volume 4011, pages 303–316. Springer-Verlag, 2006.

[BHR+00] T. Ball, D. Hoffman, F. Ruskey, R. Webber, , and L. White. State Generation and Automated Class Testing. *The Journal of Software Testing, Verification and Reliability(STVR)*, 10(3):149–170, 2000.

[Bin99] Robert V. Binder. *Testing Object-Oriented Systems Models, Patterns, and Tools.* Addison Wesley, 1999.

[BKK+01] K. Baclawski, M. Kokar, P. Kogut, L. Hart, J. Smith, W. Holmes, J. Letkowski, and M. Aronson. Extending UML to support ontology engineering for the Semantic Web. In *In Fourth International Conference on the Unified Modeling Language*, volume 2185, pages 342–360. M. Gogolla, C. Kobryn (Ed), Lecture Notes in Computer Science, Springer-Verlag, 2001.

[BKK+02] Kenneth Baclawski, Mieczyslaw M. Kokar, Paul A. Kogut, Lewis Hart, Jeffrey E. Smith, Jerzy Letkowski, and Pat Emery. Extending the Unified Modeling Language for ontology development. *Software and System Modeling*, 1(2):142–156, 2002.

[BL01a] L. Briand and Y. Labiche. A uml-based approach to system testing. In *Fourth International Conference on the Uni.ed Modeling Language*, page 194208, 2001.

[BL01b] Lionel Briand and Yvan Labiche. A UML-Based Approach to System Testing. In *Proceedings of the 4th International Conference UML 2001 - The Unified Modeling Language: Modeling Languages, Concepts, and Tools*, volume 2185, pages 194–208. Springer-Verlag Lecture Notes In Computer Science, 2001.

[BL02] Lionel Briand and Yvan Labiche. A UML-Based Approach to System Testing. *Software and Systems Modeling*, 1(1), 2002.

[BLFR02] J.C. Burguillo, M. Llamas, M.J. Fernndez, and T. Robles. Heuristic-driven techniques for test case selection. In *FMICS'02, 7th International ERCIM Workshop in Formal Methods for Industrial Critical Systems (ICALP 2002 Satellite Workshop)*, volume 66, pages 50–65. Electronic Notes in Theoretical Computer Science, 2002.

[BLTC08] Xiaoying Bai, Shufang Lee, Wei-Tek Tsai, and Yinong Chen. Ontology-based test modeling and partition testing of web services. In *ICWS '08: Proceedings of the 2008 IEEE International Conference on Web Services*, pages 465–472. IEEE Computer Society, 2008.

[BMSS09] Swarnendu Biswas, Rajib Mall, Manoranjan Satpathy, and Srihari Sukumaran. A model-based regression test selection approach for embedded applications. *SIGSOFT Software Engineering Notes*, 34(4):1–9, 2009.

[BNM08] Baikuntha Narayan Biswal, Pragyan Nanda, and Durga Prassad Mohapatra. A novel approach for scenario-based test case generation. In *ICIT '08: Proceedings of the 2008 International Conference on Information Technology*, pages 244–247. IEEE Computer Society, 2008.

[BP00] Federico Bergenti and Agostino Poggi. Exploiting UML in the Design of Multi-agent Systems. In *ESAW*, pages 106–113, 2000.

[BR07] Michael Blaha and James Rumbaugh. *Object-Oriented Modeling and Design with UML*. Prentice Hall of India, 2007.

[BRFIGC+02] Juan C. Burguillo-Rial, Manuel J. Fernandez-Iglesias, Francisco J. Gonzalez-Castano, , and Martn Llamas-Nistal. Heuristic-driven test case selection from formal specifications. a case study. In *FME 2002:Formal MethodsGetting IT Right, Lecture Notes in Computer Science*, 2002.

[BRJ05] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 2005.

[BS05] A.K. Bhattacharjee and R.K. Shyamasundar. Validated Code Generation for Activity Diagrams. *Second International Conference on Distributed Computing and Internet Technology, ICDCIT 2005, Lecture Notes in Computer Science*, 3816:508–521, 2005.

[BV06] Umesh Bellur and Vallieswaran. On OO Design Consistency in Iterative Development. In *Proceedings of the 3rd International Conference on Information Technology: New Generations*, pages 46–51, 2006.

[CANM08] Emanuela G. Cartaxo, Wilkerson L. Andrade, Francisco G. Oliveira Neto, and Patrícia D. L. Machado. Lts-bt: A tool to generate and select functional test cases for embedded systems. In *Proceedings of the 2008 ACM symposium on Applied computing*, pages 1540–1544, 2008.

[CDJ+02] A. Cavarra, J. Davies, T. Jeron, L. Mournier, A. Hartman, and S. Olvovsky. Using UML for Automatic Test Generation. In *International Symposium on Software Testing and Analysis (ISSTA 02)*, 2002.

[CGW06] Maria Victoria Cengarle, Peter Graubmann, and Stefan Wagner. Semantics of UML 2.0 Interactions with Variabilities. In *Electronic Notes in Theoretical Computer Science*, volume 160, page 141155, 2006.

[CLL05] Robert Chandler, Chiou Peng Lam, and Huaizhong Li. AD2US: An Automated Approach to Generating Usage Scenarios from UML Activity

Diagrams. In *Proceedings of the 12th Asia-Pacific Software Engineering Conference(APSEC)*. IEEE Computer Society, 2005.

[CLM04] T.Y. Chen, H. Leung, and I.K. Mak. Adaptive random testing. In *Proceedings of 9th Asian Computing Science Conference, Advances in Computer Science - ASIAN 2004: Higher-Level Decision Making*, pages 320–329. Springer-Verlag, 2004.

[CLOM06] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. Object distance and its application to adaptive random testing of object-oriented programs. In *Proceedings of the 1st International Workshop on Random Testing (ACM)*, pages 55–63, 2006.

[CLSC98] Kai H. Chang, Shih-Sung Liao, Stephen B. Seidman, and Richard Chapman. Testing object-oriented programs: from formal specification to test scenario generation. 42:141–151, 1998.

[CMK08] Mingsong Chen, Prabhat Mishra, and Dhrubajyoti Kalita. Coverage-driven Automatic Test Generation for UML Activity Diagrams. In *Proceedings of the 18th ACM Great Lakes Symposium on VLSI (GLSVLSI)*, pages 139–142. ACM, 2008.

[CMX06] Qiu Xiaokang Chen Mingsong and Li Xuandong. Automatic Test Case Generation for UML Activity Diagrams. In *Proceedings of the 2006 international workshop on Automation of Software Testing (AST'06)*. ACM Press, 2006.

[CNM07] Emanuela G. Cartaxo, Francisco G. Oliveira Neto, and Patrícia D. L. Machado. Automated test case selection based on a similarity function. In *GI Jahrestagung (2)*, pages 399–404, 2007.

[CP99] Stephen Cranefield and Martin Purvis. UML as an Ontology Modelling Language. In *In Proceedings of the IJCAI-99 Workshop on Intelligent Information Integration*, 1999.

[CP02] Stephen Cranefield and Martin Purvis. A uml profile and mapping for the generation of ontology-specific content languages. *Knowledge Engineering Review*, 17(1):21–39, 2002.

[CP03a] Yanping Chen and Robert L. Probert. A risk-based regression test selection strategy. In *ISSRE 2003: Proceeding of the 14th IEEE International Symposium on Software Reliability Engineering*, pages 305–306, 2003.

[CP03b] Yanping Chen and Robert L. Probert. A risk-based regression test selection strategy. In *Proceeding of the 14th IEEE International Symposium on Software Reliability Engineering (ISSRE 2003), Fast Abstract*, pages 305–306. IEEE Computer Society, 2003.

[CPC+04] Dan Chiorean, Mihai Pasca, Adrian Cârcu, Cristian Botiza, and Sorin Moldovan. Ensuring uml models consistency using the ocl environment. *Electronic Notes Theoretical Computer Science*, 102:99–110, 2004.

[CPS02] Yanping Chen, Robert L. Probert, and D. Paul Sims. Specification-based regression test selection with risk analysis. In *CASCON '02: Proceedings of the 2002 conference of the Centre for Advanced Studies on Collaborative Research*, pages 175 – 182,. IBM Press, 2002.

[CPTT05] T.Y. Chen, Pak-Lok Poon, Sau-Fun Tang, and T.H. Tse. Identification of Categories and Choices in Activity Diagrams. In *Proceedings of the 5th International Conference on Quality Software (QSIC)*. IEEE Computer Society, 2005.

[CQX+07a] Mingsong Chen, Xiaokang Qiu, Wei Xu, Linzhang Wang, Jianhua Shao, and Xuandong Li. UML Activity Diagram-Based Automatic Test Case Generation for Java Programs. *The Computer Journal*, 2007.

[CQX+07b] Mingsong Chen, Xiaokang Qiu, Wei Xu, Linzhang Wang, Jianhua Shao, and Xuandong Li. UML Activity Diagram-Based Automatic Test Case Generation for Java Programs. *Oxford University Press*, 52(5):545–556, 2007.

[CTF01] Philippe Chevalley and Pascale Thevenod-Fosse. Automated generation of statistical test cases from uml state diagrams. *Computer Software and Applications Conference, Annual International*, 0:205, 2001.

[DCW08] T.S. Dillon, E. Chang, and P. Wongthongtham. Ontology-based software engineering-software engineering 2.0. In *In Proceedings of the 19th Australian Conference on Software Engineering*, pages 13–23. IEEE, 2008.

[Der02] John Derrick. A framework for UML Consistency. In *Workshop on Consistency Problems in UML-Based Software Development*, 2002.

[dFBRG06] Gregory de Fombelle, Xavier Blanc, Laurent Rioux, and Marie-Pierre Gervais. Finding a path to model consistency. In *Second European Conference on Model Driven Architecture - Foundations and Applications(ECMDA-FA 2006)*, pages 101–112, 2006.

[DH99] Birgit Demuth and Heinrich Hussmann. Using UML/OCL Constraints for Relational Database Design. In *UML'99: The Unified Modeling Language - Beyond the Standard, Second International Conference*, pages 598 – 613. Springer, 1999.

[DHL01] Birgit Demuth, Heinrich Hussmann, and Sten Loecher. OCL as a Specification Language for Business Rules in Database Applications. In *UML 2001 The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, pages 104–117. Springer, 2001.

[DJK+99] S. R. Dalal, A. Jain, N. Karunanithi, J.M. Leaton, C. M. Lott, G.C. Patton, and B.M. Horowitz. Model-based testing in practice. In *Proc. Intl. Conf. on Software Engineering (ICSE 99*, pages 285–294, 1999.

[DL04] Yannick Valot Jean-Pierre Gallois David Lugato, C line Bigot. Validation and automatic test generation on uml models:the agatha approach.

International Journal on Software Tools Technology Transfer, 5(2):124–139, 2004.

[DP06] Brian Dobing and Jeffrey Parsons. How UML is used. *Communications of the ACM*, 2006.

[DP08] Brian Dobing and Jeffrey Parsons. Dimensions of UML Diagram Use:A Survey of Practitioners. *Journal of Database Management, IGI Publishing*, 2008.

[DRK04] H. Do, G. Rothermel, and A. Kinneer. Empirical studies of test case prioritization in a JUnit testing environment. In *Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE)*, pages 113–124. IEEE Computer Society, November 2004.

[DTGF06] T. Dinh-Trong, S. Ghosh, and R. France. A Systematic Approach to Generate Inputs to Test UML Design Models. In *Proceedings of the 17th IEEE International Symposium on Software Reliability Engineering*, pages 95–104, 2006.

[DuC] Bob DuCharme. *XSLT Quickly.* Manning Publications.

[eaCH+02] Paul Kogut et al, Stephen Cranefield, Lewis Hart, Mark Dutra, Kenneth Baclawski, Mieczyslaw Kokar, and Jeffrey Smith. UML for Ontology Development. *The Knowledge Engineering Review*, 17(1):61–64, 2002.

[EGR01] S. Elbaum, D. Gable, and G. Rothermel. Understanding and measuring the sources of variation in the prioritization of regression test suites. In *Proceedings of the 7th International Software Metrics Symposium*, pages 169–179. IEEE Computer Society, April 2001.

[Egy01] Alexander Egyed. Scalable Consistency Checking between Diagrams The VIEWINTEGRA Approach. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE)*, page 387. IEEE Computer Society, 2001.

[Egy06] Alexander Egyed. Instant consistency checking for the UML. In *Proceedings of the 28th International conference on Software engineering(ICSE 2006)*, pages 381–390, 2006.

[Egy07] Alexander Egyed. Fixing Inconsistencies in UML Design Models. In *Proceedings of the 29th International conference on Software Engineering(ICSE 2007)*, pages 292–301, 2007.

[EMR00] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. Prioritizing test cases for regression testing. In *Proceedings of the ACM International Symposium on Software Testing and Analysis*, pages 102–112, August 2000.

[EMR01] S. Elbaum, A. Malishevsky, and G. Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE)*, pages 329–338. IEEE Computer Society, May 2001.

[EMR02] S. Elbaum, A. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering*, 28(2):159–182, February 2002.

[ERKM04] S. Elbaum, G. Rothermel, S. Kanduri, and A. G. Malishevsky. Selecting a cost-effective test case prioritization technique. *Software Quality Journal*, 12(3):185–210, September 2004.

[FL02] Falk Fraikin and Thomas Leonhardt. SeDiTeC Testing Based on Sequence Diagrams. In *Proceedings of the IEEE International Conference on Automated Software Engineering*, 2002.

[FSS03] K. Falkovych, M. Sabou, and H. Stuckenschmidt. UML for the Semantic Web: Transformation-Based Approaches. In *Knowledge Transformation for the Semantic Web, eds., Frontiers in Artificial Intelligence and Applications*, volume 95, pages 92–106, 2003.

[FST96] Anthony Finkelstein, George Spanoudakis, and David Till. Managing Interference. In *Second International Software Architecture Workshop(ISAW-2) and International Workshop on Multiple Perspectives in Software Development (Viewpoints 96)*, page 172174. ACM Press, 1996.

[FW07] Gordon Fraser and Franz Wotawa. Test-case prioritization with model-checkers. In *Proceedings of the IASTED International Conference on Software Engineering*, 2007.

[GdF04] Rosario Girardi and Carla Gomes de Faria. An Ontology-based Technique for the Specification of Domain and User Models in Multi-Agent Domain Engineering. *CLEI Electronic Journal*, 7(1), 2004.

[GEB06] Nida Gokce, Mubariz Eminov, and Fevzi Belli. Coverage-Based, Prioritized Testing Using Neural Network Clustering. In *In Proceedings of Computer and Information Sciences ISCIS 2006, Lecture Notes in Computer Science*, volume 4263/2006, pages 1060–1071. Springer, 2006.

[GEMT06] Javier Jesús Gutiérrez, María José Escalona, Manuel Mejías, and Jesús Torres. An approach to generate test cases from use cases. In *ICWE '06: Proceedings of the 6th international conference on Web engineering*, pages 113–114, New York, NY, USA, 2006. ACM.

[GKW01] H.W. Six G. Kosters and M. Winter. Coupling use cases and class models as a means for validation and verification of requirements specification. 6(1):14, 2001.

[Gli00] Martin Glinz. Problems and Deficiencies of UML as a Requirements Specification Language. In *In Proceedings of the 10th International Workshop on Software Specification and Design(IWSSD '00)*, page 11. IEEE Computer Society, 2000.

[GLM04] Stefania Gnesi, Diego Latella, and Mieke Massink. Formal test-case generation for uml statecharts. In *Proceedings of the Ninth IEEE International Conference on Engineering Complex Computer Systems*, pages 75–84. IEEE, 2004.

[GRCJ99] Rothermel Gregg, Untch Roland, Chu Chengyun, and Harrold Mary Jean. Test case prioritization: An empirical study. In *ICSM '99: Proceedings of the International Conference on Software Maintenance*, pages 179–188. IEEE Computer Society, 1999.

[Gro] Standish Group. http://www.standishgroup.com/chaos.htm.

[Gru93] Tom Gruber. A Translation Approach to Portable Ontology Specifications. *Knowledge Acquisition*, 5(2):199–220, 1993.

[Gru03] Thomas Gruber. It Is What It Does: The Pragmatics of Ontology. In *Invited presentation to the meeting of the CIDOC Conceptual Reference Model committee, Smithsonian Museum, Washington*, 2003.

[Gru08] Tom Gruber. Ontology. *Encyclopedia of Database Systems*, Ling Liu and M. Tamer zsu (Eds.), 2008.

[HBB06] Axel Hollmann, Fevzi Belli, and Christof J. Budnik. Test case generation and selection based on statecharts extension of the holistic approach. In *Supplementary Proceedings of the 17th IEEE International Symposium on Software Reliability Engineering (ISSRE 2006)*, 2006.

[Hes06] Anders Hessel. Model-Based Test Case Selection and Generation for Real-Time Systems, PhD thesis. Technical report, In Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology, 2006.

[HHKT02] Bougumila Hnatkowska, Zbigniew Huzar, Ludwik Kuzniarz, and Lech Tuzinkiewicz. A systematic approach to consistency within uml based software development process. In *In Proceedings of Workshop on Consistency Problems in UML-Based Software Development(UML 2002)*, 2002.

[HHS02] Jan Hendrik Hausmann, Reiko Heckel, and Stefan Sauer. Extended Model Relations with Graphical Consistency Conditions. In *Workshop on Consistency Problems in UML-Based Software Development(UML 2002)*, 2002.

[HHU00] S.D. Cha D.H. Bae H.S. Hong, Y.G. Kim and H. Ural. A test sequence selection method for statecharts. In *Software Testing, Verification and Reliability(STVR)*, pages 10(4):203–227, 2000.

[HIM00] Jean Hartmann, Claudio Imoberdorf, and Michael Meisinger. Uml-based integration testing. In *ISSTA '00: Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, pages 60–70. ACM, 2000.

[HJL+01] Mary Jean Harrold, James A. Jones, Tongyu Li, Donglin Liang, Alessandro Orso, Maikel Pennings, Saurabh Sinha, S. Alexander Spoon, and Ashish Gujarathi. Regression test selection for java software. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 312–326. ACM, 2001.

[HK03] Il-Kyu Ha and Byung-Wook Kang. Meta-validation of UML structural diagrams and behavioral diagrams with consistency rules. In *IEEE Pacific Rim Conference on Communications, Computers and signal Processing(PACRIM 2003)*, pages Vol.2, 679–683. IEEE, 2003.

[HS04] Zhaoxia Hu and Sol M. Shatz. Mapping UML Diagrams to a Petri Net Notation for System Simulation. In *SEKE*, pages 213–219, 2004.

[HS06] Hans-Jrg Happel and Stefan Seedorf. Applications of ontologies in software engineering. In *In 2nd International Workshop on Semantic Web Enabled Software Engineering (SWESE 2006), held at the 5th International Semantic Web Conference (ISWC 2006*, 2006.

[HSNL06] O-Hoon Choi Hong-Seok Na and Jung-Eun Lim. A Metamodel-Based Approach for Extracting Ontological Semantics from UML Models. In *WISE 2006*. Springer Verlag, 2006.

[Inv01] Paola Inverardi. Automated Check of Architectural Models Consistency using SPIN. In *16th International Conference on Automated Software Engineering (ASE'01)*, 2001.

[JCSdPLK00] Jorge Horacio Doorn Julio Cesar Sampaio do Prado Leite, Graciela D. S. Hadad and Gladys N. Kaplan. A scenario construction process. volume 5. Springer, 2000.

[JE94] Paul C. Jorgensen and Carl Erickson. Object-Oriented Integration Testing. *Communications of the ACM*, 37(9):30–38, 1994.

[JG06] Dennis Jeffrey and Neelam Gupta. Test case prioritization using relevant slices. In *Proceedings of the 30th Annual International computer Software and Application Conference (COMPSAC)*, 2006.

[JH03a] Yan Jiong Wang Ji and Chen Huowang. Deriving software statistical testing model from uml model. In *QSIC '03: Proceedings of the Third International Conference on Quality Software*, page 343. IEEE Computer Society, 2003.

[JH03b] James A. Jones and Mary Jean Harrold. Test-suite reduction and prioritization for modified condition/decision coverage. *IEEE Transactions on Software Engineering*, 29(3):195–209, March 2003.

[JMY04] Igor Jurisica, John Mylopoulos, and Eric Yu. Ontologies for Knowledge Management: An Information Systems Perspective. *Journal of Knowledge and Information Systems*, 6:380–401, 2004.

[JRB01] Ivor Jacobson James Rumbaugh and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 2001.

[JW06] Praveen K. Jayaraman and Jon Whittle. Use case-based acceptance testing of a large industrial system: Approach and experience report. In *Proceedings of the Testing Academic and Industrial Conference - Practice And Research Techniques (TAIC PART'06)*, pages 211–220. IEEE, 2006.

[JW07] Praveen K. Jayaraman and Jon Whittle. UCSIM: A Tool for Simulating Use Case Scenarios. In *ICSE COMPANION '07: Companion to the proceedings of the 29th International Conference on Software Engineering*, pages 43–44. IEEE Computer Society, 2007.

[kaw03] *Fault Detection Effectiveness of UML Design Model Test Adequacy Criteria*. IEEE Computer Society, 2003.

[KC04] Soon Kyeong Kim and David Carrington. A Formal Object-Oriented Approach to define Consistency Constraints for UML Models. In *Australian Software Engineering Conference (AWSEC 2004)*. IEEE Computer Society, 2004.

[KFdB+04] Marcel Kyasa, Harald Fechera, Frank S. de Boera, Joost Jacoba, Jozef Hoomana, Mark van der Zwaaga, Tamarah Aronsa, and Hillel Kuglera. Formalizing UML Models and OCL Constraints in PVS. In *Proceedings of the Second Workshop on Semantic Foundations of Engineering Design Languages (SFEDL 2004)*, pages Vol. 115, pp.39–47, 2004.

[KG95] Peter D. Karp and Thomas R. Gruber. A Generic Knowledge-base Access Protocol. In *Proceedings of the International Joint Conferences on Artificial Intelligence*, 1995.

[KK03] L. Ol Khovich and D.V. Koznov. OCL-Based Automated Validation Method for UML Specifications. *Programming and Computer Software*, 2003.

[KK09a] Bogdan Korel and George Koutsogiannakis. Experimental comparison of code-based and model-based test prioritization. In *IEEE International Conference on Software Testing Verification and Validation Workshops*, 2009.

[KK09b] Debasish Kundu and Debasish Kundu. A novel approach to generate test cases from uml activity diagrams. *Journal of Object Technology*, 8(3):65–83, 2009.

[KKBK07] Hyungchoul Kim, Sungwon Kang, Jongmoon Baik, and Inyoung Ko. Test Cases Generation from UML Activity Diagrams. In *Proceedings of the 8th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing(ACIS-SNPD)*, pages 556–661. IEEE Computer Society, 2007.

[KM09] R. Krishnamoorthi and S.A. Sahaaya Arul Mary. Factor oriented requirement coverage based system test case prioritization of new and regression test cases. *Information Software Technology*, 51(4):799–808, 2009.

[KP02] Jung-Min Kim and Adam A. Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 119–129. ACM Press, May 2002.

[KPBP04] Aditya Kalyanpur, Daniel Jimenez Pastor, Steve Battle, and Julian Padget. Automatic mapping of owl ontologies into java. In *In 16th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pages 98–103, 2004.

[KR97] Joachim Karlsson and Kevin Ryan. A cost-value approach for prioritizing requirements. *IEEE Software*, 14:67–74, 1997.

[KR03] Supaporn Kansomkeat and Wanchai Rivepiboon. Automated-generating test case using uml statechart diagrams. In *SAICSIT '03: Proceedings of the 2003 annual research conference of the South African institute of computer scientists and information technologists on Enablement through technology*, pages 296–300. South African Institute for Computer Scientists and Information Technologists, 2003.

[Kri00] Padmanabhan Krishnan. Consistency Checks for UML. In *Proceedings of the 7th Asia Pacific Software Engineering Coference (APSEC '00)*, pages 162–169. IEEE, 2000.

[KS07] Debasish Kundu and Debasis Samanta. A novel approach of prioritizing use case scenarios. volume 0, pages 542–549, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

[LBMS05] K.C. Lavanya, K.V. Balakishore, Hrushikesha Mohanty, and R.K. Shyamasundar. How Good is a UML Diagram? A Tool to Check It. In *Proceedings of TENCON*, pages 386–391. IEEE Computer Society, 2005.

[LCA02] K. Lano, D. Clark, and K. Androutsopoulos. Formalising Inter-Model Consistency of the UML. In *Workshop on Consistency Problems in UML-Based Software Development(UML 2002)*, 2002.

[Lev65] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Doklady Akademii Nauk SSSR*, page 163(4):845848, 1965.

[Lev06] V. I. Levenshtein. An evaluation of combination strategies for test case selection. In *Empirical Software Engineering*, page 11(4), 2006.

[LHH07] Zheng Li, Mark Harman, and Robert M. Hierons. Search algorithms for regression test case prioritization. *IEEE Transactions on Software Engineering*, 33(4), April 2007.

[LHS01] Minkyu Lee, Dongsoo Han, and Jaeyong Shim. Set-Based Access Conflicts Analysis of Concurrent Workflow Definition. In *Proceedings of the 3rd International Symposium on Cooperative Database Systems for Advanced Applications(CODAS)*, pages 172–176. IEEE Computer Society, April 2001.

[LJX+04] Wang Linzhang, Yuan Jiesong, Yu Xiaofeng, Hu Jun, Li Xuandong, and Zheng Guoliang. Generating Test Cases from UML Activity Diagram based on Gray-Box Method. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference(APSEC)*. IEEE Computer Society, 2004.

[LL04] Huaizhong Li and Chiou Peng Lam. Optimization of state-based test suites for software systems: An evolutionary approach. *International Journal of Computer and Information Science*, 5(3), 2004.

[LL05a] Huaizhong Li and C. Pen Lam. An ant colony optimization approach to test sequence generation for state based software testing. In *Proceedings of the Fifth International Conference on Quality Software (QSIC05)*. IEEE, 2005.

[LL05b] Huaizhong Li and C. Peng Lam. An ant colony optimization approach to test sequence generation for statebased software testing. In *Proceedings of the Fifth International Conference on Quality Software*, pages 255–264, 2005.

[LL05c] Huaizhong Li and C. Peng Lam. Using Anti-Ant-like Agents to Automatically Generate Test Threads from UML Diagrams. In *Proceedings of 17th IFIP International Conference on Testing of Communicating Systems (TESTCOM)*. IEEE Computer Society, 2005.

[LLH01] X. Li, Z. Liu, and J. He. Formal and Use-Case Driven Requirement Analysis in UML. In *25th Annual International Computer Software and Applications Conference*, pages 215–224, 2001.

[LS06] Mass Soldal Lund and Ketil Stolen. Deriving Tests from UML 2.0 Sequence diagrams with neg and assert. In *Proceedings of AST '06*, 2006.

[LTY03] Boris Litvak, Shmuel Tyszberowicz, and Amiram Yehudai. Behavioral consistency validation of uml diagrams. In *SEFM*, pages 118–125, 2003.

[MA] Francisco Javier Lucas Martnez and Ambrosio Toval Alvarez. A Precise Approach for the Analysis of the UML Models Consistency. In *J. Akoka et al. (Eds.): ER Workshops 2005*, volume 3770.

[Mai78] David Maier. The complexity of some problems on subsequences and supersequences. In *Doklady Akademii Nauk SSSR*, page 25(2): 322336. ACM Press, 1978.

[MAQ] Olavo Mendes, Alain Abran, and Hc K Montral Qubec. Software engineering ontology: A development methodology. In *In Metrics News*.

[Mat07] Aditya P Mathur. *Foundations of Software Testing.* Addison-Wesley, 2007.

[MCHI97] T.K. Tsai Mei-Chen Hsueh and R.K. Iyer. Fault Injection Techniques and Tools. *Computer*, 30(4):75–82, 1997.

[MF97] Natalia Juristo Mariano Ferndandez, Asunci6n Gomez-Perez. METHONTOLOGY: From Ontological Art Towards Ontological Engineering. *AAAI Technical Report SS-97-06*, 15(2), 1997.

[MFRW00] Deborah L. McGuinness, Richard Fikes, James Rice, and Steve Wilder. An Environment for Merging and Testing Large Ontologies. In *Proceedings of the Seventh International Conference on Principles of Knowledge Representation and Reasoning(KR2000)*, 2000.

[MMB09] M.M.Kokar, C.J. Matheus, and K. Baclawski. Ontology-based situation awarness. *Information Fusion*, 10:83–98, 2009.

[Moi00] F. Moisiadis. Prioritising Use Cases and Scenarios. In *Proceedings of the 37th International Conference on Technology of Object-Oriented Languages and Systems(TOOLS)*, pages 108–119. IEEE Computer Society, August-September 2000.

[MRRE06] Alexey G. Malishevsky, Joseph R. Ruthruff, Gregg Rothermel, and Sebastian Elbaum. Cost-cognizant test case prioritization. *Technical Report TR-UNL-CSE-2006-0004, Department of Computer Science and Engineering, University of NebraskaLincoln, Lincoln, Nebraska, U.S.A.*, 2006.

[MSS] Tom Mens, Ragnhild Van Der Straeten, and Jocelyn Simmonds. Maintaining consistency between uml models with description logic tools.

[MT07] Siavash Mirarab and Ladan Tahvildari. A prioritization approach for software test cases based on bayesian networks. In *FASE'07: Proceedings of the 10th international conference on Fundamental approaches to software engineering*, pages 276–290. Springer-Verlag, 2007.

[MW] Thusitha Mabotuwana and Jim Warren. An ontology based approach to enhance querying capabilities of general practice medicine for better management of hypertension. *Artificial Intelligence in Medicine.*

[MXX06] Chen Mingsong, Qiu Xiaokang, and Li Xuandong. Automatic Test Case Generation for UML Activity Diagrams. In *International Conference on Software Engineering, Proceedings of the 2006 International workshop on Automation of Software Test.* ACM, 2006.

[Mye79] G.J. Myers. *The Art of Software Testing.* John Wiley & Sons, 1979.

[MZ08] Zengkai Ma and Jianjun Zhao. Test case prioritization based on analysis of program structure. In *APSEC '08: Proceedings of the 2008 15th Asia-Pacific Software Engineering Conference*, pages 471–478, Washington, DC, USA, 2008. IEEE Computer Society.

[NC08] Aroado Nugroho and Michel R.V. Chaudron. A Survey into the Rigor of UML Use and its Perceived Impact on Quality and Productivity. In *2nd International Symposium of Empirical Software Engineering and Measurement*. IEEE, 2008.

[NM01] Natalya F. Noy and Deborah L. McGuinness. Ontology Development 101: A Guide to Creating Your First Ontology. *Stanford Knowledge Systems Laboratory Technical Report KSL-01-05 and Stanford Medical Informatics Technical Report SMI-2001-0880*, 15(2), 2001.

[NPLTJ03] C. Nebut, S. Pickin, Y. Le Traon, and J.-M. Jezequel. Automated requirements-based generation of test cases for product families. In *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings*, pages 263 – 266, 2003.

[NPT08] Cu D. Nguyen, Anna Perini, and Paolo Tonella. Ontology-based Test Generation for Multiagent Systems. In *AAMAS '08: Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems*, pages 1315–1320. International Foundation for Autonomous Agents and Multiagent Systems, 2008.

[OA99] Jeff Offutt and Aynur Abdurazik. Generating Tests from UML Specifications. In *In Proceedings of the 2nd International Conference on the UML, Lecture Notes in Computer Science*, volume 1723, pages 416–429. Springer Verlag, 1999.

[Obe06] Daniel Oberle. Semantic management of middleware. In *Series: Semantic Web and Beyond*, volume 1, page 268. Springer, 2006.

[ocl06] OMG: Object Constraint Language Specification in OMG Unified Modeling Language Specification Version 2.0. Technical report, Object Management Group Inc., 2006.

[odm06] OMG: Ontology Definition Metamodel. Technical report, IEEE Computer Society, 2006.

[OLAA03] Jeff Offutt, Shaoying Liu, Aynur Abdurazik, and Paul Ammann. Generating test data from state-based specifications. *The Journal of Software Testing, Verification and Reliability*, 13:25–53, 2003.

[OMG] OMG. Unified Modeling Language (UML) Superstructure Specification, version 2.1. Technical report.

[OXL99] A. Jefferson Offutt, Yiwei Xiong, and Shaoying Liu. Criteria for generating specification-based tests. In *In Proceedings of the Fifth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 99*, pages 119–129. IEEE Computer Society Press, 1999.

[PAK+07] Orest Pilskalns, Anneliese Andrews, Andrew Knight, Sudipto Ghosh, and Robert France. Testing uml designs. *Information and Software Technology*, 2007.

[PAM99] K. Periyasamy, V.S. Alagar, and D. Muthiayen. Verification and validation techniques of object-oriented software systems. *Technology of Object-Oriented Languages, International Conference on*, 0:413, 1999.

[PK10] Samad Paydar and Mohsen Kahani. Ontology-Based Web Application Testing. *Novel Algorithms and Techniques in Telecommunications and Networking*, pages 23–27, 2010.

[PRB08] Hyuncheol Park, Hoyeon Ryu, and Jongmoon Baik. Historical value-based approach for cost-cognizant test case prioritization to improve the effectiveness of regression testing. In *SSIRI '08: Proceedings of the 2008 Second International Conference on Secure System Integration and Reliability Improvement*, pages 39–46, Washington, DC, USA, 2008. IEEE Computer Society.

[Pre05] Roger Pressman. *Software Engineering: A Practitioner's Approach.* McGraw-Hill, 2005.

[PST08] Sapna Ponaraseri, Angelo Susi, and Paolo Tonella. Using the planning game for test case prioritization. In *19th International Symposium on Software Reliability Engineering (ISSRE)*, 2008.

[PT06] D. Oberle E. Wallace M. Uschold E. Kendall P. Tetlow, J. Pan. Ontology Driven Architectures and Potential Uses of the Semantic Web in Software Engineering. Technical report, W3C, Semantic Web Best Practices and Deployment Working Group, Draft, 2006.

[QNXZ07] Bo Qu, Changhai Nie, Baowen Xu, and Xiaofang Zhang. Test case prioritization for black box testing. In *COMPSAC '07: Proceedings of the 31st Annual International Computer Software and Applications Conference*, pages 465–474. IEEE Computer Society, 2007.

[RH96] Gregg Rothermel and Mary Jean Harrold. Analyzing regression test selection techniques. In *IEEE Transactions on Software Engineering*, pages v.22 n.8, pp.529–551, 1996.

[RH97] G. Rothermel and M. Harrold. A safe, efficient regression test selection technique. In *ACM Transactions on Software Engineering and Methodology*, pages 6(2):173–210, 1997.

[RH06] Francisco Ruiz and Jos R. Hilera. Using Ontologies in Software Engineering and Technology. In *In Coral Calero, Francisco Ruiz and Mario Piattini(eds.) Ontologies in Software Engg and Technology.* Springer, 2006.

[RM09] R.Krishnamoorthi and S.A.Sahaaya Arul Mary. Regression test suite prioritization using genetic algorithms. *International Journal of Hybrid Information Technology*, 2(3):35–52, 2009.

[Rou03] Boris Roussev. Generating OCL Specifications and Class Diagrams from Use Cases: A Newtonian Approach. In *HICSS*, 2003.

[RPG03] Matthias Riebisch, Ilka Philippow, and Marco Gotze. Uml-based statistical test case generation. In *NODe '02: Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*, pages 394–411. Springer-Verlag, 2003.

[RS00] N. Ritter and H.P. Steiert. Enforcing modeling guidelines in an ordbms-based uml-repository. In *Proceedings of the International Resource Management Association Conference*, 2000.

[RUCH01] Gregg Rothermel, Roland Untch, Chengyun Chu, and Mary Jean Harrold. Test case prioritization. *IEEE Transactions on Software Engineering*, 27(10):929–948, October 2001.

[RW02] Holger Rasch and Heike Wehrheim. Consistency between UML Classes and Associated State Machines. In *Workshop on Consistency Problems in UML-Based Software Development(UML 2002)*, 2002.

[SBHKR08] Jocelyn Simmonds, M. Cecilia Bastarrica, Nancy Hitschfeld-Kahler, and Sebastián Rivas. A Tool Based on DL for UML Model Consistency Checking. *International Journal of Software Engineering and Knowledge Engineering*, 18(6):713–735, 2008.

[SC02] Jean-Louis Sourrouille and Guy Caplat. Constraint checking in uml modeling. In *Proceedings of the International Conference on Software Engineering and Knowledge Engineering(SEKE 2002)*, pages 217–224. IEEE Computer Soceity, 2002.

[SCG] Miguel-Angel Sicilia and Juan J. Cuadrado-Gallego. Linking Software Engineering concepts to upper ontologies. In *First Workshop on Ontology, Conceptualizations and Epistemology for Software and Systems Engineering(ONTOSE)*.

[Seo06] Seo Heui Seok. Generating test sequences from statecharts for concurrent program testing. *IEICE Transactions on Information and Systems (Institute of Electronics, Information and Communication Engineers)*, E89-D(4):1459–1469, 2006.

[Sha06] Hong Zhu Lijun Shan. Well-formedness, Consistency and Completeness of Graphic Models. In *Proceedings of UKSIM06*, pages 47–53, 2006.

[Shi06] Yoshiyuki Shinkawa. Inter-Model Consistency in UML based on CPN Formalism. In *Proceedings of the 13th Asia-Pacific Software Engineering Conference(APSEC)*. IEEE Computer Society, 2006.

[SHP06] Hamid Haidarian Shahri, James A. Hendler, and Adam A. Porter. Software Configuration Management Using Ontologies. In *2nd International Workshop on Semantic Web Enabled Software Engineering (SWESE 2006)*, 2006.

[Sic06] Miguel-Angel Sicilia. Ontology of Systems and Software Engineering. *Advanced Engineering Informatics*, 21:117–118, 2006.

[SK06] Motoshi Saeki and Haruhiko Kaiya. On Relationships among Models, Meta Models and Ontologies. In *In J. Gray, J.P. Tolvanen, J. Sprinkle(eds.) Proceedings of the 6th OOPSLA Workshop on Domain-Specific Modeling*, 2006.

[SKS] Petri Selonen, Kai Koskimies, and Markku Sakkinen. How to Make Apples from Oranges in UML.

[SM00] John Anil Saldhana and Sol M.Shatz. UML Diagrams to Object Petri Net Models:An Approach for Modeling and Analysis. In *International Conference on Software Engineering and Knowledge Engineering(SEKE)*, 2000.

[SMK07] Philip Samuel, Rajib Mall, and Pratyush Kanth. Automatic Test Case Generation from UML Communication Diagrams. *Information and Software Technology*, 49(2):158 – 171, 2007.

[SMP08] Heiko Stallbaum, Andreas Metzger, and Klaus Pohl. An automated technique for risk-based test case generation and prioritization. In *AST '08: Proceedings of the 3rd international workshop on Automation of software test*, pages 67–70, New York, NY, USA, 2008. ACM.

[Soe99] Dagobert Soergel. The rise of ontologies or the reinvention of classification. *Journal of the American Society for Information Science*, 50(12):1119–1120, 1999.

[Sok06] Dehla Sokenou. Generating Test Sequences from UML Sequence Diagrams and State Diagrams. *Lecture Notes in Informatics*, 2(94):236–240, 2006.

[SR92] M.D. Smith and D.J. Robson. A Framework for Testing Object-Oriented Programs. *Journal of Object Oriented Programming*, pages 45–53, 1992.

[Sri04] Hema Srikanth. Requirements-based test case prioritization. In *Student Research Forum at the 12th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, 2004.

[SS05] Siros Suravita and Taratip Suwannasart. Testing Polymorphic Interactions in UML Sequence Diagrams. In *Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC 05)*, 2005.

[ST02] Amitabh Srivastava and Jay Thiagarajan. Effectively prioritizing tests in development environment. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 97–106. ACM Press, July 2002.

[SW05] H. Srikanth and L. Williams. On the economics of requirements-based test case prioritization. In *Proceedings of the Seventh International Workshop on Economics-Driven Software Engineering Research*, 2005.

[SWO05] H. Srikanth, L. Williams, and J. Osborne. System test case prioritization of new and regression test cases. In *Proceedings of the 4th International Symposium on Empirical Software Engineering (ISESE)*, pages 62–71. IEEE Computer Society, November 2005.

[TAS06] Paolo Tonella, Paolo Avesani, and Angelo Susi. Using the case-based ranking methodology for test case prioritization. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 123–133. IEEE Computer Society, September 2006.

[TE00] Aliki Tsiolakis and Hartmut Ehrig. Consistency Analysis of UML Class and Sequence Diagrams using Attributed Graph Grammars. In *Workshop on Graph Transformation Systems(GraTra)*, pages 77–86, 2000.

[TSYP03] Wei-Tek Tsai, Akihiro Saimi, Lian Yu, and Raymond A. Paul. Scenario-based object-oriented testing framework. In *QSIC*, pages 410–. IEEE Computer Society, 2003.

[TZP+07] W. T. Tsai, Xinyu Zhou, Raymond A. Paul, Yinong Chen, and Xiaoying Bai. A coverage relationship model for test case selection and ranking for multi-version software. In *Proceedings of the 10th IEEE High Assurance Systems Engineering Symposium(HASE 07)*, pages 105–112. IEEE Computer Society, 2007.

[UJ99] Mike Uschold and Robert Jasper. A Framework for Understanding and Classifying Ontology Applications. In *Proceedings of the IJCAI-99 workshop on Ontologies and Problem-Solving Methods (KRR5)*, 1999.

[uml07] OMG: Unified Modeling Language Specification Version. 2.1. Technical report, Object Management Group Inc., 2007.

[Val05] Andre Valente. Types and Roles of Legal Ontologies. *Law and the Semantic Web, Springer-Verlag Berlin Heidelberg*, 3369:65–76, 2005.

[WBLH07] Yongbo Wang, Xiaoying Bai, Juanzi Li, and Ruobo Huang. Ontology-based Test Case Generation for Testing Web Services. In *Eigth International Symposium on Autonomous Decentralized Systems(ISADS)*, pages 43–50. IEEE, 2007.

[WC05] Chang E. Wongthongtham, P. and C. Cheah. Software Engineering Sub-Ontology for Specific Software Development. In *Proceedings of 29th Annual IEEE/NASA Software Engineering Workshop (SEW05)*, pages 27–33. IEEE, 2005.

[WCD05] P Wongthongtham, E Chang, and T Dillon. Towards Ontology-based Software Engineering for Multi-site Software Development. In *3rd IEEE International Conference on Industrial Informatics(INDIN)*, pages 362–365. IEEE, 2005.

[WCS07] Pornpit Wongthongtham, Elizabeth Chang, and Ian Sommerville. Software Engineering Ontology for Software Engineering Knowledge Management in Multi-site Software Development Environment. In *10th International Protege Conference*, 2007.

[WE05] David Willmor and Suzanne M. Embury. A safe regression test selection technique for database-driven applications. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 421–430. IEEE Computer Society, 2005.

[Wie99] Karl E. Wiegers. First Things First: Prioritizing Requirements. *Software Development*, 1999.

[Wil99] Clay E. Williams. Software testing and the UML. In *International Symposium on Software Reliability Engineering (ISSRE'99)*. IEEE, 1999.

[WSKR06] Kristen R. Walcott, Mary Lou Soffa, Gregory M. Kapfhammer, and Robert S. Roos. Timeaware test suite prioritization. In *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*, pages 1–12. ACM, 2006.

[XLKB04] Fei Xie, Vladimir Levin, Robert P. Kurshan, and James C. Browne. Translating Software Designs for Model Checking. In *Proceedings of Fundamental Approaches to Software Engineering(FASE 2004), Lecture Notes in Computer Science*, volume 2984, pages 324–338. Springer, 2004.

[XLL05] Dong Xu, Huaizhong Li, and C. Peng Lam. Using Adaptive Agents to Automatically Generate Test Scenarios from UML Activity Diagrams. In *Proceedings of the 12th Asia-Pacific Software Engineering Conference(APSEC)*. IEEE Computer Society, 2005.

[XLL07] Dong Xu, Huaizhong Li, and C. Peng Lam. A Systematic Approach to Automatically Generate Test Scenarios from UML Activity Diagrams. In *Proceedings of the Third IASTED International Conference on Advances in Computer Science and Technology*, 2007.

[XLLP08] Dong Xu, Wei Liu, Zongtian Liu, and Nduwimfura Philbert. Towards Formalizing UML Activity Diagrams in CSP. In *Proceedings of the International Symposium on Information Science and Engieering*, pages 73–76. IEEE, 2008.

[YH07] S. Yoo and M. Harman. Pareto efficient multi-objective test case selection. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, pages 140–150, 2007.

[YHS+99] Kim YG, Hong HS, Cho SM, Bae DH, and Cha SD. Test cases generation from uml state diagrams. *IEEE ProceedingsSoftware*, 146(4):187–192, 1999.

[YHTS09] Shin Yoo, Mark Harman, Paolo Tonella, and Angelo Susi. Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge. In *Proceedings of the eighteenth international symposium on Software testing and analysis, International Symposium on Software Testing and Analysis*, pages 201–212. ACM, 2009.

[YNC03] Y. T. Yu, S. P. Ng, and Eric Y. K. Chan. Generating, selecting and prioritizing test cases from specifications with tool support. *International Conference on Quality Software*, 0:83, 2003.

[YS06] Shuzhen Yao and Sol M. Shatz. Consistency Checking of UML Dynamic Models Based on Petri Net Techniques. In *Proceedings of the 15th International Conference on Computing (CIC '06)*, pages 289–297. IEEE, 2006.

[ZGG07] Carlos Mario Zapata, Guillermo Gonzlez, and Alexander Gelbukh. A Rule-Based System for Assessing Consistency between UML Models. In *6th Mexican International Conference on Artificial Intelligence(MICAI '07) Lecture Notes in Artificial Intelligence*, pages 215–224, 2007.

[ZH05] Hong Zhu and Qingning Huo. Developing software testing ontology in uml for a software growth environment of web-based applications. *Software Evolution with UML and XML*, pages 263–295, 2005.

[ZK01] Andrea Zisman and Alexander Kozlenkov. Knowledge Base Approach to Consistency Management of UML Specifications. In *Proceedings of the 16th International Conference on Automated Software Engineering (ASE'01)*, 2001.

[ZLQ06] X. Zhao, Q. Long, and Z. Qiu. Model checking dynamic UML consistency. In *Formal Methods and Software Engineering, Proceedings of the 8th International Conference on Formal Engineering Methods, ICFEM 2006*, page 440459. Lecture Notes in Computer Science, Springer, 2006.

[ZNXQ07] Xiaofang Zhang, Changhai Nie, Baowen Xu, and Bo Qu. Test case prioritization based on varying testing requirement priorities and test case costs. *Quality Software, International Conference on*, 0:15–24, 2007.

[ZS02] Penglin Zhu and Eckehard Schnieder. Holistic Modeling of Complex Systems with Petri Nets. In *International Conferences on System Management*, pages 3075–3080. IEEE, 2002.

Appendix A

Rules for Consistency Checking

A.0.4 Intra-Model consistency checking rules

The following rules, including those defined in [uml07] are taken into consideration when checking intra-model consistency. OCL equivalent of some rules are shown here.

Use case Model

Rule 1: An actor must have a name.

self.name $\longrightarrow$ notEmpty()

Rule 2: A use case must have a name.

self.name $\longrightarrow$ notEmpty()

Rule 3: An actor can only have associations to use cases, components, and classes. Furthermore these associations must be binary.

Rule 4: An actor must be related to one or more use cases.

Rule 5: Each use case must be used by(associated to) one or more actor.

Rule 6: Use cases can only be involved in binary associations.

Rule 7: Use cases cannot have associations to use cases specifying the same subject.

Rule 8: A use case cannot include use cases that directly or indirectly include it.

not self.allIncludedUseCases() → includes(self)

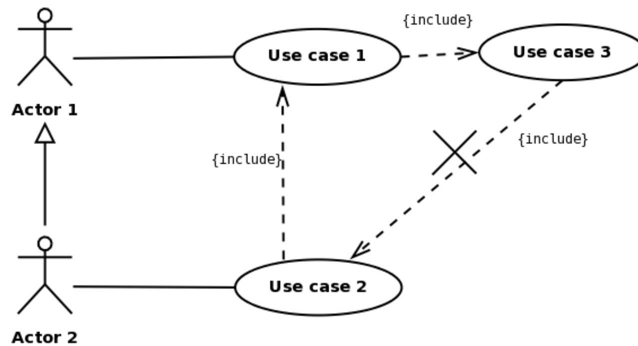


Figure A.1: Diagrammatic representation of Rule 8

Rule 9: An actor and use case can be the source or target of an association.

self.source → forAll(p | p → oclIsKindOf(Actor) or oclIsKindOf(UseCase))

self.target → forAll(p | p → oclIsKindOf(Actor) or oclIsKindOf(UseCase))

Rule 10: No two use cases in a use case diagram can have the same name, unless it refers to the same requirement.

Rule 11: A use case cannot extend use cases that directly or indirectly extend it.

Rule 12: A use case cannot inherit use cases that directly or indirectly inherit it.

Rule 13: A use case cannot inherit another use case if the corresponding actors inherit from one or the other.

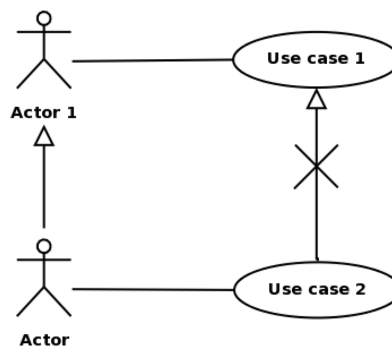


Figure A.2: Diagrammatic representation of Rule 13

Rule 14: An actor cannot be associated to use cases that have an association re-

lation between themselves.

Activity Model

Rule 15: No two activities in an activity diagram can have the same name.

Rule 16: An activity diagram must have an initial node and at least one final node.

Rule 17: Each activity must have an incoming transition and outgoing transition.

Rule 18: Each activity must belong to a class.

Rule 19: Each activity must conform to signature of method corresponding to the activity.

Rule 20: Each activity must be reachable from the initial node.

Rule 21: From each activity, at least one final node must be reachable.

Rule 22: An activity node can be the source or target of information.

context ActivityNode:

(self.source→forAll(p | p → oclIsKindOf(ActivityNode))) or (self.target → forAll(p | p → oclIsKindOf(ActivityNode)))

A.0.5 Inter-Model consistency checking rules

For inter-model consistency checking, we have taken into consideration five diagrams, namely, use case, sequence, activity, class and state diagrams. The rules are:

Use Case - Activity:

Rule 23: For each actor in a use case diagram there must exist a matching class in atleast one activity diagram.

The corresponding OCL constraint is given below:

package Behavioral_Elements

context classifier

invariant rule-23:

let actor =self.feature → select(f|f.oclIskindof(Actor))in

let class =self.feature → select(f|f.oclIskindof(Class))in

⟶ exists(g|g.name=f.name)

Here, f is the set of all actors and g is the set of all classes. f and g are compared to check if they are equal. If equal, then each actor has a corresponding class, indicating that the model is consistent. If not, then the models are inconsistent

Use Case - Class:

Rule 24: Each actor in a use case diagram must find a corresponding class in the class diagram.

inv rule 24:

let actor = self.feature → select(f—f.oclIskindof(Actor))in

let class =self.feature → select(g—f.oclIskindof(Class))

!exists (g|g.name=f.name)

Here, f is the set of all actors and g is the set of all classes. f and g are compared to check if they are equal. If equal, then each actor has a corresponding class, indicating that the model is consistent. If not, then the models are inconsistent.

Use case - Sequence:

Rule 25: For each actor associated to a use case diagram there must exist a matching actor atleast one sequence diagram.

Activity - Sequence:

Rule 26: Each class in an activity diagram must correspond to an object in a sequence diagram.

Rule 27: Each activity in an activity diagram must correspond to a message in a sequence diagram.

Sequence - Class:

Rule 28: Each object and message in a sequence diagram must have a corresponding class and method in the class diagram.

Rule 29 : Each object in a sequence diagram must reference a valid class in the class diagram.

Rule 30: Every message received by an object must correspond to a method of the object's class.

Rule 31: For every message between sender and receiver pertaining to different objects(belong to different classes), there should be a relationship between the sender and receiver class. Also, the association should be navigable from the sender to the receiver.

Activity - Class:

Rule 32: Each class and activity in an activity diagram must have a corresponding class and method in the class diagram.

Rule 33: For every method, if the sender and receiver pertain to different objects(belong to different classes), then there should be a relationship between the sender and receiver class. Also, the association should be navigable from the sender to the receiver.

Rule 34: For every activity between sender and receiver pertaining to different objects(belonging to different classes), there should be a relationship between the sender and receiver class. Also, the association should be navigable from the sender to the receiver.

Class - Sequence:

Rule 35: The public methods of a class in a class diagram must be called in atleast one sequence diagram to depict the interaction of classes.

Class - Activity:

Rule 35: The public methods of a class in a class diagram must be called in atleast one diagram to depict the interaction of classes.

Class - State:

Rule 36: Each class in a class diagram must have a corresponding state.

Rule 37: Each state in a state diagram must be related to one and only one class.

Activity - State:

Rule 38: Each object(class) and corresponding activity in an activity diagram must match with a state(message) of the class in the state diagram.

Sequence - State:

Rule 39: Each message of an object must match with a state(message) of the object's class in the corresponding statechart diagram.