2011

Model based test suite minimization using metaheuristics

Usman Farooq
Edith Cowan University

Edith Cowan University

Copyright Warning

Model Based Test Suite Minimization Using Metaheuristics

by

Usman Farooq

This thesis is submitted for the degree of

Doctor of Philosophy

at the Edith Cowan University

School of Computer and Security Science

July 2011

USE OF THESIS

The Use of Thesis statement is not included in this version of the thesis.

Dedication

To

My Mother

for her endless prays, encouragement and support

that made all of this happen.

Abstract

Software testing is one of the most widely used methods for quality assurance and fault detection purposes. However, it is one of the most expensive, tedious and time consuming activities in software development life cycle. Code-based and specification-based testing has been going on for almost four decades. Model-based testing (MBT) is a relatively new approach to software testing where the software models as opposed to other artifacts (i.e. source code) are used as primary source of test cases. Models are simplified representation of a software system and are cheaper to execute than the original or deployed system.

The main objective of the research presented in this thesis is the development of a framework for improving the efficiency and effectiveness of test suites generated from UML models. It focuses on three activities: transformation of Activity Diagram (AD) model into Colored Petri Net (CPN) model, generation and evaluation of AD based test suite and optimization of AD based test suite.

Unified Modeling Language (UML) is a de facto standard for software system analysis and design. UML models can be categorized into structural and behavioral models. AD is a behavioral type of UML model and since major revision in UML version 2.x it has a new Petri Nets like semantics. It has wide application scope including embedded, workflow and web-service systems. For this reason this thesis concentrates on AD models. Informal semantics of UML generally and AD specially is a major challenge in the development of UML based verification and validation tools. One solution to this challenge is transforming a UML model into an executable formal model. In the thesis, a three step transformation methodology is proposed for resolving ambiguities in an AD model and then transforming it into a CPN representation which is a well known formal language with extensive tool support.

Test case generation is one of the most critical and labor intensive activities in testing processes. The flow oriented semantic of AD suits modeling both sequential and concurrent systems. The thesis presented a novel technique to generate test cases from AD using a stochastic algorithm. In order to determine if the generated test suite is adequate, two test suite adequacy analysis techniques based on structural coverage and mutation have been proposed. In terms of structural coverage, two separate coverage criteria are also proposed to evaluate the adequacy of the test suite from both perspectives, sequential and concurrent. Mutation analysis is a fault-based technique to determine if the test suite is adequate for detecting particular types

of faults. Four categories of mutation operators are defined to seed specific faults into the mutant model.

Another focus of thesis is to improve the test suite efficiency without compromising its effectiveness. One way of achieving this is identifying and removing the redundant test cases. It has been shown that the test suite minimization by removing redundant test cases is a combinatorial optimization problem. An evolutionary computation based test suite minimization technique is developed to address the test suite minimization problem and its performance is empirically compared with other well known heuristic algorithms. Additionally, statistical analysis is performed to characterize the fitness landscape of test suite minimization problems.

The proposed test suite minimization solution is extended to include multi-objective minimization. As the redundancy is contextual, different criteria and their combination can significantly change the solution test suite. Therefore, the last part of the thesis describes an investigation into multi-objective test suite minimization and optimization algorithms.

The proposed framework is demonstrated and evaluated using prototype tools and case study models. Empirical results have shown that the techniques developed within the framework are effective in model based test suite generation and optimization.

List of Published Papers

1. Farooq, U., Lam, C. P., Li, H., “Transformation Methodology for UML 2.0 Activity Diagram into Colored Petri Nets”, Proceedings of the third IASTED Conference on Advances in Computer Science and Technology, 2007. ISBN: 978-0-88986-656-0

2. Farooq, U., Lam, C. P., Li, H., “Towards Automated Test Sequence Generation”. 19th Australian Software Engineering Conference, Perth, WA., pp. 441-450, 25-28th March 2008. ISSN:1530-0803

3. Farooq, U., Lam, C. P., “Mutation Analysis for the Evaluation of AD Models ", International Conference on Innovation in Software Engineering (ISE'2008), December, 2008. (Vennia, Australia, 9-12/12/2008, Publisher: IEEE Computer Society Press)

4. Farooq, U., Lam, C. P., “Evolving the Quality of a Model-Based Test Suite”, International Conference on Software Testing, Verification and Validation (ICST 2009), Denver, USA, 2009

5. Farooq, U., Lam, C. P., “A Max-Min Multiobjective Technique to Optimize Model Based Test Suite”, 10th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD 2009), 2009

Declaration

I certify that this thesis does not, to the best of my knowledge and belief:

(i) incorporate without acknowledgement any material previously submitted for a degree or diploma in any institution of higher education;

(ii) contain any material previously published or written by another person except where due reference is made in the text; or

(iii) contain any defamatory material.

I also grant permission for the Library at Edith Cowan University to make duplicate copies of my thesis as required.

Signed: …Usman Farooq…….

Date: ……30/11/2011…….

Acknowledgements

First of all, I am thankful to Allah, Al-Haadi, for leading me to the path of knowledge and wisdom, Al-Waliyy, for giving me the opportunity and strength to complete my Ph.D.

I am greatly thankful to many people whose help and support made this journey possible. First of all, I would like to thank my supervisor C. P. Lam for continuously supporting and guiding me throughout in my research and for providing valuable feedback on my thesis. Her critical and insightful comments greatly improved my work and presentation of this thesis.

I owe my gratitude to my co-supervisor Huaizhong Li for the useful discussion which helps me in shaping up some of the ideas in this thesis.

Many thanks go to School of Computer and Security Science for providing me the scholarship without which it would be very difficult to finish my thesis.

I am truly thankful to the Paul Nicholson and Paolo Cantoni at Ripple Systems for allowing me to use the ETP and DTP models for case study. I also appreciate their support to continue my research and giving me the opportunity to learn about model driven development in practice.

I owe my gratitude to my PhD lab colleagues for providing motivating and friendly environment. Special thanks goes to Yunous Vagh for many interesting discussions and proofreading various sections of this thesis. I am also very grateful to my friends, Muhammad Abid Akhtar, S. P. Singh, Muhammad Kashif and Ahmad Sattar Khan, for encouragement and support.

Last but not least, many thanks go to my family for their personal and moral support during my studies and they were always there for me. Special thanks to my mother who always supported and encouraged me to whatever degree I want. My brother and sisters for supporting every decision I made.

Table of Contents

List of Figures

List of Tables

List of Acronyms

UML	Unified Modeling Language
AD	Activity Diagram
SD	Sequence Diagram
SM	State Machine
CPN	Colored Petri Nets
FSM	Finite State Machine
SDL	Specification and Description Language
EC	Evolutionary Computation
GA	Genetic Algorithm
ES	Evolutionary Strategies
EP	Evolutionary Programming
MOEA	Multi-Objective EC Algorithm(s)
NPGA	Niched Pareto Genetic Algorithm
NSGA	Non-Dominated Sorting Genetic Algorithm
SPEA	Strength Pareto Evolutionary Algorithm
PAES	Pareto Archived Evolutionary Strategy
VEGA	Vector Evaluated Genetic Algorithm
HCSA	Steepest Ascent Hill Climbing (Algorithm)
HCNA	Next Ascent Hill Climbing (Algorithm)
HCRA	Random Ascent Hill Climbing (Algorithm)
GD	Greedy (Algorithm)
ONVGR	Overall Non-dominated Vector Generation Ratio (metric)
ER	Error Ratio (metric)
S	Space (metric)
SDLC	Software Development Life Cycle
CASE	Computer Aided Software Engineering
MDE	Model Driven Engineering
MDA	Model Driven Architecture
CIM	Computational Independent Model
PIM	Platform Independent Model
PSM	Platform Specific Model
CWM	Common Warehouse Metamodel
XMI	XML Metamodel Interchange
XSLT	eXtensible Stylesheet Language Transformation
MOF QVT	Meta Object Facility Query/View/Transformation

MBT	Model Based Testing
TTCN	Testing and Test Control Notation
SUT	Software Under Test
ECCS	Enterprise Customer Commerce System
ATM	Automated Teller Machine
ETP	Edit Trend Properties
DTP	Delete Trend Properties
RW	Random Walk (Algorithm)
MCDC	Modified Condition/Decision Coverage
TSM	Test Suite Minimization
TCS	Test Case Selection
TCP	Test Case Prioritization

Chapter 1 Introduction

Testing, like in other manufacturing and engineering processes, is an integral part of the software development process. Generally, it consumes thirty to fifty percent of the software development budget, resources and time [Beizer 1990; 1979]. Furthermore, in case of life and safety critical systems, it accounts for an even significantly higher percentage. Although testing is critical for controlling software quality, it is often largely neglected or is performed inefficiently [Beizer 1990; Butler 2004]. As software is getting ubiquitous, there is an increased dependence upon the features and services provided by software. Ironically, software systems are far too often failing due to anomalies and defects, thereby causing severe problems and damages, costing lives, reputations and fortunes. In 2005, Toyota had to recall 160,000 of its hybrid Prius cars due to faults in its embedded software [Garfinkel 2005]. In another incident, the lack of a robust testing procedure contributed to the death of five patients and injuries to others by a radiation therapy device [Huckle 2005]. A 2002 study conducted by the National Institute of Standards and Technology reported that software bugs cost the USA's economy around $59.5 billion annually and, more than a third of that cost - about $22.2 billion, was caused by ineffective testing [Tassey 2002]. The report further concluded that the lack of rigorous and robust testing during and after software development usually contributed to its failure. The problem is universally persistent in software projects irrespective of its geographical location, scale and type of software and developing organization [Tan 2009]. Software projects often running under pressure of limited time and budget, are prone to compromises in the testing effort and program quality. Furthermore, poor program quality can generally be ascribed to inadequate and ineffective testing owing to a broad reluctance to accept robust techniques that may be highly expensive and lacking direct and obvious value for the project (or to their customers).

Software testing remains a tedious, complex and error-prone process as a consequence of the latest tools and technologies perpetually trying to match the ever growing size, functionality, complexity and heterogeneity of software systems. The capability for delivering high quality software under competitive pressure and tight schedules has become the strategic factor for software organizations. Spending too much time or money on unnecessary testing can possibly

result in late delivery and wastage of resources. On the other hand, premature software release with inadequate testing to meet deadlines, or to cut costs and minimize effort, may lead to accidents, injuries, and loss of time or data due to undetected faults. Software organizations, similar to the challenges faced in other industries, face an ongoing struggle between the aim for higher quality production and the desire to reduce both costs and time-to-market (release cycle) [El-Far and Whittaker 2002]. Testing always needs a significant investment in terms of both time and effort. To remain competitive in the software business, organizations need to improve their testing activities resulting in higher productivity (performance) and better value.

1.1 Motivation

Exhaustive testing is impractical and infeasible due to the infinite number or combinations of inputs or paths even in the case of a small software program [Beizer 1990; Binder 1999; Myers 1979]. One of the approaches in addressing this problem is the selection of an adequate set of effective test cases according to a given test criteria. However, as testing is often performed under limited time and resource constraints in practice, the questions of the amount of testing that is adequate, or when testing should stop is raised. One option is to stop testing when time or resources run out [Pressman 2001]. However, as this approach does not ensure the effective and efficient use of time and resources, it has several implications. For instance, a critical module could not be tested as the allocated testing time has exhausted or too much time was spent on redundant test cases. Myers [1979] and Kaner [1997] argued that, owing to limited time and resources, a less but reasonable or 'good enough' testing should be done.

For effective and efficient use of testing resources, several test suite optimization approaches [Jones et al. 1996; Jones et al. 1998; Michael et al. 2001; Pargas et al. 1999; Tracey et al. 1998; Tracey et al. 1998] have been proposed. Generally, these approaches can be classified into two categories viz, the optimal test suite generation [Shiba et al. 2004; Sthamer 1996; Wegener et al. 2001] and the optimization of pre-generated test suites [Elbaum et al. 2001; Jones and Harrold 2003; Offutt et al. 1995; Rothermel et al. 2001]. In the former case, given the fact that exhaustive testing is infeasible, the focus is to generate an adequate set of test cases according to some given criteria, whereas in the latter case, the focus is on enhancing the efficiency and/or effectiveness of a pre-generated test suite. This is done by prioritizing or selecting a subset of test cases based on information (e.g. usage profile, complexity or risk) collected during the analysis, design or previous execution of the system as selection, prioritization or reduction criteria. Both types of techniques can be further categorized into

code-based and model-based optimization techniques. Code-based techniques derive test cases using source code based information. Whereas, model-based techniques rely on models of a system under test for test suite reduction, test case selection or prioritization.

Optimization of test suites with model-based techniques is relatively inexpensive in comparison to code-based techniques [Bogdan et al. 2005]. Bogdan et al. [2005] argued that the execution of a system model is cheaper than full system execution, and therefore, the smaller optimization overhead of model-based techniques makes it more feasible than the code-base techniques. Another recent study [Bogdan and Koutsogiannakis 2009] found that the model-based prioritization techniques (a form of test suite optimization) are even more effective in fault detection than the code-based prioritization techniques.

Model based testing (MBT) has been advocated for its several advantages including the potential for decreasing the cost by reusing and sharing the model artifacts, scaling up the testing process to large and complex systems, and the capability to accommodate the change with minimum cost and effort [El-Far 2001; Utting and Legeard 2006]. Incorporating models in the development process enables the use of various verification and validation techniques such as simulation, testing and formal analysis to ensure the correctness of the software. Late fault detection and correction is expensive and time-consuming. In conventional code-based techniques, testing starts after the coding phase which not only misses the requirement and design defects but also increases the cost. MBT allows testing to start earlier in the development process, thereby enabling faults to be detected sooner and fixed for cheaper. Test generation is the most expensive and crucial activity in the testing process. In MBT, the underlying formal structure of a modeling technique facilitates automation of the test generation process making it quicker and simpler to generate test cases according to the given criteria. Several academic and commercial MBT tools are already available [Utting and Legeard 2007]. Furthermore, considerable research and industrial experiences have confirmed the success and benefits of MBT [Bogdan and Koutsogiannakis 2009; Bouquet and Legeard 2003; Briand et al. 2004; Dias Neto and Travassos 2009; Farchi et al. 2002; Florin 2008; Jonas 2008; Kansomkeat et al. 2008; Pedro et al. 2008; Pretschner et al. 2005; Stobie 2005].

Several studies have also shown [Jones, Eyres and Sthamer 1998; Li et al. 2007; McMinn et al. 2006; Wegener et al. 1997; Yoo and Harman 2007] that incorporating optimization techniques in the testing process can significantly enhance the efficiency of a test suite without compromising its fault detection capability. Optimization is referred to as the process of finding the best solution from the set of available values under a given objective. According to Corne et al. [1999], an optimization problem is a class of problems for which there exist a number of

different possible solutions with respect to some quality criteria. Mathematically, the optimization problem can be defined as a function for finding the minimum or maximum possible value. Some types of optimization problems are simple to solve, e.g. a problem with few candidate solutions will have a small solution space which can be searched exhaustively and where the best solution can be revealed in a reasonable amount of time. However, not all optimization problems are easy to solve as the search space is so large that an exhaustive search of all possible solutions is practically impossible. Such problems are formally classified as NP-hard, which means that there is no algorithm that can solve them in polynomial time. Since it is unlikely that there exists any exact algorithm that can solve these computationally hard problems in a feasible time space, one approach is to take the constrained version of a problem and then obtain the optimal solution using an exact algorithm. Another approach is to solve the original problem using an approximation algorithm, also known as a heuristic algorithm, in a significantly reduced amount of time, however at the cost of a guarantee of finding the actual optimal solution. Nevertheless, several studies [Aguilar 2001; Dorigo and Gambardella 1996; Goldberg and Samtani 1986; Grefenstette et al. 1985; Prins 2004] have shown that these heuristic techniques can find the optimal solution for such NP-hard problems in finite (reasonable) time, if they have been customized or tailored to the problem.

Metaheuristic techniques are a class of Artificial Intelligence (AI) techniques which are used to find near-optimal solutions with the help of a particular cost (or objective) function. They are generally considered suitable for dealing with very large search spaces and non-linear constraints. They are also adaptable and powerful, with the ability to obtain good solutions for many extremely difficult problems. Their flexibility is due to the fact that the search is simply directed by a cost function, and very little other problem specific knowledge is required [Corne, Glover and Dorigo 1999]. The successful application of metaheuristic techniques in many engineering, business, science and management areas showed that they are capable of producing near-optimal solutions for problems that are computationally hard to solve exactly in polynomial time.

Evolutionary Computation (EC) is a metaheuristic technique which is inspired by the evolution process and characterized as Genetic Algorithms (GA) by Holland [1992] and Evolutionary Strategies (ES) by Rechenburg [1973]. Evolution by natural selection has proven to be an effective search process and has successfully been applied to various research and application fields such as combinatorial optimization, planning and scheduling, industrial design, machine learning and pattern recognition. The potential of Evolutionary Computing (EC) in the realm of Software Engineering has been investigated for software testing [Jones, Eyres and

Sthamer 1996; Lakhotia et al. 2007; Li, Harman and Hierons 2007; McMinn 2004; McMinn, Harman, Binkley and Tonella 2006; Michael et al. 1997; Patton et al. 2003; Roper 1996; Watkins 1995; Zhan and Clark 2005], next release problem [Zhang et al. 2007], optimal service composition and deployment [Yves et al. 2008] and software understanding [Reynolds et al. 1994].

1.2 Aims and objectives of this thesis

In general, the aim of test suite optimization is to find one or more minimized or prioritized combinations of test cases in such a way that by selecting and executing one of them will either reduce the testing time and cost without compromising the required coverage level of test requirements (i.e. mutation score and branch coverage) or increase the rate of test requirement fulfillment (i.e. fault detection rate). In order to determine a credible and valid measure of counteracting the test suite optimization problem, and taking into account the characteristics of metaheuristic techniques, the overall objective of the study was to investigate the potential application of metaheuristic techniques into model-based test suite optimization. This involved exploring solutions for problems, developing specific tools and empirically analyzing them. The objectives of this thesis are as follows:

1) **Develop an automatic transformation of a UML behavioral model into an equivalent and valid executable model.**

   The key advantage of model-based testing over other testing methodologies is the fact that model execution is cheaper than full software system execution. Although, Unified Modeling Language (UML) has become the lingua franca for software modeling in academia and industry, the models developed in it are not executable. The first and foremost objective is to develop an automatic mechanism for transforming a UML behavioral model into an equivalent and valid executable model.

2) **Develop an automatic model-based test generation and evaluation technique.**

   Software requirements and design methodologies have seen major changes in the last two decades due to the high cost and impact of the errors introduced at these stages. UML provides several visual modeling formalisms (languages) for capturing requirements and intended behavioral depiction. These visual languages are very attractive and user-friendly. Moreover, they enable the generation of complete, consistent, and unambiguous specifications of system behavior that can be used for the test generation purpose. One of the main requirements of the model-based testing is the ability to directly and automatically generate test suites from software models.

Another essential and pertinent requirement is to determine the adequacy of a generated test suite. Therefore, it is important to investigate and develop test generation techniques and evaluation/adequacy criteria for the selected modeling formalism.

3) **Investigate the application of metaheuristic in model-based test suite optimization.**

The key to the application of metaheuristic techniques to a problem is to formulate it as a searchable problem, which means that an appropriate solution representation and a cost or fitness function relative to the desired objective or criterion need to be defined. The heuristic seeks a 'better' solution according to the cost/fitness function through the search space, which represents the set of all possible solutions. Therefore, suitable representation and good cost/fitness functions are critical to the success of the search. Although, metaheuristic techniques are generally considered generic techniques and can find the optimal solution without needing much domain specific information, they usually need adaptation for better performance. Therefore, in order to apply metaheuristic techniques to model-based test suite minimization, it is important to cast the test suite minimization first as an optimization problem and then to adapt the selected heuristic strategies.

4) **Investigate the multi-objective model-based test suite optimization using metaheuristic.**

The last but equally important objective to model-based test suite optimization[1] is to incorporate multiple criteria into the optimization process. The multi-criteria optimization of model-based test suite is important to testing for practical reasons. In optimization, solutions are considered good or bad according to the given criterion. However practical optimization problems often involve more than one criterion or factor and no one solution can be rendered optimal according to multiple criteria. As mentioned earlier, a tester may need to consider multiple factors in testing in order to perform effective and efficient testing. In this scenario, finding the optimal test suite according to all of the required factors is more appropriate than finding an optimum for each of the factors separately.

[1] Throughout the thesis, the term minimization is interchangeably used with optimization and whenever the term test suite optimization is used it means test suite minimization except where other usages are stated explicitly.

1.3 Contribution of this thesis

The central insight in driving this study is the fact that model-based test suite optimization has several merits. These include such things as cheaper model execution as opposed to whole system execution, earlier availability of models in the software development process which eliminates the need to wait for the completion of coding, as well as early fault detection of the separate models, value adding to the testing process through linking of tests to requirements and the convenience of being able to incorporate multiple factors (i.e. usage and risk information) into the optimization process [Bogdan et al. 2007; Bogdan and Koutsogiannakis 2009; Jonas 2008]. As the purpose of the study is to investigate the application of metaheuristic techniques in model-based test suite optimization, a framework for UML-based test suite optimization is developed and demonstrated with an Activity Diagram (one of the UML modeling formalisms) and an Evolutionary Computation. The incorporation of metaheuristic techniques in test suite optimization will lead to more effective and efficient testing by finding the near-optimal combination or arrangement of test cases. Given the fact that test suite optimization is a computationally difficult problem [Harman and Jones 2001; Harrold et al. 1993; Rothermel and Harrold 1993], the typical analytic techniques are not always viable, as these techniques cannot always successfully find the optimal solution under limited available resources and timeframes. Hence, the general aim is to integrate the automated model-based test generation techniques and optimization techniques in a framework so that the tester or project manager can produce and use one or more efficient and effective test suites by selecting or prioritizing test cases according to the value, risk, time and/or budgetary considerations.

Description of the Model-based Test Suite Optimization Framework

The Integrated Framework for Model Based Test Generation and Optimization (IFMBTGO) framework contains two processes (Figure 1-1). Firstly, a model based test generation which takes a UML based model as an input and derives test sequences from it, and secondly, an optimization process for enhancing the efficiency and effectiveness of the generated test suite. During the test generation process, a UML based behavioral model is explored automatically with respect to a given test requirement. This enables the tester to record the test sequences from a design model. The test suite (set of test sequences) is then analyzed in order to decide whether they are adequate according to the pre-defined evaluation metrics (e.g. node coverage and mutation score). If this is the case, then it passes the test suite onto the optimization process, otherwise more test sequences are generated using an alternate model exploration/execution technique. This process is depicted in Figure 1-2. A number of prototype

tools were developed to demonstrate the vital building blocks of the proposed test suite optimization framework. Four case study Activity Diagram-based models of various sizes and complexities were used for empirical study and the various experimental results were statistically analyzed. The following is a summary of major contributions of this thesis:

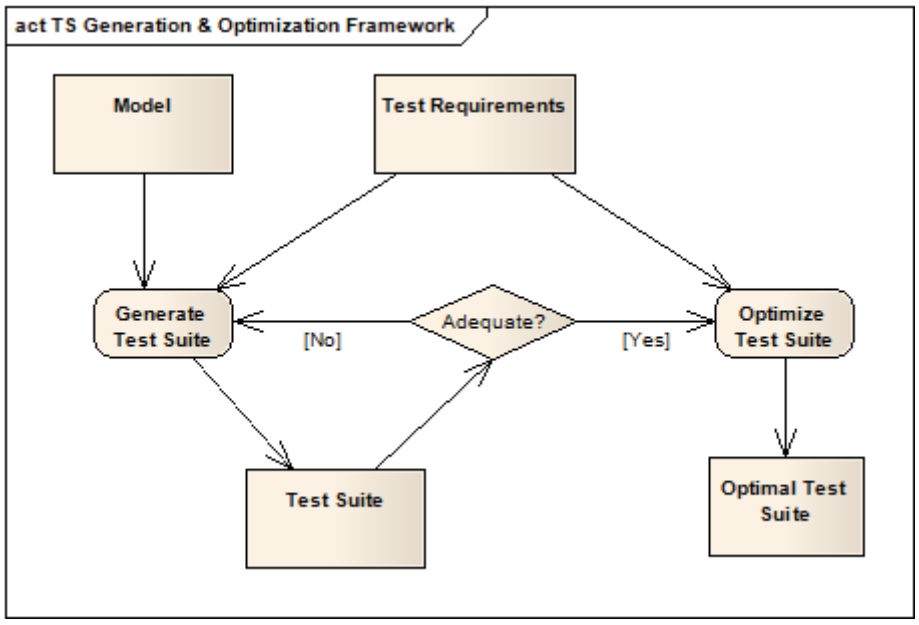


Figure 1-1: The high level depiction of test generation and optimization process underlying the IFMBTGO framework

1) The thesis proposes a model transformation methodology for automatic and seamless transformation of AD models into Colored Petri Net (CPN) models while addressing the common sources of ambiguities through transformation and simplification rules [Farooq et al. 2006]. Modular model transformation of AD has never been performed before and this thesis particularly addresses modular model transformation of AD models. The proposed transformation methodology is substantiated by a transformation tool which takes the AD models of case studies in XMI format as shown in Figure 1-2 and produces the equivalent CPN model also in a valid XML according to the CPN Tools schema [Jensen 1997]. The significance of AD into CPN transformation is that it will enable the static and dynamic analysis (i.e. state space and simulation) of a model depicted in AD.
2) Generating test cases from a model of the software system is a key task in model-based testing. A number of AD-based verification and validation techniques have been reported in the literature (for detail about these techniques see the Chapter 2 and 4). However, due to the significant changes in AD2 (version 2.x) as compared to AD1

(version 1.x), most of the existing AD-based testing techniques have been outdated. A novel automated technique is proposed in the thesis to devise test sequences from an AD model by executing its CPN version according to the given test requirements. The AD based TSG is significant due to the fact that AD has wide application scope, ranging from high level business processes and distributed system modeling to low-level embedded systems modeling.

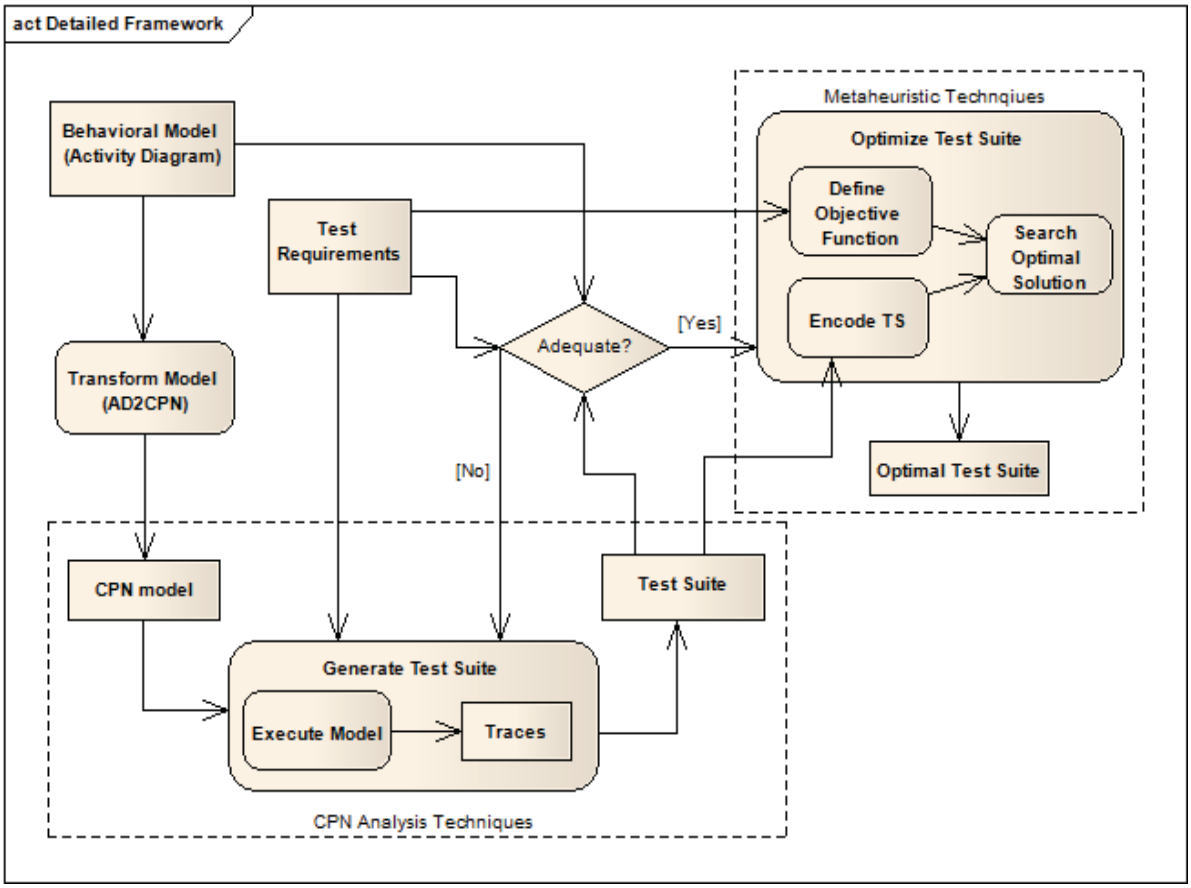


Figure 1-2: Detailed sketch of the IFMBTGO Framework

Generally, coverage analysis is performed in order to determine the adequacy of the test suite according to some given criteria. Similar to code-based testing, the test suite in MBT is also generated with the aim of providing maximum coverage of the system under test [Utting 2005]. However, during the test case generation process the model-based coverage adequacy metric is used according to the source model. In the literature, a large number of coverage criteria have been suggested for model-based testing. However, most of them are not applicable to AD models. This study developed two classes of coverage metrics specifically for AD based test suite evaluation, namely sequential and concurrent coverage criteria. The two proposed coverage metrics will

allow the systematic analysis of the coverage information for AD-based test suites and for assessing its adequacy according to a given coverage criterion.

Mutation analysis is a promising testing technique, and empirical studies have already confirmed its effectiveness in gaining the confidence in the correctness of the program as well as the adequacy of the test suite. In this thesis, the mutation analysis for AD-based behavioral models and test suite adequacy is proposed. The contribution of this work is twofold. Firstly, it defines mutation operators based on the faults patterns defined in [Farchi et al. 2003; Li et al. 2006] for mutation analysis of the AD models, and secondly it introduces mutation analysis for the AD-based test suite adequacy. The significance of this technique at the design level for an AD is that it enhances the confidence in design correctness by showing the absence of the potential or actual faults. Moreover, it will provide an automated analysis technique for the AD models that are often undervalued for their informal semantic and the lack of automated analysis tools.

A CPN-based model execution and trace recording technique is developed and reported in [Farooq et al. 2008]. The test suites are generated from the case studies models using the algorithm and evaluated with the proposed coverage metrics and mutation analysis for adequacy as shown in the Figure 1-2. If a generated test suite is found adequate, it is passed onto the next phase otherwise, more test cases are generated.

3) Kaner et al. [1993] argued that 'good test cases' need to be unique and non-redundant in order to avoid the wastage of resources and time. A novel metaheuristic-based, model-based test suite minimization technique is proposed and demonstrated with four types of metaheuristic techniques. In order to study the comparative performance of the heuristic techniques and other factors on test suite reduction, an empirical study was performed and results were partly reported in [Farooq and Lam 2009].
4) Most of the real-world optimization problems involve more than one decision parameter, and often the good tradeoffs are searched for, in competing constraints. For these types of problems, more than one equally good solution usually exists. Choosing the best one always depends upon the application context. Therefore, in typical multi-objective optimization problems, all of the possible solutions represent some sort of trade-off relationship between the objectives. A novel Evolutionary Computation (EC) based multiobjective optimization approach is proposed to incorporate multiple criteria into model-based test suite optimization. The approach is flexible and can be extended

to other modeling languages and can incorporate more objectives. Furthermore, it shows that the convenience of being able to incorporate various types of information, i.e. requirements and usage profile in the optimization process, gives model-based techniques an extra advantage, which may enable multi-objective test suite optimization without sacrificing the integrity and thoroughness of the test suite.

1.4 Structure of the Thesis

The subsequent chapters of this thesis are:

Chapter 2 —Literature Review. This chapter provides the background information for the thesis. It examines the model-based testing, modeling techniques, model-based test generation and evaluation criteria. It also reviews the evolutionary computation, important algorithms and basic building blocks of the technique. Finally, it reviews the search-based testing and shows how metaheuristic techniques have been applied in test suite optimization.

Chapter 3 —Model transformation: This chapter firstly introduces the case study models and then showcases the common issues related to the ambiguities in the AD models. It presents the transformation rules between AD and CPN and subsequently shows how an AD model can be transformed into CPN model.

Chapter 4 —Model-based testing: In this chapter, a model-based test sequence generation algorithm is presented in order to derive test suites from the case study models introduced in Chapter 3. Two classes of coverage criteria and mutation analysis are also presented.

Chapter 5 —Test suite optimization: This chapter shows the test suite minimization problem as an optimization problem and reformulates it as the well-known knapsack problem. This involves using an evolutionary computation technique followed by comparisons with other types of heuristic algorithms. The heuristic-based optimization is shown to apply to various test suites of different sizes and composition. A fitness landscape analysis is performed on the test suite minimization problem in order to characterize its fitness landscape. Furthermore, it presents the parametric recommendations for the evolutionary computation with regards to the test suite minimization problem.

Chapter 6 —Multiobjective test suite optimization: Multiobjective optimization problems are often cited as difficult but natural to practical problems. This chapter shows how multiple criteria can be incorporated into model-based test suite minimization which can then be formulated as a multiobjective optimization problem and resolved through multiobjective evolutionary computation algorithms.

Chapter 7 —Conclusions: This chapter concludes the thesis research, examines the achievements and limitations of the research reported in this thesis and identifies directions for future research.

Appendix A — Supporting Material: This section contains the CPN-models of the case study models and AD to CPN transofrmation templates.

Appendix B — Supporting Material: This section contains the templates of the mutant operators for the generation of mutant models.

Chapter 2 Literature Review

This chapter reviews the work related to automated model-based software testing through search based techniques. The aim of software engineering is to build and maintain high quality software products with effective and efficient development processes. Typically, software projects are constrained by limited time and resources. Therefore, a trade-off is inevitable for testing process according to a given test objective. Given the fact that testing is the most expensive and laborious activity in software development life cycle (SDLC), it is necessary to determine the most efficient use of the test budget. Selecting a small yet adequate subset of test cases according to a given objective could potentially save test resources and improve the efficiency. The optimization of testing is based on the premise that effective and efficient testing can not only meet the test objectives but also save the cost of testing.

2.1 Model driven engineering

Model Driven Engineering (MDE) is an emerging software development approach that relies on abstraction and automation in order to achieve several significant benefits including reduction in development and maintenance cost, improvement in software quality and support for controlled evolution of the software. Studies indicate that MDE not only reduces the potential faults in the system by 80% but also enhances the productivity by the factor of four [Selic 2008]. MDE has been applied to many software domains, such as real-time and embedded systems, telecommunication systems, and, more recently, to the development and integration of enterprise information systems. MDE is a model-centric development approach where models are used to specify, develop, analyze, verify and manage the software system at a higher level of abstraction than the traditional development approaches.

Modeling is essential for dealing with the complexity and enormity of real-world systems. The Model-Driven Architecture (MDA) is a multi-tier modeling approach fostered by the Object Management Group (OMG) for addressing the complexity of the software development and maintenance process. A common way of development using MDA is by describing the situation of interest in the world, i.e. business process or workflow as a conceptual model formally known

as Computational Independent Model (CIM). This is followed by the definition of a solution model also known as Platform Independent Model (PIM) for the software to specify its content, structure and behavior under the assumption of an ideal computing platform. Using model transformation, one or more implementation models (a.k.a. platform-specific models (PSMs)) considering all the computing platform specific constraints and limitations are obtained from the PIM. In this context, CIMs are a conceptual representation of the domain using the vocabulary that is familiar to the practitioners of the domain, while PIMs are the specification of the software and PSMs are the implementation model. So in the MDA approach, models are reusable artifacts that consolidate the software development effort.

One of the main advantages of model-driven development approaches is the provision of a conceptual structure which defines the mapping between the abstract and detailed models. The chain relationship between models from higher to lower levels not only allows validating the implemented system but also enables the automatic transformation of models. Model transformation is an essential mechanism in MDE for automatic model to model transformation and to generate other key artifacts such as code and test cases from the design models. The model-based testing is a promising approach for software testing and is getting popular due to the inherent advantages of MDE and fault detection effectiveness. The focus of this thesis is model-based testing and in the following section it is elaborated further.

2.2 Model-based testing

The complexity and ever growing size of software systems has made testing even more difficult and challenging. Models are easier to develop and analyze than the original system and therefore are often used for design verification and validation in many engineering disciplines. A model is a simplified, abstract, conceptual and/or graphical representation of a system (or a component of a system). It allows the user to visualize, simulate, analyze and gain understanding about the depicted components of a system and their relationships or their behavior.

According to the McGraw-Hill Dictionary of Scientific & Technical Terms, a model is “a mathematical or physical system, obeying certain specified conditions, whose behaviour is used to understand a physical, biological, or social system to which it is analogous in some way” [McGraw-Hill 2003]. Another definition in the context of computing is given in the Web Dictionary of Cybernetics and Systems and it states that a model is

a system that stands for or represents another typically more comprehensive system. A model consists of a set of objects, described in terms of variables and relations defined on these and either (a) embodies a theory of that portion of reality which it claims to represent or (b)

corresponds to a portion of reality by virtue of an explicit homomorphism or isomorphism between the model's parameters and the given Data. [Heylighen 2010]

Generally, a model is used to depict a particular aspect or view of the system. In a software engineering context, software models depending on the depicted aspect of the system can be classified into two types, viz. structural and behavioral. Structural models depict the structural properties of the system i.e. a set of components and relationship between these components. As they represent the fixed aspects of the system, they are also referred to as static models. Models that describe the operational or stimulus response characteristics of a system i.e. state transition or control flow are referred to as behavioral models or dynamic models. A model needs to be a true but simpler representation of the system. Stachowiak gave a definition of a model which is more commonly accepted in the scientific community wherein he described three key properties of a model [Stachowiak 1973]:

- Representation: A model is a representation of something [Stachowiak 1973, page 131].
- Simplification: Abstraction is one of the approaches for handling complexity. Therefore model should not show all the attributes of the original [Stachowiak 1973, page 132].
- Pragmatic: A model exposes a particular view of the original and fulfils a certain objective.

One of the objectives of software testing is to detect as many bugs as possible. In order to achieve this objective, it is necessary for a tester to fully understand the purpose and functionality of the system under test. With the growing size and complexity of software systems, it is increasingly hard for a tester to comprehend and test the whole system. Therefore, Binder [1999, Page 111] suggested that models be used in testing instead by stating that “We cannot test without first understanding what the implementation under test is supposed to do. The complexity of software requires development of models to support test design.”

Once the knowledge of a system is understood and captured in a model, it can be used to test the correctness and conformance of the software [Apfelbaum and Doyle 1997; Dalal et al. 1999; Robinson 1999]. As models typically express the required features and functionalities, they become the reference points in model-based testing. Therefore, Beizer [1990] noted that the art of testing involves creating, exploring and revising test models. Models are fundamental to software testing as testers usually use mental picture about the system to develop test cases [Binder 1999]. In manual testing, an informal mental model is used to generate test cases;

whereas in model-based testing an explicit model is systematically used to generate test cases and evaluate the test quality.

2.2.1 Model-based testing workflow

Model-based testing is generally refer to as an approach that leverages the model of a software under test in key testing activities such as test generation, execution and evaluation. Automation of such activities is fundamental to the successful application of MBT. Generally, the model-based testing is defined as a process comprising of the following three main automated tasks as shown in Figure 2-1.

1. Model development: The first and foremost step of MBT is developing a model of system under test with sufficient information for testing. As no single model type can specify all aspects of a system, it is necessary to choose an appropriate model type (e.g. state-based and control-flow). Usually, there are multiple modeling notations or languages available for each model type with different express-ability level and tool support. Several selection criteria such as type of the system under test and the focus of testing have been suggested as guidelines for choosing an appropriate model type and notation. For more detail and guidelines on selecting suitable model type, please see [El-Far and Whittaker 2002; Utting and Legeard 2006].

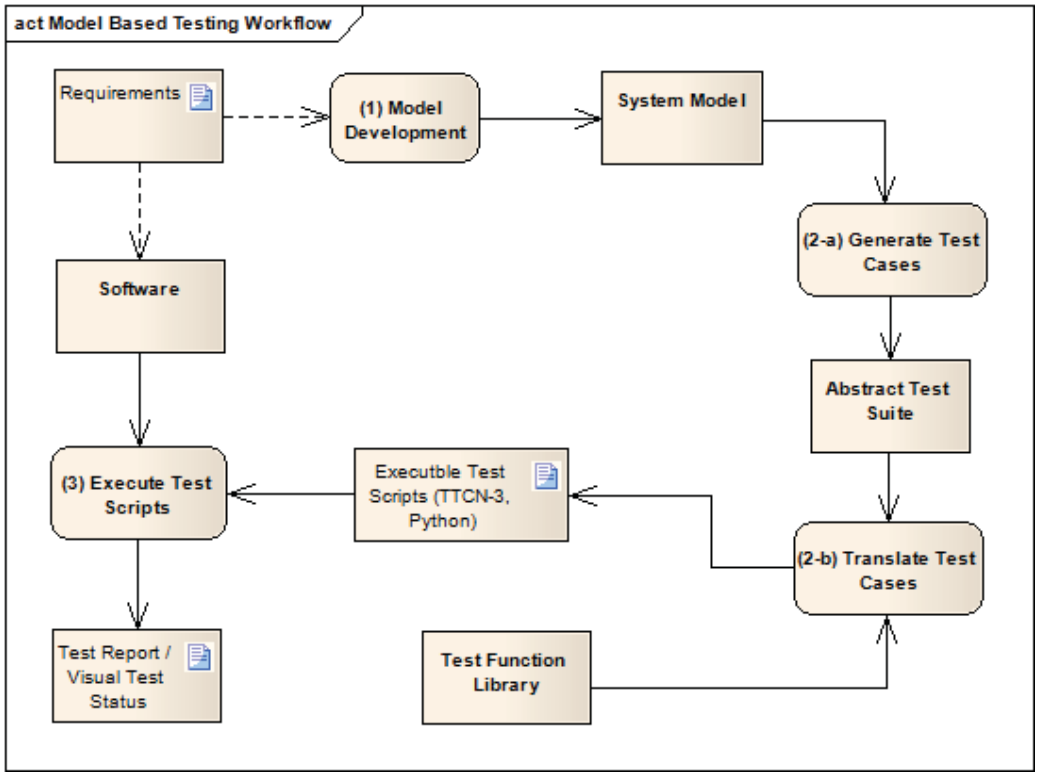


Figure 2-1: Model-based testing workflow (adapted from [Utting and Legeard 2006])

2. Test generation: Test generation is one of the most complex and tedious activities in the testing process. In MBT, this issue is addressed through automatic test generation by leveraging the theoretical background of most of the modeling formalisms. Usually, the models exist at different levels of abstraction than the software under test, so the task of model-based test generation is usually performed in two steps.
    a. Abstract test case generation: In the first step, abstract test cases are derived from the model of the system under test according to some given selection criteria. The abstract test cases also referred as test sequences, represent the threads of token-flow in the system. The term token flow is used to abstractly refer to the different types of test sequences such as control-flow, object-flow and state-flow in the different types of behavioral models.
    b. Concrete test case generation: In the second step, concrete test cases are generated from the abstract test cases in an executable script format using a transformation mechanism. There are numerous general purpose languages (e.g. Java, C#, Python and Perl) and test-specific languages (e.g. TTCN-3) that have been used for executable test scripts.
3. Test execution and evaluation: Similar to a typical testing process, test scripts are executed on the system under test and the results are recorded. An important characteristic of MBT is that the test scripts can be executed online or offline. In the online mode, they are executed as they are generated and the result is analyzed on-the-fly and fed back to the test case generation process. Whereas, in the offline mode, the generated test scripts are executed after all the test cases have been generated and the results are recorded for analysis separately.

Several types of behavioral models, test generation techniques and test execution mechanisms associated to model-based testing have been reported in literature. Synthesis of these works have appeared in [Broy et al. 2005; Neto et al. 2007; Utting and Legeard 2006; Utting et al. 2006].

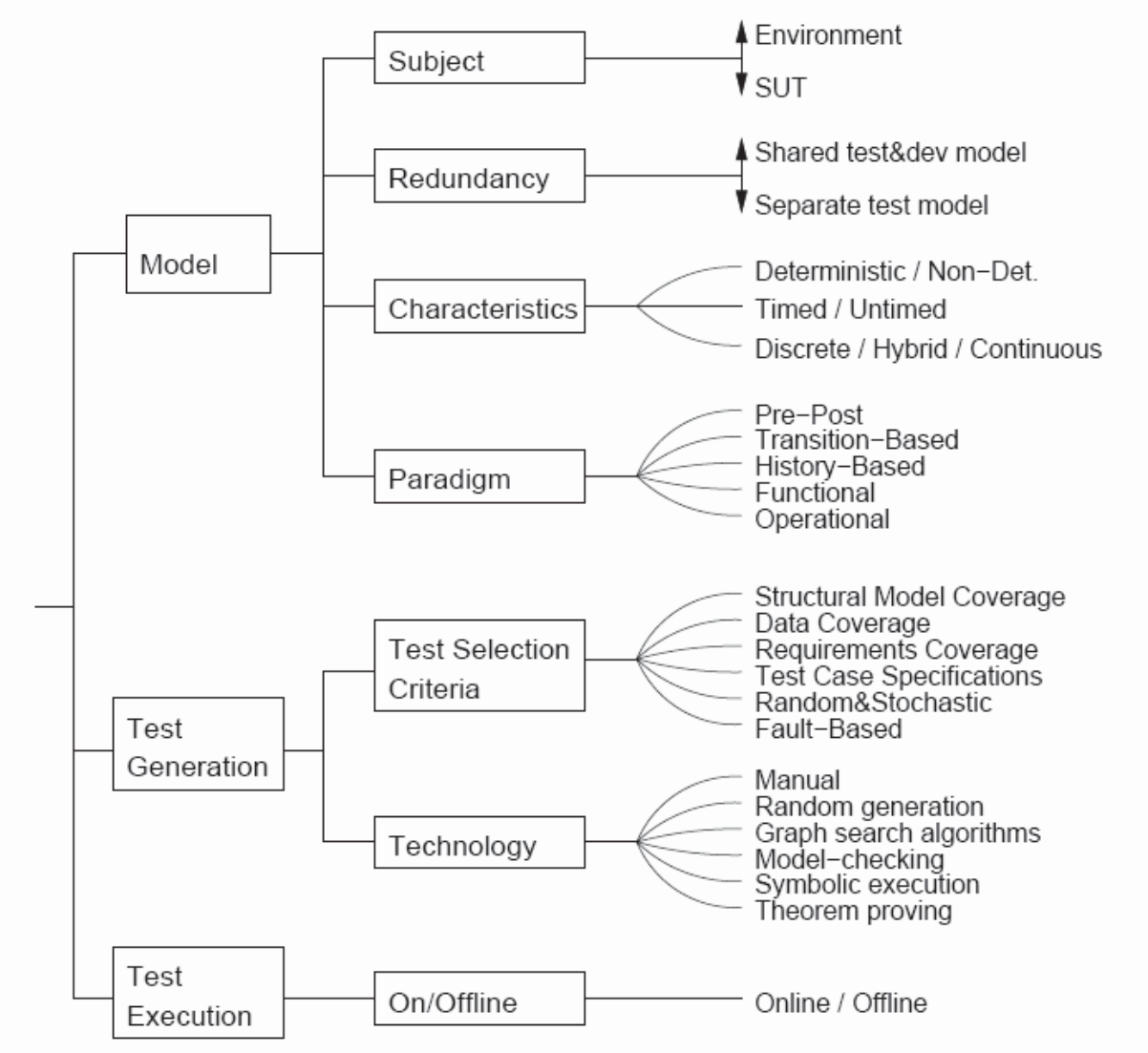


Figure 2-2: Taxonomical overview of model-based testing [taken from [Utting, Pretschner and Legeard 2006]]

Utting et al. [2006] argued that model-based testing techniques have some specific characteristics that can help to understand the strengths and weaknesses of a particular technique. They defined a classification of model-based testing techniques based on seven different dimensions as shown in Figure 2-2. Accordingly, the subject of a model (i.e. system, environment or both) can affect the test suite as system oriented information are used to generate test cases whereas the environment related information can help to identify the valid test scenarios. So the subject or focus of the model is one dimension for categorizing the applications of MBT. Pretschner and Philipps [2005] elaborated four different configurations of test and development models for model-based testing. Using a common or separate model for both testing and development indicate the redundancy in modeling. So the level of redundancy is another dimension to classify the instances of MBT. Non-determinism, temporal constraints,

and the continuous or discrete nature of a model are some of the key characteristics that can be used to differentiate different MBT techniques or their applications. Several model paradigms have been defined for describing the system behavior. Statecharts, Z and Petri Nets are some of the well known examples of Pre-Post, Transition-based and Operational modeling notations respectively. Similar to conventional code based testing, test selection criteria and technology are the two main test generation factors to classify MBT techniques. Test selection criteria and technology are elaborated further in Section 2.2.4 and Section 2.2.5. As model-based test cases can be executed offline or online, the mode of test execution is another dimension used to characterize a MBT technique.

2.2.2 Advantages of model-based testing

Modeling is an effective and economical approach for addressing the complexity and enormity of software systems during their development and maintenance phases. The incorporation of models in the testing process improves it in a number of ways, including the earlier fault detection, automation, greater reusability and higher level of coordination between design and testing activities.

The availability of models in the earlier stages of the SDLC enables the commencement of testing activities from the earlier stages where the 'bug fixing' can have maximum benefits. Studies have shown that faults are cheaper to fix soon after they occur rather than at later stages [Perry 2006]. Fault detection effectiveness is critical to any software testing technique. El-Far and Whittaker studied the use of models in software testing and noted that system specifications depicted in the form of model are effective for both test case generation and fault detection [El-Far and Whittaker 2002]. Recent studies in both commercial and academic environments have also confirmed the fault detection effectiveness of model-based testing [Jonas 2008; Paradkar 2005; Pretschner, Prenninger, Wagner, Kuhnel, Baumgartner, Sostawa, Zolch and Stauner 2005; Stobie 2005].

Table 2-1 shows the relative cost of fixing a fault relative to the stage when it was detected. Unfortunately, requirements and specifications are a major source of software bugs (see Figure 2-3). Studies have found that in some cases the proportion of such bugs to the overall detected bugs can be 50% or more [Beizer 1990; Perry 2006]. A more important fact is that the cost of fixing such faults at the later phases can be up to 100 times more than that at the analysis stage. Therefore, it is often recommended to start testing activities from the earlier development stages and to detect faults as they occur. Design flaws are another expensive

source of software bugs (see Figure 2-3). A flawed design or requirement usually results in a flawed implementation, in which case even the best programming techniques will not mitigate this problem. Industrial experience has shown that modeling is highly effective in exposing requirements and design flaws [Research 2010; Stobie 2005].

Fault detection effectiveness is critical to any software testing technique. El-Far and Whittaker studied the use of models in software testing and noted that system specifications depicted in the form of model are effective for both test case generation and fault detection [El-Far and Whittaker 2002]. Recent studies in both commercial and academic environments have also confirmed the fault detection effectiveness of model-based testing [Jonas 2008; Paradkar 2005; Pretschner, Prenninger, Wagner, Kuhnel, Baumgartner, Sostawa, Zolch and Stauner 2005; Stobie 2005].

Table 2-1: Relative cost of fault detection and repair at different project phases [adapted from [Perry 2006]].

Phase when fault occurred	Phase in which fault is detected and fixed				
	Analysis	**Design**	**Coding**	**System Test**	**Operation**
Analysis	1x	3x	5-10x	25x	75-100x
Design	-	1x	10x	10x	30-100x
Coding	-	-	1x	10x	10-25x

- x is an assumed unit of cost for detecting and repairing fault at a SDLC phase

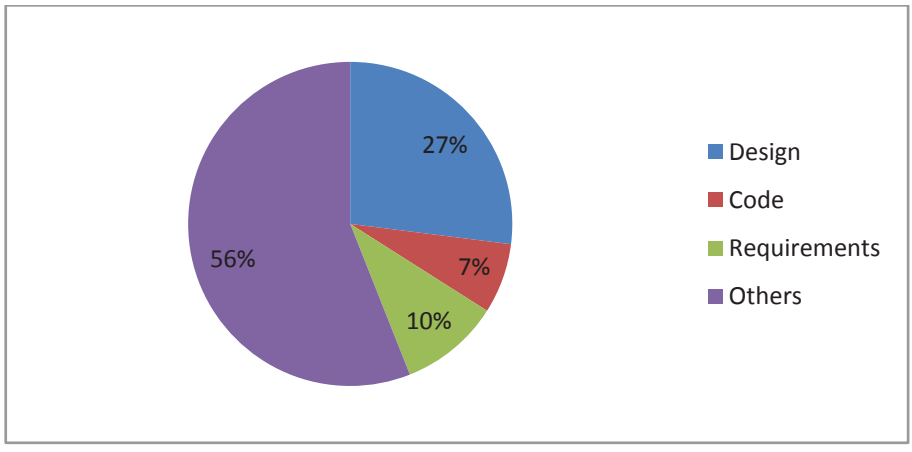


Figure 2-3: Fault proportion according to the source phase [adapted from [Rice 2010]]

One key feature of using models in software testing is that the test cases can be directly generated from the model of a system under test using an automated technique. Automated test generation not only saves cost and time but also reduces the risk of faulty and missing test

cases due to human error. Most of the modeling formalisms are amenable to automated test generation due to their theoretical foundation and mathematical or logical structure. Furthermore, as model-based testing is code independent and does not require code source to be examined, it allows the tester to use the same model to generate test cases for any language of implementation.

Ease of change management in evolving systems, and low maintenance cost are important features that make MBT a viable approach for iterative and incremental software development processes. In model driven development, multiple views of a system are developed and maintained in multiple layers (levels of abstraction) of models. Separate views emphasize different aspect of the system. This helps in identifying and rectifying the ambiguity and missing or incomplete information by focusing on specific aspects of the system. The capture of system knowledge in multiple layers enables automatic tracing between requirements, test cases and implementation [Bouquet et al. 2005]. Developing models in this way allows better handling of frequently changing requirements as well. Multiple views and layers are synchronized or updated by automated model transformation. In evolving systems, as are most of the real world systems, requirements change often and throughout the program's lifespan. Consequently, each change may require the tester to redo the testing activities in order to update the test suite. Moreover, subsequent bug-fixtures or new features in the product will require the addition of new test cases or the update of the existing test suite. The cost of manually regenerating test cases or updating the major part of a test suite can be significant. With model-based testing, the testing artifacts affected by these changes are simply handled by updating the model and the new test cases are subsequently regenerated from the updated model.

Software testing is usually carried out at three levels, namely, unit, integration and system. At the start, basic system units (i.e. methods, components and modules) are tested to determine if they work correctly and this activity is referred to as unit testing. At this level, implementation specific details are usually needed to generate test cases. For example, specific paths in the module's control structure are selected to ensure its correctness. Then, integration testing is performed to ensure that various modules when put together function properly as per expectation. Mostly, design of the software guides the identification of possible interaction scenarios of the modules that can be used to ensure the coverage of major control paths. Finally, system testing is conducted when all modules are integrated and then software is validated in accordance with the system requirements. Model-based testing can be useful at all levels of software testing. Assuming that model driven architecture (MDA) is being used in software

development, the models developed at PSM level contain low level detail that can be used to generate test cases to evaluate the particular part or module of the software. Models at the PIM level are simpler than the PSM level models. It is easy to identify potential interaction between modules with PIM level models for integration testing. The CIM level models contain business and/or system requirements that can be used for deriving systems level test cases. System level testing is relatively complex and therefore comparatively it is more beneficial by using model-based testing [Utting and Legeard 2006].

Applying MBT in practice requires well-structured modeling languages as well as scalable and practical tools for constructing and managing models and for test case generation, execution and evaluation.

2.2.3 Unified Modeling Language (UML)

A large variety of modeling languages (i.e. decision tables, finite state machines and variations, Markov chains, Statecharts and Petri Nets) are reported in literature. However, this review is restricted to UML only for various reasons. The reason for focusing on UML is that it is a mainstream software modeling language and yields a more natural representation of the world than any other approach due to its object oriented theoretical foundation. Moreover, it provides modeling notations for three of the main modeling paradigms (transition-based, history-based and operation-based as shown in Figure 2-2). UML State Machine, Sequence Diagram and Activity Diagram support modeling transition-based, history-based and operation-based behavior of the system respectively.

UML is an industry as well as ISO standard language for modeling software systems [ISO/IEC 2005; OMG 2007]. It provides a set of modeling languages for specifying, visualizing and documenting the structure and behavior of a system. The structural modeling formalisms are used to describe or visualize the static view of the system. The class, component and deployment diagrams which are classified as structural modeling formalisms focus on data or object elements and the relationship between them. The Class diagram is used to define classes, their attributes, operations, and their relationship i.e. inheritance, association and dependency with other classes. The Component diagram shows the static design of the system or part of the system using encapsulated classes, interfaces and ports. The behavioral modeling formalisms are used to specify or visualize the dynamic aspect of the system. The Activity, State Machine, Use Case and Interaction Diagrams are behavioral modeling formalisms. The Sequence diagram (subtype of Interaction) is used to express time-oriented message sequencing between objects. The Activity diagram is devised to visualize the flow-oriented aspects of the system that may

encompass simple sequential, branching, looping and concurrency. The State Machine diagram is used for specifying the behavior of the system in terms of sequence of states that a system can pass through during its lifecycle. The Use Case diagram is provided to specify the usage scenarios and Actors in a system.

2.2.3.1 UML-based testing

In order to generate test cases from a model, it must have complete and correct information about the system under test. Simple input and output information about methods of a class can be used to generate a large set of test cases which may be suitable for unit testing. However, in order to generate complete and effective test cases for functional and system testing, behavioral models are necessary. Therefore, in MBT, behavioral models are used at the start in order to determine the valid test scenarios of a system from which the relevant test cases are then selected. The following table (Table 2-2) describes the suitability of UML diagrams with the different types of testing. The focus of unit testing is to determine the correctness of a program unit (i.e. methods, objects and components). The class diagram for a component along with the control-flow or state-flow information can be used to generate a test suite with maximum code coverage for the unit under test. The Interaction and class diagrams are suitable to determine that the integrated components are working as expected. The Use Case, Activity and State Machine diagrams are particularly suitable for system level testing.

As the focus of this thesis is behavioral testing, only the behavioral types of UML diagrams are reviewed.

Table 2-2: UML and Software Testing [adapted from [Williams 1999]]

Test Type	Coverage Criteria	Fault Model	UML Diagram
Unit	Code	correctness, error handling pre or post conditions, invariants	Class, Activity and State Diagrams
Function	Functional	Functional and API behavior, Integration issues	Class, Interaction, Activity Diagram
System	Operational Scenarios	workload, contention, synchronization, recovery	Use Case, Activity, State Machine, Interaction Diagrams
Regression	Functional	Unexpected behavior from new or changed function	Same as Function
Solution	Inter system communication	Interpretability Problems	Use Case, Activity and Deployment Diagrams

2.2.3.2 Activity Diagram

The Activity Diagram is one of several behavioral diagrams in UML with particular strengths in modeling the object and control flows aspects of a system [OMG 2007]. It suits the modeling of both applications and business process systems with a variety of high and low level notations. In UML2, the AD notations are defined in multiple layers. The provision of high and low level notations (i.e. StructuredActivity) supports modeling the behavior of a system at various level of abstraction. One of the key features of the AD is the built-in modeling support for concurrency and synchronization. It can specify multiple sequences of operations executing concurrently and control their execution order with built-in fork and join constructs. Moreover, it may be used to depict various modes of parallel and distributed processing, such as synchronous or asynchronous execution of some activities.

An AD is a graph of nodes and edges that depicts the behavior of a system with a sequence of operations. Activity and action are operation nodes that are used to represent the execution of a statement in a program or the processing of a step in a workflow. Decomposition of complex operations or reuse of predefined operations can be depicted by subordinate operation nodes, such as sub or nested activities or invoked actions. The flow of control or objects through steps of operations indicates the logical paths of execution. The alternate and concurrent paths can be depicted with branch and fork nodes respectively. The sequential and synchronous paths can be depicted with merge and join nodes respectively. An operation node (i.e. Activity or Action) starts executing as soon as tokens are received on each of its inputs. An AD starts executing when the ActivityInitial node receives a token and stops when the ActivityFinal node receives a token. The diagram shown in Figure 2-4 illustrates some of the basic features of an AD. AD based testing focuses on the functional correctness of the system, and the logical paths in the model manifest the test sequences in order to detect defects like interface, decision and synchronization errors. The following paragraphs describe three AD-based test case generation approaches that have been reported in literature.

Andrews, France and Craig [2003] introduced a technique for dynamic analysis of the software design model comprising of class, activity and interaction diagrams. Their technique was based on UML 1.4 and involved testing using an executable model. An interesting aspect of the approach was that the AD was used to generate an executable model for capturing the behavior of a class and to obtain the interactions between objects from a set of ADs.

Linzhang et al. [2004] proposed a test generation technique to derive test cases directly from an AD and named it the gray-boxed method due to the fact that it synthesizes the conventional white-box (path-based) and black-box (category-partition) based test generation

techniques. According to the authors, all information for the test case generation, such as input/output parameters, conditions and expected method sequence, is extracted from the implementation model and the final test data (possible values of all the input/output parameters) is generated through black-box techniques.

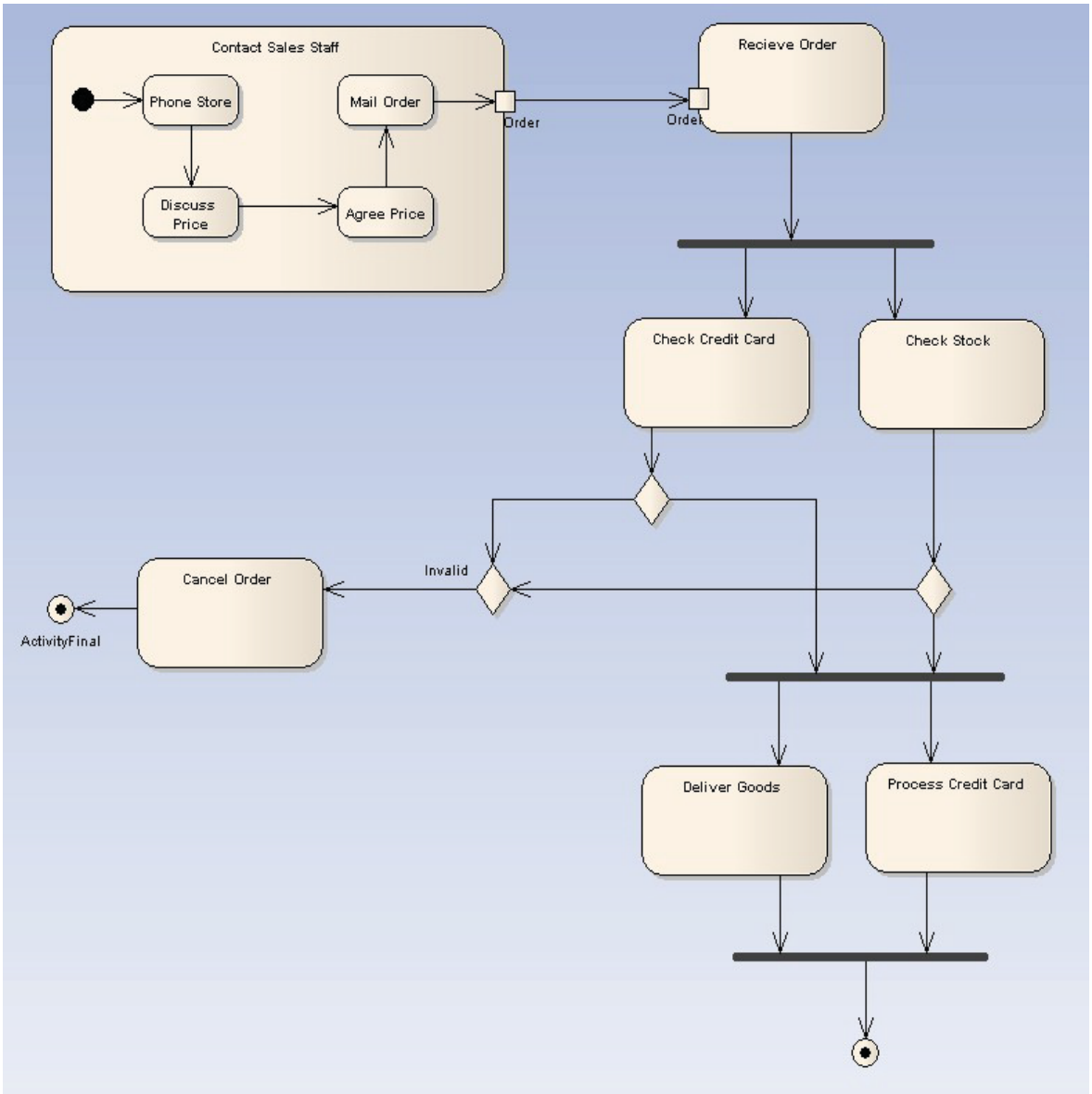


Figure 2-4: Activity diagram (taken from [Systems 2008])

Mingsong, Xiaokang and Xuandong [2006] reported a test generation technique that uses an AD as the design specification. According to the proposed technique, test cases were generated randomly and then the software under test (SUT) was executed using each of the generated test cases to obtain the corresponding execution trace. Finally the traces were

evaluated against the design specification and the specified coverage criterion. In order to obtain the execution traces, the approach involved program instrumentation where probes were inserted into the code of the software under test.

2.2.3.3 Sequence Diagram (SD)

The Sequence Diagram is one of the interaction diagrams that model the interaction between cooperating objects [OMG 2007]. An interaction is defined as a set of information messages that are exchanged or call messages to invoke some operations. A SD is a structured representation of an inter-object behavior as a series of sequential steps over time. The creation of objects is shown with lifelines running down the page. The inter-object interactions over time represented as messages are drawn as arrows from the source lifeline to the target lifeline. The SD is suitable for depicting which objects communicate with which other objects, and what messages trigger those communications. However, it is also good for showing the complex procedural logic. Sequence diagrams can be used as explanatory models for Use Case scenarios or depicting simple workflow, message passing and general interactions of elements over time to achieve a result. By creating an Actor and elements involved in the Use Case, a SD depicts the flow of steps a user and the system undertake to complete the required tasks. The diagram shown in Figure 2-5 illustrates some of the basic features of the SD.

The focus of SD based testing is on the interaction between collaborating components of a system. Since a SD captures such interaction through message exchange, message sequences are recorded to verify that the integration between the components is correct and behaves as expected. Several researchers have proposed the use of SD in software testing.

Binder [1999] defined a start-to-end message in a SD as a test sequence and presented a technique to generate test sequences from a SD by transforming it into a control flow graph. Basanieri and Bertolino [2000] proposed a Use Case and SD based technique to generate test cases for integration testing. Their approach uses a two-step strategy to generate abstract test cases. Initially test units (objects) were identified for each sub-use case and then test data (choices) were generated along the message sequences between the interacting objects using Category Partition method.

Briand and Labiche [2002] presented a methodology to derive test sequences from SD and other UML diagrams for system testing. Accordingly, the Use Case and SD are verified for correctness as a part of analysis model. Thereafter, the Use Case diagram is converted into Activity Diagram from which all possible usage scenarios are derived. Test sequences are then produced from the generated usage scenarios and message paths in the corresponding SD.

Dinh-Trong, Gosh and France [2006] proposed a sequence diagram based design analysis technique where UML design models were converted into an executable form (a program which simulates the behavior of the specified models) for testing the design models. Their approach made use of symbolic executions and a variable assignment graph that incorporates information from UML class diagrams and SD for generating test data which could then be subsequently used for testing design models.

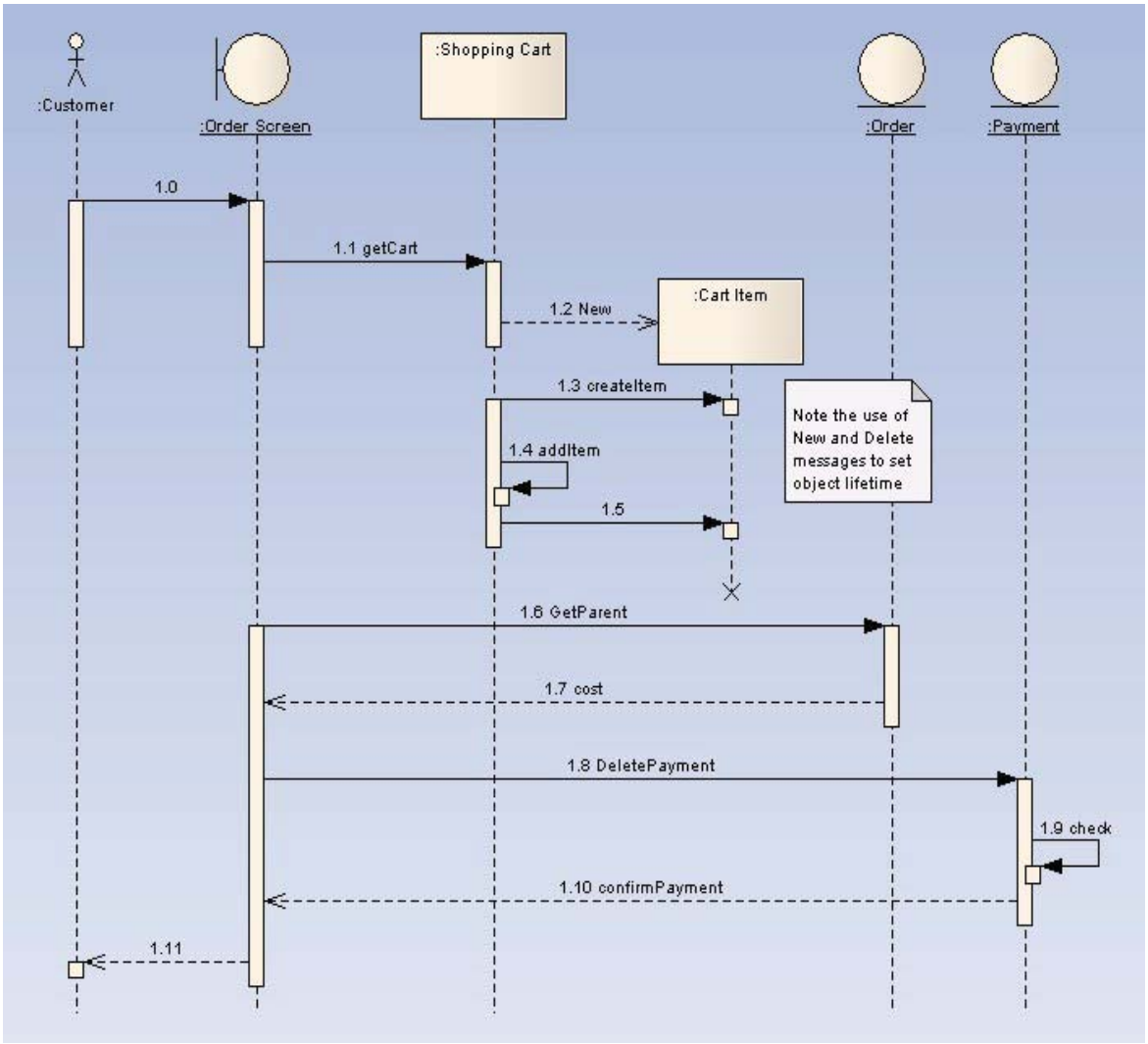


Figure 2-5: Sequence diagram (taken from [Systems 2008])

2.2.3.4 State Machine Diagram (SM)

A State Machine diagram illustrates the transition of an element between states, identifying and classifying its behavior according to transition triggers and constraining guards [OMG 2007]. Furthermore, it depicts changes in the states and behavior of a system in response to events. From UML version 2 onwards, two types of SMs have been defined. The behavioral

SM is for behavioral modeling and the protocol SM is provided to express the usage protocols. The behavioral SM diagram is adapted from the formally defined Harel statecharts [Harel and Gery 1997] for object oriented modeling [OMG 2007]. Harel statechart is itself an extension to the finite state machine (FSM) and supports hierarchy, concurrency and data variables which makes it more expressive and scalable.

Generally, a SM depicts the states of a system or component and associated events with node-arc notation. Nodes are used to represent the states of the system while arcs are used to represent the actions or operations of the system. A state represents a scenario where some invariant static or dynamic condition holds true. In a static condition, the system waits for an event to occur and in the dynamic condition the system performs a set of activities. Additionally, it supports the hierarchical and parallel representations of states with submachine and composite state elements. The composite state is an expanded state element that subsumes other state elements which are then referred to as sub-states. The submachine state element is used to refer a state to another SM which is then considered a sub-SM within that context. The transitions in one sub-SM can occur without affecting the other sub-SMs. Furthermore, there are some pseudo-states which are similar to simple states but with a pre-defined implication. Initial and final states are examples of Pseudo-states that are used to depict the start and end of a SM execution. The diagram shown in Figure 2-6 illustrates some of the basic features of a SM. In SM based testing, the sequence of events or states that a system or component may undergo during its lifetime in response to an event manifests the potential test sequences. The following paragraphs are some of the SM based testing techniques found in the literature.

Binder [1999] proposed a Flattened Regular Expression (FREE) state model-based methodology to derive test cases from state based models particularly for class testing. Accordingly, a testable SM model should be developed for flattened classes and follows FREE conventions about state, transition and unspecified event/state pairs. Furthermore, he specified a number of strategies, namely All-explicit-transitions, All-transitions, N+ and Opaque, to generate test cases from the testable SM [Binder 1999].

Offutt and Abdurazik [1999] described a technique to generate test cases from a state machine for system testing. They developed a tool integrated with Rational Rose (CASE tool) for parsing a model file and generating test sequences. They developed two test case generation algorithms and empirically evaluated them. They also proposed coverage criteria for SM based test suite and these are described in the next section. They found that the transition coverage adequate test suite was slightly better than the statement coverage adequate test suite.

Chevally and Thevenod-Fosse [2001] proposed a probabilistic technique for the generation of test cases from a SM model for functional testing. They automatically generated test cases using the Rational Rose Realtime tool and evaluated them for the transition coverage criterion.

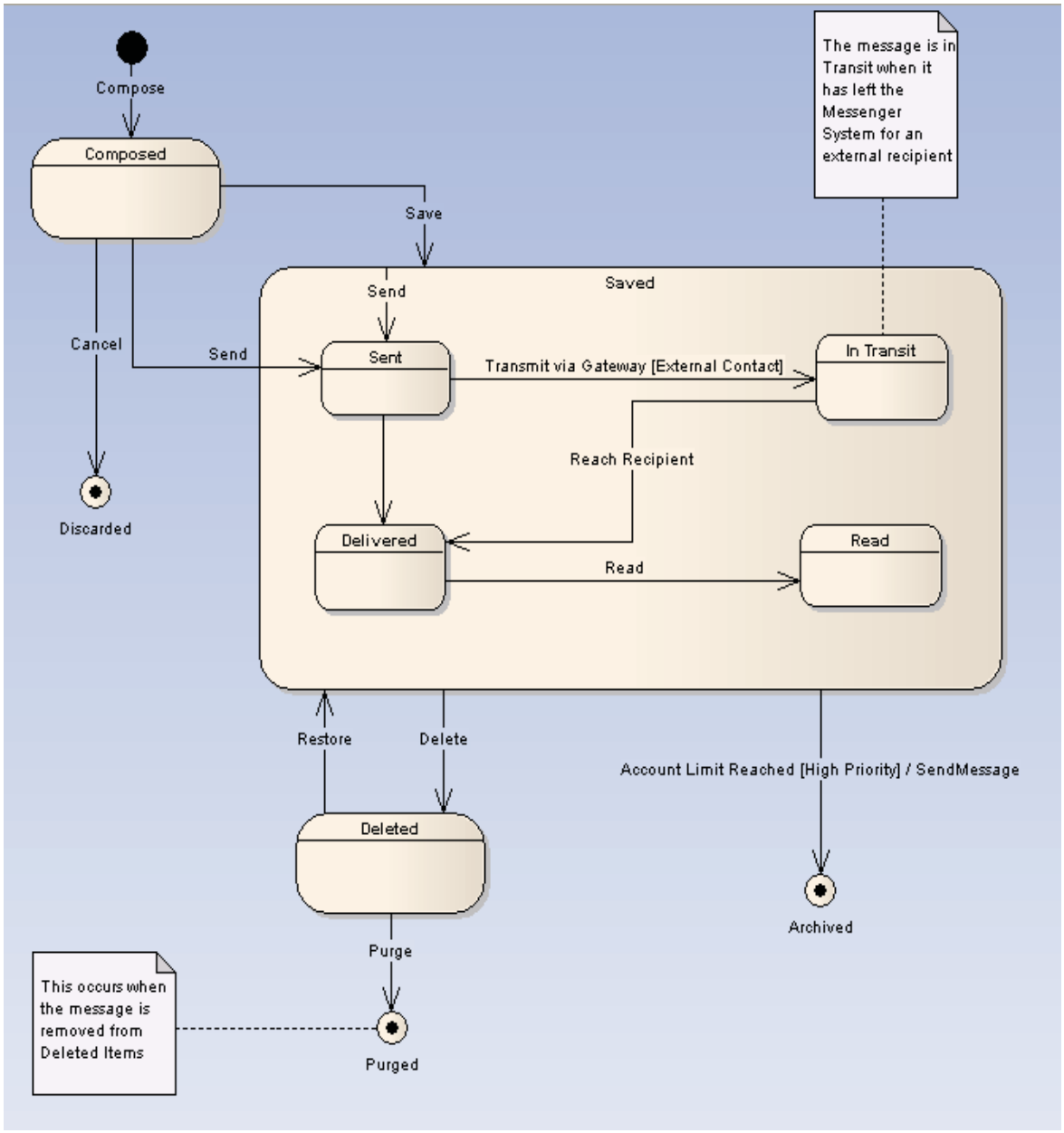


Figure 2-6: State machine diagram (taken from [Systems 2008])

2.2.4 Model-based Test Case Generation

Test case generation is the most demanding and crucial task among the testing activities. Bertolino [2003] articulated that test case generation is the most extensively researched topic in software engineering due to its complexity and importance. Furthermore, Ould [1991] asserted that it is the process of test case generation which, if automated, would give the biggest beneficial effect. The automation of test case generation is of particular significance as it can help in reducing the cost of test case generation substantially. The potential of automated test case generation is one of the key factors in the success of model-based testing [Utting and Legeard 2006].

Traditionally, the automation of the test case generation has been focused on code-based or specification-based techniques. Code-based techniques have the limitation that testing can only start after the coding phase. Specification-based techniques are typically based on formal methods that enable the automation of specification-based testing. However, due to the higher learning curve associated with formal methods, these methods could not find much presence in practice except where required for highly safety or mission critical systems.

Modern software systems tend to be large, highly interactive and often involve complex data manipulations. Simple input-output test cases are not adequate for such non-trivial systems. The required functionality cannot be tested directly and need to be done through invoking a series of operations. Typical test cases of such complex systems involve sequences of operations or usage scenarios. Thus, a testing technique must treat both the input-output relation as well as the possible sequences of interaction. As the focus of model-based testing is the behavioral correctness of the system, the generated test cases enforce the functional correctness of tasks/operations, order of execution and the dependencies among the various tasks or operations.

In software testing, the definition of a test case is contextual and relates to the corresponding test generation technique. In the context of model-based testing, a test case is a sequence of tasks or operations directly generated from a behavioral model according to a particular test objective. The terms test case and test sequence is used interchangeably in the thesis. Throughout this thesis, when the term test suite is used, it implies the collection of test cases.

The technique for generating test cases depends on the structure of a model. For some models (i.e. input/data model) combinatorial testing is suitable [Dalal, Jain, Karunanithi, Leaton, Lott, Patton and Horowitz 1999], as the combination of certain constraints or actions described

in the model guides the test case generation procedure. Whereas for some models (i.e. state machine), any simple graph traversal algorithm would be able to generate paths by traversing over the State Machine graph and then test cases are the sequences of inputs along the generated paths. In case of a Markov chain model, a random process is used to generate test cases based on the probabilities defined along the transitions (edges).

2.2.4.1 Graph Traversal Techniques

Typically, a model developed using a modeling language can be translated into a graph structure. A graph consists of a set of abstract elements called nodes and the relation between the nodes is called an edge or a link. The model depicting control or data flow of a program represents the a sequence of statements. The nodes represent the statements in the program and the edges express the flow of control or data between the statements. Nodes which have no following nodes are called leaf nodes. One of the strategies for selecting test sequences from graphs is the well-known path-based strategies of structural testing. A flow which starts from executing an entry node (initial statement) and ends at an exit node (last statement) is considered as a path [White and Cohen 1980]. Whilst there could be only one entry node, a path may have multiple exit nodes. Randomly traversing through the graph and stochastically following any available link out of a node is a simple and popular approach. More detail about path-based strategies can be found in [Beizer 1990] and [Beizer 1995].

The graph structure of a model enables the application of graph traversal algorithms to extract test sequences through the model. Graph-based test case generation can be elucidated as a path traversing through the model as a graph. A variety of graph traversal algorithms can be used to navigate through the model and generate test case. Node coverage (exercising each node at least once) or edge coverage (exercising each edge at least once) or basic path coverage (exercising all possible unique paths) are the examples of graph-based strategies. The Chinese Postman algorithm is very simple and fast graph traversal algorithm which produces complete coverage of the model with shortest test sequences [Robinson 1999]. The State-changing Chinese Postman algorithm is another technique to traverses only those links that lead to different states [Robinson 2000; Rosaria and Robinson 2000]. Another technique for selecting test cases is the Shortest Path First algorithm that starts from the initial state and then gradually evaluates all paths of length 2, 3, 4 and so on. For instances where testing according to the value of the functionality is needed, the Most Likely Paths First algorithm can be used to generate test cases for the certain areas of interest [Robinson 1999]. Moreover, new and useful test cases may also be generated by incorporating the behavioral information depicted in models into a graph traversal technique.

2.2.4.2 Formal Analysis Techniques

Formal techniques aspired to be precise and systematic, but these techniques vary in rigor and exhaustiveness. According to [Leveson 1995], formal analysis depends on the formal design process and can be very costly and time-consuming. In high risk systems the exhaustive and rigorous verification of the system is obligatory [Standard 2001]. For instance, formal specification and verification is required for safety and mission critical software systems with safety integrity of level 4 and above. Nevertheless, the enormous cost and time required for applying these techniques renders these techniques impractical for non-trivial models.

One of the formal approaches to generate a test suite from a model is based on model checking [Heimdahl et al. 2003; Khurshid et al. 2003]. Model checking is a technique that can be used to determine whether a specific property of interest is verifiable or the system exhibits a particular functional behavior. In case the property of interest is refuted, it is considered as a defect and so a supporting counterexample trace is generated. The trace is a sequence of states that are undesired according to the given property, but which the model is supporting. A counterexample represents the potential or actual fault in the system and constitutes a powerful scenario for testing. However, model checking is an exhaustive analysis technique that requires a large amount of memory to create and explore the whole state space associated with the model. The technique is prone to the state explosion problem [Merz 2000] and not suitable for non-trivial models.

2.2.4.3 Optimization Heuristic Techniques

The graphical models developed in high-level modeling languages and graphs are related at the structural level and this fact can be exploited by using graph traversal techniques to generate test sequences from such models. However, these techniques are prone to produce an incomplete test suite or a large number of invalid test sequences from graphical models as they cannot use the behavioral information during test case generation. It is important to note that both the graphical model and a simple graph exist on a different level of abstraction. In order to generate valid and adequate test sequences from such models, one solution is to transform the graphical model into a detailed graph structure. Transforming a model into a graph structure is prone to yield very large and complex graphs which could inhibit the application of typical graph traversal algorithms. Another possible approach is developing semantic aware heuristic algorithms that can traverse the model and produce effective and useful test sequences [Li and Lam 2005; Xu et al. 2005].

Heuristic techniques are typically used when exact solutions are hard to find or when there is no known way to do it that is significantly faster than trying every possible solution. A heuristic technique iteratively selects better solutions that are more likely to produce optimal outcome, rather than evaluating all possible solutions. It learns which areas of the search space to explore and which ones to disregard by evaluating a metric (e.g. Euclidean distance) for a solution in each iteration with the globally best identified solution. For many practical problems, due to the impeding cost and size of the problem a heuristic technique could be the only approach in finding a good solution in a reasonable amount of time. A metaheuristic is a general type of heuristic which can be applied to a wide range of problems, but an optimal performance is never guaranteed due to the stochastic nature of the search. Heuristic techniques particularly customized for a problem produce better results than off-the-shelf heuristic algorithms [Deb 2007]. More details about heuristic techniques are presented in Section 2.3.

2.2.5 Model-based Test Suite Evaluation

The ultimate goal of automation of test case generation is to produce test cases to confirm that adequate testing has been completed according to a given criterion. A set of such criterion are generally named as test case selection criteria or test adequacy criteria. Control-flow and data-flow coverage criteria are two typical examples of test coverage criteria. The aim of control-flow criteria is that test cases must fully exercise certain control constructs of the program under test, while the concern of data-flow coverage criteria is whether test cases will completely execute certain patterns of data manipulation in the program. Various studies have shown a strong correlation between the testing effectiveness and the coverage achieved by a test suite [Namin and Andrews 2009; Phyllis and Oleg 1998; Ye and Malaiya 2002].

Test case selection or adequacy criteria are fundamental to any testing method and act as guidelines for test selection or measuring the adequacy of the generated test suite. In order to evaluate the completeness or quality of model-based test suites three types of coverage criteria are generally used, namely model-based, implementation-based and fault-based [Pretschner, Prenninger, Wagner, Kuhnel, Baumgartner, Sostawa, Zolch and Stauner 2005]. In case of implementation-based criteria, test suites are applied to the implementation of the model, followed by the use of conventional code-based criteria for evaluation. Model-based criteria are defined by means of the coverage of model artifacts. Fault-based criteria are used to evaluate a test suite in terms of the score of actual or seeded faults detected by the test suite.

The following review is restricted to model-based and fault-based criteria. In terms of model-based criteria only those related to with the modeling techniques mentioned in the

previous section are reviewed. For more details or the explanation of coverage criteria, readers are referred to the survey on test adequacy and coverage criteria [Zhu et al. 1997].

2.2.5.1 Coverage Based Analysis

Typically, the test case generation process is guided by the predefined coverage criteria to manage the test case explosion. A number of coverage criteria have been proposed and one way to classify them is by the source of information used to specify testing requirements and in the measurement of test adequacy. Test coverage is measured as the percentage of constructs - as defined by the coverage criterion that have been executed at least once during testing. Moreover the test coverage measures (e.g. statement coverage) can also define the stopping rule to determine when this process can stop. According to Beizer [1990], satisfying a criterion does not assert that the test suite is complete in an absolute sense, however not achieving it completely must implied that something is left untested.

2.2.5.1.1 Graph-based Coverage

A test suite generated through the graph of a model must at least exercise all nodes or all edges on the graph. Table 2-3 presents three graph based coverage criteria which can e used to generate or evaluate a test suite.

Table 2-3: Coverage Criteria based on Graph Structure

Criterion	Definition
Node Coverage	Execute all nodes in the graph at least once under some test. By achieving this coverage, the test suite can be said to have achieved 100% node coverage. It is the weakest criterion in the family of graph-based criteria and is similar to statement coverage in code-based testing.
Edge Coverage	Select enough test cases to assure that every edge has been exercised at least once. By achieving this coverage, the test suite can be said to have achieved 100% edge coverage. Edge coverage is stronger than node coverage and therefore edge coverage a.k.a. branch coverage strictly includes node coverage.
Path Coverage	Execute all paths (entry/exit paths) in the graph at least once by some test. By achieving this coverage, the test suite can be said to have achieved 100% path coverage. This criterion is the strongest in the graph-based criteria family and practically impossible to achieve.

2.2.5.1.2 Model-based Coverage

In model-based testing, models are the main source of the test cases. Models define the behaviors of a system by identifying the intended usage and requirements of the system or implemented functionality of the system. Some model-based coverage criteria are briefly described in the following section. More details of the UML-based coverage criterion mentioned here or associated with other UML diagrams can be found in McQuillan and Power [2005].

Sequence Diagram Criteria

A SD as an interaction model depicts the exchange of messages between the interacting objects at runtime [OMG 2007]. It focuses on representing the behavior of a system with the timed ordering of the messages. An entry-exit path in a SD is the sequential ordering of messages that starts with an external stimulus and ends with a response to satisfy that stimulus. One requirement for the adequacy of a test suite based on SDs is that all entry-exit message paths in the diagram are covered by the test suite and characterized as All-Path coverage criterion.

Rountev, Kagan and Sawin [2005] defined the Interprocedural Restricted Control-Flow Graph (IRCFG) and used it to define a sequence diagram based coverage criteria family. An IRCFG is a graph which specifies the concrete representations of the message sequences in a sequence diagram. A node in IRCFG represents a control-flow graph referred to as restricted control flow graph (RCFG) for a particular method call with a message sequence invoked in response to that method call. Moreover, an IRCFG path is defined as a sequence of messages that starts at an entry (method call) and ends at the final message in the IRCFG and corresponds to an entry-exit message path in a sequence diagram. The all IRCFG-paths criterion means the coverage of all such paths. The definition of each of the sequence diagram based coverage criteria is given in Table 2-4.

Table 2-4: Coverage Criteria based on Sequence Diagram (adapted from [McQuillan and Power 2005])

Criterion	Definition
All Paths Coverage	A set of message paths P satisfies the all-paths coverage criterion if and only if, P contains all start-to-end message paths in a sequence diagram.
Condition/Iteration Coverage	A set of paths P satisfies the branch coverage criterion if and only if, for all edges e in the control flow graph, there is at least one path p in P

	such that p contains the edge e. Given a test set T and Sequence Diagram SD, for each loop L in SD, T must cause the loop to be either bypassed or taken for the minimum number of iterations, or to be taken at least once for the maximum number of iterations.
All IRCFG Paths	Similar to All Paths criterion but defined using IRCFG.
All RCFG Paths	A set of IRCFG paths P satisfies the all-RCFG-paths coverage criterion if and only if P contains all RCFG paths.
All RCFG Branches	A set of IRCFG paths P satisfies the all-RCFG-branches coverage criterion if and only if for all edges e in each RCFG, there is at least one path p in P such that p contains the edge e.
All Unique Branches	A set of IRCFG paths P satisfies the all-unique-branches coverage criterion if and only if for each edges e (up to equivalence) there is at least one path p in P such that p contains the edge e.

State Machine Diagram Criteria

UML SM diagram is an adaptation of Statechart for object oriented modeling and depicts the life-cycle of an object. A SM diagram specifies the states of an object during its lifetime along with the transitions between the states. One criterion for the adequacy of a test suite based on SM diagrams is that all transitions in the diagrams are covered by the test suite and is characterized as All-Transitions coverage criterion.

Table 2-5: Coverage Criteria based on State Machine Diagrams (adapted from [McQuillan and Power 2005])

Criterion	Definition
All Transitions Coverage	A test suite T satisfies the all-transitions criterion if and only if for each transition t_r in a SM diagram there exists t in T such that t causes t_r to be traversed.
Full Predicate Coverage	A test suite T satisfies the full predicate coverage criterion if and only if for each clause c in each predicate in a SM diagram there exists t_1 in T such that t_1 causes c to be evaluated to TRUE and there exists t_2 in T such that t_2 causes c to be evaluated to FALSE while all other clauses in the condition have values such that the value of the

	condition will always be the same as the clause under test.
Transition-Pair Coverage	A test suite T satisfies the transition pair coverage criterion if and only if for each pair of adjacent transitions $S_i:S_j$ and $S_j:S_k$ in a SM, there exists t in T such that t causes the pair of transitions to be traversed sequentially.
Complete Sequence Coverage	A test suite T satisfies the complete sequence criterion if and only if for each complete sequence s defined by the test engineer there exist t in T such that t causes s to be taken.
All Context-Dependence Relationship Coverage	A test suite t satisfies the all context dependence relationships criterion if and only if for each context dependence relationship r derived from a SM diagram, there exists t in T such that t tests r.

Offutt and Abdurazik [1999] proposed a UML State Machine (Statechart) based test generation technique and several associated coverage criteria (e.g. all-transitions coverage, full-predicate coverage and transition-pair coverage criteria). Transition coverage is equivalent to branch coverage in structural testing. All-transitions criterion is supposed to try every transition in the specification graph. Transition-pair criteria is relatively stronger than the All-transitions and tracks faults that may occur owing to either an invalid sequence of transitions is being allowed to execute, or a valid sequence is not allowed to execute. For the definition of each SM based coverage criterion, see Table 2-5.

Activity Diagram Criteria

Up to UML version 1.5, the SM (Statechart) based coverage criteria were also used for Activity diagram [McQuillan and Power 2005]. For instance, all-edge criterion was defined for the adequacy analysis of AD based test suite which was adapted from transition coverage criterion for Statechart [Dinh-Trong et al. 2005]. Mingsong et al. [2006] proposed three types of test adequacy criteria for AD based testing, namely activity, transition and simple path coverage. These criteria are also adapted from SM based coverage criteria. Accordingly, “all activity states” and “all transitions” in an activity diagram are required to be exercised at least once for activity and transition coverage criteria respectively.

From UML version 2 (UML2), AD has new Petri Net-like token based semantic and has become a separate diagram (instead of a derived class of SM diagram). The SM based test criteria are not suitable for UML2 AD based testing. In UML version 1.x (UML1), the action and activity node were state nodes and the transitions that connect two states could usually trigger

on the completion of a source state. In the case of multiple transitions that could trigger in response to a completion event, only one transition would trigger. In UML2, the Action and Activity nodes are executable nodes in AD and ActivityEdge is used to connect two ActivityNodes (includes both executable and control nodes) types of nodes. The AD in both UML1 and UML2 are provided for the same flow-based modeling but they have major differences. The relationship between the inputs or outputs of an activity in UML2 and UML1 are very different. In UML1, the inputs and outputs of an activity have Boolean OR relationship between them which means that if an activity state has multiple inputs (inward transitions) then it can be activated by getting stimulus on any one of its inputs (triggering of a transition). Similarly, in case of multiple outputs (outward transitions) of an activity state, only one of them will trigger. However, in UML2 the activity node have implicit join and fork relationships (Boolean AND) between its inputs and outputs respectively. It means that in UML2 an activity node cannot start execution until it receives tokens on all of its inputs and similarly it provides tokens to all of its outputs on the completion of its execution. The connectors between activity nodes that may represent alternate paths in UML1 usually do not represent alternate paths in UML2. It means that the all-transitions criterion which was defined for AD of UML1 is not effective for AD2.

The simple path coverage criterion or basic path coverage criterion as defined by [Linzhang et al. 2004] is adapted from McCabe's basic path testing strategy. It is based on the control-flow graph and requires all paths in an AD that have no loops or concurrency must be covered at least once. Given the fact that the AD is basically a flow graph model, the basic-path coverage criterion is valid for AD based testing irrespective of the version of UML. The formal definition of the criterion is presented in Table 2-6.

Table 2-6: Coverage Criteria based on Activity Diagram (adapted from [McQuillan and Power 2005])

Criterion	Definition
Basic-Path- Coverage	Let BP be the basic path set of an activity diagram, a test suite T satisfies the all basic paths coverage criterion if and only if for each p in BP there exists t in T such that t causes p to be traversed.

2.2.5.2 Fault Based Analysis (Mutation Analysis)

An alternate and complementary approach to coverage based analysis for the adequacy or quality of a test suite is mutation analysis. When a program passes all tests in a test suite, mutant programs are generated by introducing faults into the code of the program under test.

Then, the test suite is assessed in terms of how many mutants it can distinguish from the original program. Mutation analysis also referred to as mutation testing is a fault-based testing technique which was introduced by Hamlet [1977] and DeMillo et al. [1978].

Mutation analysis is traditionally used to gain confidence in the correctness of the software and to evaluate the effectiveness of test suites. It provides a comparative technique for assessing and improving multiple test suites. Previous studies [Andrews et al. 2005; Andrews et al. 2006; Do and Rothermel 2005] have confirmed the relationship between the mutation faults and real faults and have asserted that mutation analysis is an appropriate evaluation technique for test suite fault detection capability.

The conceptual basis of mutation analysis is a well-known statistical procedure of capture-recapture for estimating the size of a population. For instance, in order to estimate the number of fish in a pond, suppose 100 fish were captured, tagged and released. Later, a sample of 50 fish was taken and among them 20 was found tagged. Now the estimated population of the pond will be 250. If we could catch all the tagged fish then theoretically we have captured the entire population. Similarly, in mutation analysis some tagged faults are seeded into the program and then testing is performed to detect these faults. The testing will detect the seeded faults and some other faults as well if there is any. If all the seeded faults are detected then the test suite is considered adequate and program is deemed free from the seeded types of faults. In the case some seeded faults are still undetected then the test suite is deemed inadequate and more test cases need to be generated to find the remaining faults.

As the number of possible faults made by a programmer can be very large, only a subset of all these are targeted. It is assumed that targeting only a limited set of faults can detect other types of faults as well. Basically, mutation analysis relies on two hypotheses: (1) the program produced by a competent programmer is either correct or near correct, and (2) the coupling effect [DeMillo, Lipton and Sayward 1978]. The competent programmer hypothesis states that a program written by a competent programmer can be incorrect but it will be slightly different from the correct program. Coupling effect is the relationship between test data which is based on the fact that data that can detect the mutants with simple faults can also detect the more complex faults as well.

Mutation testing comprises of four steps: mutant generation, execution of artifact under test (AUT) using a given test suite, mutant mA execution with the given test suite and the evaluation of outcomes. Mutant artifacts mAs are generated by injecting simple faults in the AUT. A fault that can be fixed by making a single change in the source artifact is considered as a simple fault and a fault that needs multiple changes to fix is deemed as complex fault. A mutant

which is generated by inserting a single fault is called first-order mutant. More higher-order mutants are generated by inserting multiple faults but due to the coupling effect between the first-order mutants and the higher-order mutants [Offutt 1992], only first-ordered mutants are usually used in mutation analysis. Mutant operators based on a classification of faults for a given language are used to systematically inject these faults and to obtain a set of mutant artifacts. A mutant is considered killed by a test case that causes the mutant artifact to behave or output differently from the original artifact; otherwise, it is considered alive. In the case a test suite fails to kill a mutant then there could be two reasons for it; either the given test suite is not adequate to execute the faulty block of the mutant, or the original artifact (AUT) and the living mutant mA are equivalent. Equivalent mutants mean that the mutant artifacts are semantically equivalent to the original artifact despite the syntactical difference and therefore could not be killed by any test case in the test suite. In the former case, more test cases are generated until all the non-equivalent mutants are killed. While in the latter case, the equivalent mutants are determined manually as the automatic detection of equivalent mutants is an undecidable problem [Jia and Harman 2010]. So the objectives of mutation testing remain the same; to assure that the AUT is free from a particular fault set, and to generate a test suite with the ability to kill all non-equivalent mutants.

The mutation analysis of a test suite TS relative to a given program P is performed by executing P against every test case in TS. A set of mutant programs P_i' is produced by injecting a modification in P in such a way that each mutant program P_i' slightly differs from P. If in executing each mutant P_i' against TS, the output produced differs from the P's output, then the test suite has fulfilled its obligation and detected the injected fault. Once a mutant is detected and killed, it is assumed that the test suite is effective in detecting that particular type of fault. The ratio of dead mutants to the remaining live mutants indicates the quality of the test set and is called the mutation score MS. It can be defined as follows:

$$\mathrm{MS}(\mathrm{TS}, \mathrm{P}) = [\, \mathrm{M_d}/(\mathrm{M_g} - \mathrm{M_e}\,)] \times 100$$

where $\mathrm{M_d}$ is the number of mutants killed by the test suite TS, $\mathrm{M_g}$is the total number of mutants generated and $\mathrm{M_e}$ is the number of equivalent mutants which cannot be differentiated from the original program. Mutation analysis with a code-based technique is prohibitively expensive for non-trivial programs due to the large number of mutants as the number of mutants generated for a program is proportional to the product of the number of data references and the number of data objects [Yu-Seung et al. 2005].

Initially, mutation testing was introduced as a code-based technique. However, later it was adapted for specification- and model- based techniques as well. Budd & Gopal [1981]

adapted mutation testing for a specification given in predicate calculus and provided several examples of program mutations. Potential faults are represented as mutants of the specification and then test cases are generated that can distinguish between the mutated and original specification.

Stocks [1993] extended mutations analysis to Z formal specification by defining a collection of mutation operators for it. For example, a mutant operator is defined to exchange the union operator of sets with the intersection operator. Test cases are generated from these mutants for demonstrating that the implementation does not implement one of the mutant specifications and the generated test suite is adequate for detecting the given classes of faults.

Kuhn [1999] evaluated the application of mutation analysis to specification-based testing with various fault classes. Faults were represented as mutated formal specifications. Boolean operators based conditions for distinguishing a mutant specification from the original specification were calculated, and which was then used to calculate the fault coverage hierarchy and test suite effectiveness.

Fabbri et al. [1999] have conducted mutation analysis for the Finite State Machines (FSM). A set of mutation operators for FSM are defined to confirm the absence of particular faults types in the FSM model. In the work of [Fabbri et al. 1999] the authors extended the FSM-based fault model presented in [Fabbri, Maldonado, Masiero and Delamaro 1999] to Statecharts and introduced new mutation operators to address the specific to the Statechart features (e.g. parallelism, communication and hierarchy). Furthermore, Fabbri et al. [1996] have explored the application of mutation analysis in other formal specification languages such as Petri Nets and SDL [Sugeta et al. 2004]. In the work of [Souza et al. 1999], they used mutation analysis for Estelle specification. Although, the application of mutation analysis for various specification and design languages has been investigated, no work has been reported for UML in the available literature. Considering the wide use of UML, it is deemed that mutation analysis can be valuable for evaluating the UML-based test suites.

2.2.6 Issues with Model-based Testing

Earlier in section 1.2, three key properties (i.e. representation, simplification and pragmatic) of a good model are presented. Although these properties are fundamental to models in general, they are not adequate for test generation and evaluation purposes. Binder [1999] argues that a test-ready model should have enough information to derive test cases directly. Fault detection using test cases derived from imprecise and ambiguous models could be very difficult. A recent study confirms that the adequacy and level of details of a model,

influence the effectiveness of the generated test cases [Pretschner, Prenninger, Wagner, Kuhnel, Baumgartner, Sostawa, Zolch and Stauner 2005]. Binder [1999] notated that a test-ready or testable model:

- provides complete, consistent and accurate requirements of the system that need to be tested and,
- abstracts details to minimize the testing cost.

Developing a model at the right level of abstraction for effective testing is one of the main challenges for model-based testing. As the test cases are directly derived from models of the system under test, the effectiveness of the test cases depends on the information available in the model. One approach to address this issue is to keep separate models for different stages of testing. Models at a high level of abstraction (i.e. CIM level model in MDA) might not be useful for unit testing but are quite appropriate for system or user acceptance testing. On the other hand, using the implementation level model will make the cost of system testing prohibitive. It is important to note that developing or maintaining a separate model for different testing stages could also be problematic and may lead to consistency and management issues.

Automated test generation in model-based testing can quickly generate a large number of test cases. However, the increase in test cases does not improve the quality of the test suite necessarily and may compromises its efficiency. Utting and Legeard [2006, page 40] stressed that the quality of the generated test suite is equally as important as the quantity. Usually, the effectiveness of a test suite is measured in terms of satisfying test requirements (i.e. faults, coverage and mutation score) and the efficiency is measured by the cost to achieve the test requirements. Several studies have been reported on the effectiveness of model-based testing [Dalal et al. 1999; Farchi, Hartman and Pinter 2002; Pretschner, Prenninger, Wagner, Kuhnel, Baumgartner, Sostawa, Zolch and Stauner 2005]. However, there is not much work reported on the efficiency of model-based testing and this thesis focuses on improving the efficiency of model-based test suites using metaheuristic techniques.

2.3 Optimization Techniques

Optimization problems arise in almost every field (e.g. bioinformatics, manufacturing, telecommunication, economics and transportation). Consequently, many different optimization techniques have been developed. Heuristic reasoning is provisional and is a non-stringent type of reasoning with an objective to discover the solution of the present problem. Heuristic techniques are optimization techniques which are used to find minimal (or maximal) values of a

particular cost (or objective) function. They are adaptive and powerful, with the ability to obtain good results for many extremely difficult problems.

2.3.1 Metaheuristic Techniques

Metaheuristic is a class of search techniques which aspire to find an optimal solution using a given cost (or objective) function. Metaheuristic techniques are considered suitable for infinite state spaces and non-linear constraints. The strength of these techniques stems from their flexibility of the search which is simply directed by a cost function and very little other problem specific knowledge is required [Blum and Roli 2003; Corne, Glover and Dorigo 1999].

Formally, a metaheuristic is a framework that guides underlying heuristic methods to iteratively find an optimal solution for a computationally hard problem. This is achieved by dynamically balancing the convergence and divergence of underlying search strategies; as convergence leads to a quick identification of a quality solution but is prone to be trapped in a sub-optimal (local optimal) solution and divergence wastes too much time on already evaluated regions of the search space. In the literature, various heuristic techniques have been reported. Greedy, local and global searches are the three general classes of heuristic strategies [Talbi 2009].

2.3.2 Greedy Search

Greedy search is a heuristic technique that constructs solution by always selecting the locally fittest (first/ best) solution based on an objective function. The greedy selection gradually prunes the search space and finally returns an optimal solution. Although, this strategy makes it a very fast and cheap (time and space wise) heuristic, it is nevertheless prone to converging to a suboptimal solution for the same reason as the choice of one solution at one step can constrain the search space at the next step. Greedy search is a constructive type of heuristic [Silver et al. 1980] and generally performs well if each locally selected optimum is part of the global optimum.

The pseudo code for greedy algorithm is presented in Figure 2-7.

```
item = 0;
sort(item-List);
  for each x in Item-List do
    if (Evaluate(item) ≥ Evaluate(x)) then
        item = x;
end
```

Figure 2-7: Greedy Algorithm Pseudo Code

2.3.3 Local Search

The local search is a simple heuristic technique that starts from an arbitrary solution and iteratively improves upon it by moving through the search space by selecting a fitter solution and replacing the existing solution with the new one. The moves are based on local information that continues until a termination condition is met. An important difference between local and greedy search is that local search is based on solution improvement rather than solution construction which makes it stagnant and revisiting the same region continuously. Hill Climbing, Iterated local search and Guided local search are some of the popular techniques in this category. In this study Hill Climbing is used as an example of the local search.

There are three forms of Hill Climbing that differ in the criterion by which the next fitter neighbor (solution) is selected. These criteria are described as follows: Given that a bit-string representation is being used, the ***Steepest Ascent Hill Climbing*** (HCSA) systematically flips all bits in the string and chooses the string that gives the highest increase in fitness. Using ***Next Ascent Hill Climbing*** (HCNA), a single bit from left to right is flipped and evaluated progressively. The process terminates when a neighbor is found that increases the fitness and this is then chosen as the current item. This process continues until the fitness value stops improving for a given number of iterations. With ***Random Ascent Hill Climbing*** (HCRA), bits are randomly flipped until a neighbor is found that gives an increase in fitness and replaces the current item.

The pseudo code for hill climbing is presented in Figure 2-8.

```
currentItem = startItem;
loop do
  neighbours = getNeighbours(currentItem);
  for each item in neighbours
    if (Evaluate(item) > nextEval) then
        nextItem = item;
        nextEval = Evaluate(item);
  if nextEval <= Evaluate(currentItem) then
    return currentItem;
  currentItem = nextItem;
end
```

Figure 2-8: Hill Climbing Pseudo Code

2.3.4 Evolutionary Computation

The objective of global searching techniques is to seek the globally fittest solution. The search is guided by an objective function and a solution is only selected if it improves the existing globally fittest solution. Some of the well known global searching techniques are, Ant

Colony Optimization (ACO), Particle Swarm Optimization (PSO) and Evolutionary Computation (EC). In this study EC is used as an example of the global search. In the following, a review of the key features and methods of EC is presented.

EC is a population-based metaheuristic inspired from the natural process of evolution. There are a variety of EC techniques that have been proposed and studied which are in general referred to as evolutionary computation as they share common concept of evolution through individual structures, selection and reproduction processes. These techniques are characterized as Genetic Algorithms (GA) by Holland [1973], Evolutionary Programming (EP) by Fogel et al. [1966] and Evolutionary Strategies (ES) by Rechenburg [1973]. Evolution by natural selection has proven to be an effective search process and has been successfully applied to various research and application fields such as combinatorial optimization, neural nets evolution, planning and scheduling, industrial design, management and economics, machine learning and pattern recognition, to name a few [Back et al. 1997].

The evolutionary process that makes EC a very effective and efficient technique for large and complex problems comes under the influence of two fundamental sub-processes which are selection and reproduction. The former process involves determining individual members of a population for selection, survival and reproduction. The latter process performs the recombination of genes of parents to produce new offspring. One of the advantages of EC is that it thoroughly evaluates promising areas of the solution space. It is therefore inherently effective when searching or optimizing input spaces which are not smooth or continuous. Another advantage of EC is the fact that despite its stochastic nature, EC is not a simple random search. It is adaptive and the search is simply guided by a fitness function. It learns from experience and takes advantages of knowledge held in a parent population in reproducing a better generation with improved performance. Thereby, a population undergoes simulated evolution at each generation. Relatively good solutions are reproduce whilst; relatively bad ones die out and are replaced by fitter offspring.

EC techniques do not exercise any operation directly in the problem space but use an encoded space (either binary or real). Initially, a string of codes that represents the population *P* of candidate solutions is randomly generated. In the case of GA, each individual which is analogous to a chromosome in a population can have a value of $\{1,0\}$ while in ES each individual can have a real value. The fitness function *F(P)* is used to calculate the quality of each individual in a generation. A set of better individuals from the population is selected for breeding. New offspring are produced with crossover and mutation. The offspring replace the parents in the next generation if they are fitter than the parents otherwise they are just discarded. The size of

the population is very important for performance. A small population is quick at the fitness calculation and evaluation but takes longer to converge whereas a large population converges very fast but is slow on the fitness calculation. The impact of reproduction operations is significant on the quality of the solution. The crossover process exploits the available fitness information and the mutation process leads to the exploration of the search space. The pseudo code for an EC algorithm is given in Figure 2-9.

Procedure EC()

Initialization:

$c = o$ //generation counter

$G_C = \{1,2, \dots, n\}$

Solution:

Repeat

Calculate Fitness:
$F(G_C)$

Reproduce Offspring O:
$P = Select(G_C)$ // select parents
$O = crossover(P)$

Perform Mutation:
$M = Select(G_C)$ // select an individual to mutate
$G'_C = Mutate(M, G_C)$

Select New Generation:
$G_{C+1} = NextGeneration(G'_C, O)$

Until termination condition met

End Procedure EC()

Figure 2-9: Pseudo code of an evolutionary computation algorithm

EC includes several evolution based algorithms such as classifier systems, genetic programming, coevolution, memetic algorithms, GA, EP and ES. Although similar at a higher level, these algorithms differ in many aspects, such as problem representation, selection mechanism, genetic operators and performance measure. Details of each of these techniques are out of the scope of this study. For the sake of brevity only the two main evolutionary techniques (GA and ES) are reviewed in the following sections. Other evolutionary techniques i.e. Genetic Programming (GP) and EP are similar in theme to the two mentioned above but differ in the details of their implementation and the nature of the particular problem to which they have been applied. The similarities and differences are summarized in Table 2-7.

Table 2-7: Main features of Evolutionary Algorithms [Back and Schwefel 1993]

	ES	EP	GA
Representation	Real valued	Real valued	Binary valued
Self-adaptation	Standard deviations and covariances	Variances (in metaEP)	None
Fitness	Objective function value	Scaled objective function value	Scaled objective function value
Mutation	Main Operator	Only Operator	Background Operator
Recombination	Different variants important for self-adaptation	None	Main operator
Selection	Deterministic, extinctive	Probabilistic, extinctive	Probabilistic, preservative

2.3.4.1 Genetic Algorithm

The Genetic Algorithm (GA) is a stochastic search algorithm that allows the evolution of a solution to the problem. The underlying concept is that a population evolves through generations according to a set of genetic rules that lead to the emergence of ever fitter individuals. For GA, the crossover mechanism drives the solutions or in other words offspring are breed using recombination of existing individuals at each iteration step, known as a generation. The GA develops a solution constructively and maintains a constant population size.

The canonical Genetic Algorithm has the following distinct features:

Binary Encoding: For a given problem, each individual represents a feasible solution in the problem space through a suitable mapping mechanism. Holland and his associates suggested the use of strings of bits (binary digits) for the problem space mapping to individuals [Holland 1973]. The basic encoding scheme for GA was a binary string of fixed length. Suppose the search space consists of n states. The binary string (chromosome) will consist of n bits to encode the states. The length of the chromosome will remain the same in each generation.

Proportional Selection: The chances of selection for an individual depend on its fitness relative to the fitness of other individuals in the population. According to Goldberg (1989), the selection process consists of the fitness and selection functions [Goldberg 1989]. Fitness functions involve the decoding of individuals to real values, calculating the fitness

according to the objective function and then rating each individual according to its fitness relative to the other individuals. The fitness function defines the selection of individuals with a probability proportional to their relative performance in the population. The selection function uses the rating produced from the fitness function in order to select parents for breeding. Initially a number of individuals are generated at random. Each individual is evaluated and assigned a fitness value. Then each new generation of individuals is bred from the parents selected from the existing population using the fitness function.

Reproduction: The production function for generating new offspring (new regions in search space) is based on two operators which are the crossover and mutation. For GA, the crossover mechanism drives the reproduction process. The offspring are produced using a recombination of existing individuals at each generation. The mutation operator randomly inverts some bits in the binary strings with a given probability called the mutation rate. The mutation rate defines the number of bits that will be flipped in each iteration. Through crossover the search converges toward the promising regions of the search space. The mutation operation, acts as a background operator and is used only to prevent the premature convergence of the search process to a local optima by randomly sampling new points in the search space.

2.3.4.2 Evolutionary Strategies

Evolutionary Strategies (ES) also evolve solution similar to GAs but have different problem representations and breeding schemes. In the canonical ES, the floating point numbers are suggested for representing problems having continuously changing parameters and mutation is the key operator for reproduction rather than crossover.

The key features of Evolutionary Strategies are as following:

Encoding Scheme: The representation of each individual in a chromosome comprises of two parameters: genotypic and phenotypic. The genotypic parameter is a problem related floating point value and the phenotypic parameter includes evolution related strategic information for each individual that can affect the evolution process such as the mutation step size.

Selection: Two selection strategies $(\mu + \lambda) - ES$ and $(\mu, \lambda) - ES$ are defined and they imply that λ offspring will be produce from μ parents in one generation. In the case of $(\mu, \lambda) - ES$, only offspring will be considered and $\mu - best$ individuals will be selected for next generation. On the other hand, in $(\mu + \lambda) - ES$, $\mu - best$ individuals will be selected from both μ parents and λ offspring for next generation.

Reproduction: Crossover is used to reproduce offspring as in GA; however mutation is more influential than crossover in ES. In contrast to the GA, a separate mutation mechanism is used for each individual. Mutating a certain individual means perturbing the genotypic value with a random number based on a dynamic mutation step size. The mutated individual is only accepted if the mutated individual is better than the original individual.

2.3.5 Applications of Evolutionary Computation

EC algorithms are adaptive search techniques inspired by the natural process of selection and have been applied to various optimization problems. The main goal of EC techniques is to find an approximately good solution for problems that are computationally hard to solve exactly. The initial application of a GA was the classical optimization problem whereas the ES was initially applied to engineering problems. However, later a hybrid of both of these two techniques or with another optimization method (i.e. local search or hill climbing) was used in most of the applications [Whitley 1993].

Evolutionary algorithms have been applied to several classical combinatorial problems and have proven to be a robust optimization technique for obtaining consistent results. For instance, Grefenstette et al. [1985] and Oliver et al. [1987] adapted the GA for the well known combinatorial optimization Travelling Salesman Problem (TSP). The TSP problem is defined as finding the shortest distance (normally Euclidean distance) between n number of cities. Mühlenbein [1991], Fujiki and Dickinson [1987] and Wilson [1987] applied GA to the Prisoner's Dilemma problem where the two prisoners has a choice of co-operating with each other or defecting to minimize his sentence. A GA with binary encoding intuitively suits the classical 0/1 knapsack problem and many researchers have developed variants of GA to solve various versions of the knapsack problem. Khuri et al. [1990] used GA to resolve the 0/1 knapsack problem. In [Sami et al. 1994], Khuri and Batarekh adapted GA for the multiple knapsack problem. Furthermore, Chu and Beasley [1998] have studied the application of GA for the multidimensional knapsack problem. Quadratic Assignment Problem (QAP) is a more general version of TSP and is considered as one of the hardest problems. The QAP is a problem associated with the assignment of a set of resources to a set of locations while minimizing the assignment cost. Tate and Smith [1992] used GA to resolve the QAP which consistently performed equal to or better than the known techniques. Vehicle Routing Problem (VRP) is an extension of TSP and requires the optimal route for a fleet of vehicles stationed at a depot to serve a given set of customers while adhering to given conditions, like capacity, time window,

backhauling and maximum tour length. Ang Juay et al. [1999], Prins [2004] and Baker and Aychew [2003] developed various hybrid versions of GA for the VRP with time-window.

EC has seen widespread application to various practical optimization problems as an adaptive search method. Goldberg [1989] reported various engineering projects that had applied a GA to solve optimization problems, e.g. optimizing the gas pipeline control, optimizing the design of ten member plane truss with an objective to minimize the weight of the structure under minimum and maximum stress constraints [Goldberg and Samtani 1986], the design of a concrete shell of an arch dam (large scale hydropower scheme) [Parmee and Denham 1994]; design of microwave absorbers (low-profile radar absorbing materials) which resulted in a reduction of the radar signature of military hardware [Tennant and Chambers 1994] and the generation of test sequences for VLSI circuits [O'Dare and Arslan 1994].

The potential of evolutionary computing in the realm of software engineering has been investigated for various activities related to software development including requirement engineering [Bagnall et al. 2001; Zhang, Harman and Mansouri 2007], project planning and cost estimation [Aguilar-Ruiz et al. 2001; Giuliano et al. 2004], software designing [Gerardo et al. 2005; Yves, Peter and Yolande 2008], test planning, test case generation [Jones et al. 1995; Michael, McGraw, Schatz and Walton 1997; Roper 1996; Watkins 1995] and test suite optimization [Li et al. 2007; Yoo and Harman 2007], compiler optimization [Keith et al. 1999], quality assessment [Taghi et al. 2004] and software understanding [Reynolds, Zannoni and Posner 1994]. For a detailed review of the application of evolutionary and others search based techniques in software engineering, readers are referred to a recent review report [Harman et al. 2009].

2.3.6 Basics of Evolutionary Techniques

In the following sections the basic building blocks of an evolutionary technique (ET) is explained. In addition, the basic structure, a variety of operators and control parameters are also explained. Various key concepts including the population, chromosome, fitness, selection, crossover, mutation and replacement procedure will be covered briefly.

2.3.6.1 Population and generation

Central to every evolutionary technique is the concept of population which is the collection of information about a set of individuals. The individuals in a population represent solutions. The size of the population affects the performance of an ET. De Jong [1988] recommended a population of 50-100 members for optimal results. The advantage of using a population with many members is that many points in a space are searched in one generation. In

order to start the optimization process, the first population needs to be available. The initial population can be randomly seeded with a set of parameter values for each individual in the population, for instance, for binary encoding the parameter set will be {0, 1}. Alternatively, parameter values from previous experiments can be used to provide a portion of the initial population. With evolution, the individual in the population changes from one generation to next.

2.3.6.2 Representation

The chromosomes in an ET represent the individuals and provide the space of candidate solutions. De Jong [De Jong 1993] found out that the representation of the chromosome can itself affect the performance of an evolutionary algorithm. There are different possible methods of chromosome encoding that can be used in an ET, e.g. using binary, gray, integer or real value types, finite-state machine and tree encodings.

Binary encoding is the most common representation, invented by Holland [1992]. The potential solution values are encoded as bit strings composed of binary characters {0, 1}. Binary representation is not only simple and convenient in terms of problem encoding, but also for implementation. It is a genotypic representation which makes the evolutionary algorithm problem independent and traditional mutation and crossover operators can be used. In general, all bit strings within a population have the same format and length. A bit string format is described as contiguously placed binary values.

2.3.6.3 Fitness function

Evolutionary techniques require a fitness function which allocates a score to each chromosome in the current population. The fitness value of an individual is calculated based on its performance relative to the optimal (ideal) solution. Generally, the fitness is defined by: $f_i/\bar{f}$ where f_i is the fitness of individual i and $\bar{f}$ is the average fitness of all the individuals in the population. It is used to compare the individuals and to differentiate their performance in the population. An individual solution which is close to an optimal solution gets a higher fitness value than the one which is far away. An ET does not need any problem specific knowledge. The fitness value is the only feedback from the problem which guides the ET search process and exploits the area of higher fitness in the search space. The main issue in the application of an ET is often the attempt to find a suitable fitness function that expresses the problem as well as possible.

2.3.6.4 Selection

The selection operator chooses individuals from a generation to become parents for breeding the next generation. According to Darwin's evolution theory (survival of the fittest) the best individuals should survive and create new offspring. In EC, natural selection is considered as an adaptation operator which leads the search to promising regions of the search space. Selection intensity and genetic diversity are the two competing factors in a selection process that need to be considered. Selection intensity means to select only the best individuals of the current generation for mating/ reproduction which is required to drive the underlying heuristic. Diversity of solutions in the population is also required to ensure that the search does not converge prematurely and that the solution space is adequately searched. A very high selection intensity can lower the genetic diversity and risks the heuristic converging prematurely. However, low selection intensity could cause the heuristic to wander around and not converge to an optimal solution in a reasonable amount of time. Consequently, there are different techniques (e.g. elitist selection, roulette-wheel, rank selection and tournament selection) with varying emphasis and control over selection intensity and genetic diversity that can be used for selection.

Elitist Selection

The Elitist selection method is used to increase the probability of choosing fitter individuals to reproduce more often than individuals with lower fitness values. The selection process takes into account the fitness of an individual. Individuals that have higher values (fitter) are more likely to be selected for reproduction, whereas, those with low values are merely discarded. Thus, this method introduces high selection intensity which rapidly converge the heuristic to promising regions of search space.

Random Selection

Selecting only members with a high fitness can lead to inbreeding which can cause strong convergence towards a local optimum and a loss of diversity. It can be difficult to abandon the local optimum in order to find the global optimum. A pseudo random number generator is used with a uniform distribution to select the members of a generation to become parents for the mating process. The random selection works very fast, is easy to implement and guarantees genetic diversity and healthy mating.

Roulette Wheel Selection

The Roulette-wheel is a selection of parents proportional to their fitness. Conceptually, it is like a circular roulette wheel where slices for each individual have the area according to the

individual's fitness [Mitchell 1998]. Suppose the wheel is spun N times, where N is the number of the individuals in the population. On each spin, the individual under the wheel's marker is selected to be in the pool of parents for the next generation [Mitchell 1998]. The Roulette-wheel selection maintains a high selection intensity with some genetic diversity by allowing some weaker solutions to survive the selection process.

Rank Selection

The Rank selection is a ranking-based technique to select individuals according to their fitness ranking. In cases where the fitness of individuals differs very much e.g. if the best individual fitness is 90% of all the individuals in the roulette wheel then the other individuals will have very little chance of selection. In the Rank selection, first the population is ranked and then every individual receives a fitness score according to its ranking. The worst will have fitness 1, second worst 2 and so on. The best will have fitness N (number of individual in population).

Tournament Selection

The Rank selection technique first seeks to sort the population which is usually considered unattractive for large problems. In general, the tournament selection process involves randomly choosing a group of individuals from the current population, comparing their fitness, and then selecting the fittest from the group for mating. Various tournament selection parameter control schemes have been defined. Examples include fixed tournament size, probabilistic tournament selection, Boltzmann selection with annealing, self-adaptive tournament size and fuzzy tournament selection.

The following figures show the effect of various selection techniques on same individuals in a population. Suppose A={(a,5),(b,6),(c,9),(d,2),(e,1),(f,4)} is a set of individuals with their fitness values. In case of elitist selection (Figure 2-10), the selection intensity is highest and two individuals 'b' and 'c' with highest fitness values will dominate the selection as parents. In case of random selection, any two individuals will be selected for mating with every individual have equal chance of selection as shown in Figure 2-11. The Roulette-wheel selection and the Rank selection without truncation are similar approaches. However, in Rank selection the selection intensity is less than that in the Roulette-wheel selection so the less fit individuals have slightly more chances of selection as shown in Figure 2-12 and Figure 2-13. The Rank selection with truncation (Figure 2-14) has more selection intensity than the Rank selection without truncation. Tournament selection is often used in practice as it offers greater control over the selection intensity than other technique [Tobias and Lothar 1995]. It allows control of the selection intensity by specifying the tournament size and the number of members chosen to compete in

each tournament. For instance, see the selection intensity of tournament selection with different tournament sizes (k=2, k=5 and k=6) as illustrated in Figures Figure 2-15-Figure 2-17. The binary tournament (k=2) implies the weakest selection intensity whereas N-size tournament (k=6) implies the strongest intensity

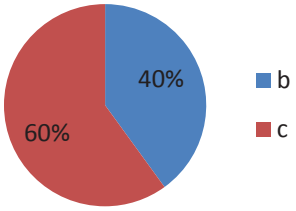


Figure 2-10: Elitist selection

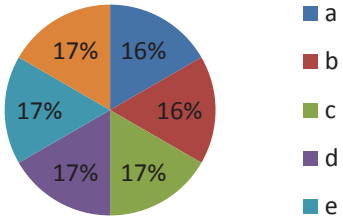


Figure 2-11: Random selection

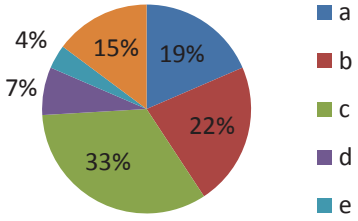


Figure 2-12: Roulette-wheel selection

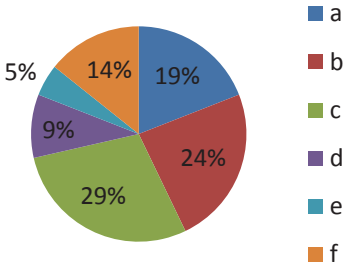


Figure 2-13: Rank selection

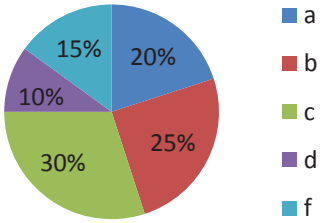


Figure 2-14: Rank selection (truncation=5)

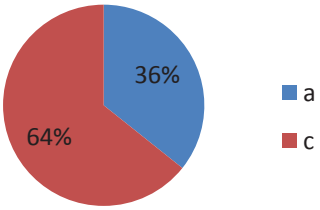


Figure 2-15: Tournament selection (k=2)

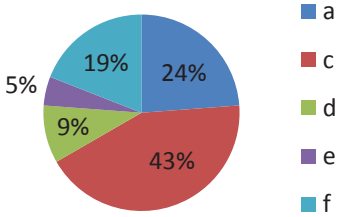


Figure 2-16: Tournament selection (k=5)

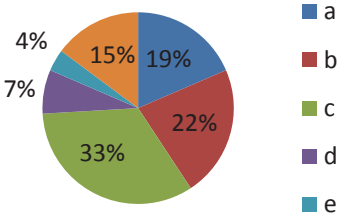


Figure 2-17: Tournament selection (k=6)

2.3.6.5 Reproduction Operators

The crossover and mutation operators have the pivotal roles in GA. They are the operators which create new individuals with the idea that the new individuals will improve the solution and move the search closer to a global optimum. The crossover process exploits the available fitness information and the mutation process leads to exploration of the search space.

Crossover Operator

Crossover is the process of combining the bits of one parent with those of another. During this process, the two parents exchange sub-string information (genetic material) at a random position in the chromosome to produce two new strings (offspring). The objective here is to create new individuals by combining material from pairs of (fitter) members from the parent population. Crossover is performed according to a crossover probability. Various crossover methods have been proposed and examples include single-point, double-point and uniform crossover. The simplest one is the single-point crossover that selects a locus randomly and exchanges the sub-string before and after that locus between two parents to create new offspring.

Single crossover

A crossover point is randomly chosen for two parents. This point occurs between two bits and divides each individual into left and right sections. Crossover then swaps the left (or right) section of two individuals. For example, consider the following parents and a crossover point at position 3:

Parent 1: 1 0 0 | 0 1 1 1 1

Parent 2: 1 1 1 | 1 0 0 0 1

Offspring 1: 1 0 0 1 0 0 0 1

Offspring 2: 1 1 1 0 1 1 1 1

In this example, Offspring 1 inherits bits in position 1, 2, and 3 from the left side of the crossover point from Parent 1 and the rest from the right side of the crossover point from Parent 2. Similarly, Offspring 2 inherits bits in position 1, 2, and 3 from the left side of Parent 2 and the rest from the right side of Parent 1.

Double crossover

Double Crossover operates by selecting two random bits within the parent strings with subsequent swapping of bits between these two crossover points. For example, consider the following parents and two crossover points at position 2 and 5:

Parent 1: 1 0 | 0 0 1 | 1 1 0

Parent 2: 1 1 | 1 1 0 | 0 0 1

Offspring 1: 1 0 1 1 0 1 1 0

Offspring 2: 1 1 0 0 1 0 0 1

In this example, Offspring 1 inherits bits in position 1, 2, 6, 7 and 8 from the Parent 1 and the rest of the bits (3, 4 and 5) from the Parent 2. Similarly, Offspring 2 inherits bits in position 1, 2, 6, 7 and 8 from the Parent 2 and the rest (3, 4 and 5) from the Parent 1.

Uniform crossover

Uniform crossover is the extension of 1-point crossover to n-point crossover where n is the number of crossover points. Uniform crossover means that each bit of the parents can be selected according to some probability P_C so that these two bits are exchanged to create offspring. The number of genes exchanged during uniform crossover is on average $(\frac{S}{2})$ crossings on strings of the length S for $P_C = 0.5$. For example, consider the following parents with S=8:

Parent 1: 1 0 0 0 1 1 1 0

Parent 2: 1 1 1 1 0 0 0 1

Offspring 1: 1 1 0 0 0 0 1 1

Offspring 2: 1 0 1 1 1 1 0 0

In this example, Offspring 1 inherits bits in position 1, 3, 4, and 7 from the Parent 1 and the rest of the bits (2, 5, 6, and 8) from the Parent 2. Similarly, Offspring 2 inherits bits in position 1, 3, 4 and 7 from the Parent 2 and the rest from the Parent 1.

Mutation operator

The mutation operator in evolutionary algorithm is analogous to biological mutation and aims to introduce genetic diversity in the population. It triggers randomly chosen genes to change (flip) bits, whereas crossover allows random exchange of information of two individuals. If selection leads the heuristic to converge to an optimal value then mutation allows it to avoid the local optima. Mutation can introduce new information that may not be present in the current population because it was lost during crossover or performance-base d selection.

Mutation changes the new offspring by flipping bits from 1 to 0 or from 0 to 1. In a string, any bit can get flipped with some probability, usually very small (e.g. 0.01). Single point mutation is a simple technique and in the case of constrained problems, where the gene encoding is restrictive (i.e. as in permutation problems) then mutation can be implemented with swaps, inversions and scrambles.

Normal mutation

Mutation when applied to new individuals created through crossover is called normal mutation and it flips some bits with a pre-defined probability. For example, consider the following bit-string of an individual with mutation point at position 2:

Pre-mutation Individual: 1 0 0 0 1 1 1

Post-mutation Individual: 1 1 0 0 1 1 1

The 0 at position 2 flips to 1 after mutation.

Weighted mutation

Weighted mutation is only executed when the population seems to be stuck. Weighted mutation increases the probability of mutating certain bits depending on the representation type. Usually, it flips the most significant bit and/or some of the least significant bits.

2.3.6.6 Replacement

The replacement method decides which individual of the offspring and parent population will survive into the next generation and which will be discarded or dies. To accomplish this, the procedure may copy parts of the offspring population to the parent population according to some criteria, such as the fitness of the individuals similar to the selection process. There are many different methods of replacement, i.e. random replacement, parent replacement, worst replacement, most-similar replacement (crowding) and elite replacement. Some of these are explained in the following sections.

Parent replacement

In direct replacement, parents are replaced by their offspring. As with this approach, parents are replaced by their offspring so it allows the preservation of information in less fit individuals in the population. One advantage of using this technique is that it can avoid deceptive traps in the search space.

Elitist replacement

Selects only the very best individual with regard to the fitness value of the offspring population, and overwrites the very worst individual from the parent generation. Since the fittest individual always survives it could mean that after a couple of generations the population comprises several copies of the same very dominant individual which can lead to inbreeding and loss of genetic diversity. The advantage of this technique when used in conjunction with other operators is to have more exploitation and less exploration. One drawback is that it can fall into deceptive traps i.e. suboptimal region in a multi-peaked space.

2.3.6.7 Termination conditions

The evolutionary algorithm executes until a termination condition has been reached. There are a number of termination conditions:

- Search gets stagnant such that successive iterations stop producing better results after finding a fittest solution or reaching a peak in the search plateau.
- The population is converged; De Jong defined a convergence as when a particular gene has the same value in at least 95% of all individuals in a population [De Jong 1975].
- A solution is found that satisfies the expected optimal criteria.
- The predefined number of generations is reached.
- Any combinations of the above.

2.3.6.8 Performance Metric

Evolutionary techniques have been applied to many optimization problems with variety of search spaces i.e. linear, multi-dimensional, multimodal, discontinuous and noisy. Although for simple problems, an evolutionary algorithm may perform well, it needs appropriate optimization operators and parameters in the case of complex problems. In many such cases, the choice of the optimization operators may not be obvious. Even when an appropriate class of optimization operators is available, there are several other parameters that need to be tuned, e.g., crossover and mutation rate, tournament size for selection. The problem of tuning the algorithm is dependent upon the performance evaluation of the algorithm. De Jong [1975] and Grefenstette [1986] proposed three performance evaluation metrics for evolutionary algorithms which are as follows:

Online Performance:

The online performance is the average fitness value of all individuals that have been generated and evaluated by the algorithm up to the current evaluation s. It means that if the heuristic concentrates on the areas with higher fitness values it will perform better than searching the area with lower fitness values.

$$\mathrm{onl}(T) = \frac{1}{T}\sum_{s=1}^{T} f(s) \qquad (2\text{-}1)$$

Where $f(s)$ is the fitness value on s evaluation step. A low value for online performance means that the heuristic is wasting too many evaluations on 'bad' solutions.

Offline Performance:

The offline performance at time t is the running average of the best fitness values found by an optimization technique up to s evaluation steps (generations). It means that the heuristic which produces better solutions by exploring poor regions of the search space may perform better than the heuristic which focuses on a restricted area of the search space. The offline performance can be defined as:

$$\mathrm{offl}(T) = \frac{1}{T}\sum_{s=1}^{T} f^{*}(s) \qquad (2\text{-}2)$$

Where $f^{*}(s)$ is the best fitness value found up to s evaluation steps. The offline performance measure is also used to see how quickly the heuristic converges to the optimal solution.

Best Value:

An obvious metric is the global best solution. The global best solution is the individual with the highest fitness from all generations that has been generated.

For example, if at t = 10, five individuals have been produced and evaluated yielding fitness values of 10, 20, 8, 4, and 25, the on-line performance will be (10 + 20 + 8 + 4 + 25)/10, the off-line performance will be (10 + 20+ 20 + 20 + 25)/5, and the best-value performance will be 25. The online and offline performance metrics are greatly affected by the population quality. In a population with high diversity the online performance will be low which indicates a bad

performance for the heuristic. In a population with low diversity, the offline performance may not be an effective measure.

2.3.7 Multi-Objective Evolutionary Algorithms

Evolutionary Computation (EC) is a population based meta-heuristic which means that it processes a population of solutions in every generation, thus making EC an ideally suitable technique for multi-objective optimization problems. For single objective optimization, EC algorithms (EAs) are designed to optimize only a single objective or decision variable. However, in the case of multi-objective optimization, there are two approaches. A multi-objective EA (MOEA) that produces multiple Pareto-optimal solutions without having pre-defined relationship or preference among objectives is called the Pareto-based technique while a MOEA that produces single solution according to a given preference or prior trade-off relationship among the objectives is usually referred to as non-Pareto or priori technique. An EA incorporated with preference or prior trade-off relationship among the objectives is usually referred to as non-Pareto or priori technique while the EA that produces multiple Pareto-optimal solutions without having pre-defined relationship or preference among objectives is called the Pareto-based technique. Formally, the multi-objective optimization is stated as follows:

$$\text{minimize or maximize } F(x) = \left(f_1(x), \dots, f_n(x)\right) \qquad \text{(2-3)}$$

$$\text{subject to} \quad x \in D$$

where $F(x)$ is the vector of objectives; $n \geq 2$ is the number of objective functions; $x = (x_1, x_2, \dots, x_n)$ is the vector of decision variables; and D is the feasible solution space. Although, multi-objective problems can be optimized with single objective algorithms, in these cases all objectives are optimized one by one with a single algorithm being run, while others are handled as constraints. In the end, using this approach, a trade-off solution is manually identified among the objectives. However with multi-objective algorithms, all objectives are optimized simultaneously and a trade-off solution or a set of solutions is devised automatically. Multi-objective optimization elevates the need to run the single-objective algorithm multiple times in order to generate the optimal solution set for each objective.

Generally, MOEAs are classified according to the underlying solution propagation mechanism, i.e. Pareto and non-Pareto based techniques. The Pareto based techniques (e.g. Niched Pareto Genetic Algorithm (NPGA), Non-Dominated Sorting Genetic Algorithm (NSGA) and Strength Pareto Evolutionary Algorithm (SPEA)), produce multiple distinct solutions known as

Pareto optimal (solution) sets, whereas the non-Pareto class of techniques such as VEGA and Min-Max, are designed to propagate a globally unique solution.

Pareto Optimum:

The optimality criterion for multi-objective problems is defined as follows [Coello 2000]: a point is said to be Pareto optimal, if there is no other point dominating the point with respect to a set of objective functions. A point *x* dominates a point *y*, if *x* is better than *y* in at least one objective function and no worse with respect to all other objective functions. Coello defines the Pareto Optimum as " x is Pareto optimal if there exists no feasible vector x which would decrease the same criterion without causing a simultaneous increase in at least one other criterion." [Coello 1998]

Pareto Front:

The Pareto optimum is not always a single solution. It usually comprises of a set of solutions called the Pareto optimal set or the non-dominated solution set or Pareto front [Coello 1998].

2.3.8 Non-Pareto-Based Evolutionary Algorithms

In typical multi-objective problems, all the possible solutions represent some trade-off relationship among the objectives. However, as mentioned earlier, an EA incorporated with a preference or prior trade-off relationship through fitness or cost function mechanism is called a non-Pareto or priori technique. The following paragraphs describe some of the examples of non-Pareto-based evolutionary techniques.

2.3.8.1 Aggregation-based Approach

Usually in an evolutionary algorithm, a scalar fitness function guides the search directly. One approach in dealing with multiple objectives is to combine them into a single function so that each potential solution is evaluated according to some predefined relation (i.e. preference and weight) between the objectives. The approach of combining objectives into a single (scalar) function is normally defined as aggregating functions. An example of an aggregating function is as follows:

$$min \sum_{i=1}^{k} w_i f_i(x) \qquad (2\text{-}4)$$

Where $w_i > 0$ are the weighting coefficients representing the relative importance of the k objective functions of our problem and it is assumed that:

$$\sum_{i=1}^{k} w_i = 1$$

The main problem with aggregation-based techniques is that they need to know the normalization, prioritization or weight relationship among the various objectives for a suitable measure. Combining objectives that interact or conflict with each other (such as increasing one, can reduce others, especially in a nonlinear way) makes the function very complex. Moreover, the set of solutions produced through this technique will be highly dependent on the function and the relationship defined.

2.3.8.2 Vector Evaluated Genetic Algorithm (VEGA)

Schaffer adapted the simple GA for multi-objective optimization with a vector valued fitness function and a performance-proportion selection operator that selects individuals according to each objective at each generation [Schaffer 1985]. The algorithm was named the Vector Evaluated Genetic Algorithm (VEGA) as it requires selecting parts of the population (subpopulations) according to each of the objectives separately. So for a problem with k objectives and a population of size N, k sub-populations of size N/k each would be generated. These would then be shuffled together to obtain a new population of size N, on which the usual crossover and mutations operators of GA would be applied.

The objective of the algorithm is to find solutions with moderate performance for all objectives as a set of compromised solutions. However, the algorithm does not have explicit diversity control mechanism so the solutions generated by VEGA suffer a speciation problem such that the algorithm evolves species within the population dominated on different objectives rather than producing a compromised solution.

2.3.9 Pareto-based Evolutionary Algorithms

As an EA is a population-based technique, it can produce a large Pareto-optimal solution set in a single iteration intuitively. From the population of solutions, it preserves the diverse niche of multiple non-dominated solutions by considering all non-dominated solutions equally. At each generation multiple good solutions are exploited and the search gradually converges close to the Pareto-optimal front with a good spread.

2.3.9.1 Pareto Archived Evolutionary Strategy (PAES)

Knowles and Corne [2000] proposed Pareto-based extension of evolutionary strategy and named Pareto Archived Evolutionary Strategy. As the name suggests, it uses an archive of all the non-dominated solutions generated previously in selecting parents for mating. Furthermore, it uses a single-parent, single-offspring strategy for reproduction. An important feature of PAES is the crowding procedure that divides the objective space recursively which spares the need to define the niche size. Therefore, niches are simply produced by placing the solutions into a number of divisions in the objective space.

PAES Procedure
Begin
Initialize population *pop*
Generate solution S_i randomly and add it archive
Repeat until (*terminal condition is not satisfied*) Do
Mutate S_i to produce S_{i+1} and evaluate S_{i+1}
If (S_i dominates S_{i+1}) Discard S_{i+1}
Else if (S_{i+1} dominates S_i)
Replace S_i with S_{i+1}, and add S_{i+1} to the archive
Else if (S_{i+1} is dominated by any member of archive) discard S_{i+1},
Else apply test (S_i, S_{i+1}, archive) to determine the new current solution and whether to add S_{i+1} to the archive.
End
End

2.3.9.2 Niched Pareto Genetic Algorithm (NPGA-2)

Horn et al. [1994] first proposed the Niched-Pareto Genetic Algorithm (NPGA) which is an extension of the traditional GA with a Pareto domination based tournament selection, fitness sharing and fixed niche radius. It uses the same production operators (i.e. crossover and mutation) as in a single objective GA. However, for the selection operator, a Pareto dominance strategy is used to select the winner. The selection strategy is defined as follows: two candidate solutions are selected randomly from the population. A subset of the population is also selected randomly for comparison. Each of the candidate solutions is compared against each individual in the selected subset. If one of them is dominated and the other is not dominated, then the non-dominated individual wins. If there is no clear winner (i.e. neither dominated nor non-dominated) then the result of the tournament is decided through fitness sharing. Errickson et al. [2001] subsequently proposed a revised version of the NPGA and named it NPGA-2. This algorithm uses Pareto ranking instead of Pareto dominance in tournament selection. In the

algorithm, a dynamically updated niching strategy is used and the niche count is calculated using individuals in the partially filled next generation.

NPGA-2 Procedure
Begin
    Initialize population *pop*
    Repeat until (*terminal condition is not satisfied*) Do
        Perform *Pareto-Rank based Tournament Selection*:
            Select *Parent* P_1 and P_2
        Perform *crossover*:
            Between P_1 & P_2, produce *offspring* O_1 & O_2
        Perform *mutation*:
            On O_1 & O_2
        Update *offspring population* with O_1 & O_2
    End
End

2.3.9.3 Elitist Non-Dominated Sorting Genetic Algorithm (NSGA-II)

NSGA-II is one of the most efficient multi-objective evolutionary algorithms proposed by Deb et al. [2000] which is based on the elitist non-dominated sorting approach. The approach uses the elitist diversity-preserving mechanism to allow the elite of the population to compete for survival over the next generation. The fitness assignment scheme is based on the non-domination level. Offspring are generated using a standard bimodal crossover and mutation operators. A binary tournament selection is made on a non-dominance and diversity basis. The crowding comparison procedure is used in the tournament selection and the population reduction phase in order to keep the diversity in the solution.

The following is a step-by-step detail of the algorithm. Initially, a random population P_i is created and sorted into different non-dominated classes. The fitness for each solution is calculated based on its non-domination level. Once the offspring population P_o is created, both offspring and parent populations P_i are combined from the temporary population P_t of size 2N. P_t is sorted in a non-dominated form which classifies the population into groups termed as non-dominated fronts. Thereafter, the best N members of the P_t are chosen to form the next generation in the following manner. Firstly, solutions of the best non-dominated front are selected. This is followed by the second and third fronts and so on, until the new population is full and the rest of the solutions are simply deleted. In the case where the solutions in a non-dominated front are more than the size of the new population, a niche of the solution is selected from the least crowded region of the front. However, the solutions are selected based on their crowding distances so that no extra niche parameter (such as niche radius in NPGA) is required.

2.3.9.4 Strength Pareto Evolutionary Algorithm (SPEA)

Zitzler and Thiele [1999] extended a multi-objective evolutionary algorithm by introducing a clustering technique to maintain diversity in the populations and non-dominated solutions set based archiving mechanism, similar to PAES for selecting individuals that will survive into the next generation. The strength of each of these non-dominated individuals is proportional to the number of solutions which it dominates. The fitness of each member of the population is computed according to the strengths of all archived non-dominated solutions that dominate it.

SPEA Procedure
Begin
    Initialize population *pop* and create external non-dominated set S'
    Repeat until (*terminal condition is not satisfied*) Do
        Copy non-dominated members of *pop* to S'
        Update the external non-dominated set S':
            Remove the duplicate solutions
            If (Size of S' > maximum N')
            Prune S' using clustering mechanism
        Compute fitness of each individual of *pop* and S'
        Perform *Binary Tournament Selection* on *pop* and S' for mating pool
        Perform *crossover* and mutation.
        Update *pop* with new generation.
    End
End

2.3.9.5 Multi-objective Genetic Algorithm (MOGA)

Fonseca and Fleming [1993] proposed an extension of the GA for multi-objective optimization consisting of a selection technique in which the rank of a certain individual corresponds to the number of individuals in the current population by which it is dominated. The algorithm uses a rank-based fitness assignment method that assigned the non-dominated individuals rank 1 and the dominated individuals are penalized according to the population's density of the corresponding region of the trade-off surface. Moreover, the authors proposed the use of a niche-formation method to distribute the population over the Pareto-optimal regions and suggested some guidelines for the determination of the niche sizes. Restricted mating is also recommended in order to avoid the excessive competition.

2.3.10 Multi-Objective Performance Metric

For single objective algorithms, generally the best solution is considered to be an ultimate performance measure. However, for multi-objective algorithms, as the optimization seeks the

trade-off between the objectives, there is mostly more than one optimal solution. It means that a multi-objective optimization algorithm generates a set of non-dominated solutions also known as a Pareto-optimal set. An important assumption about Pareto-based techniques is the absence of prior preference information about the objectives which means that each solution in the Pareto-front is equally as good as the others. However, there are three main factors that are used to scale the performance of an algorithm: 1) the closeness of the generated Pareto front to the Pareto-optimal front (true Pareto front), 2) the degree of solution diversity in the Pareto front, and 3) the width or the spread of the generated Pareto front. Generating wider Pareto front (maximum spread) closest to the true Pareto front is the most important objective of the Pareto-based optimization as it provides more alternatives for the decision maker.

Given the fact that multi-objective heuristic techniques are approximate techniques, the solution set produced by them are approximate Pareto fronts. Hansen and Jaszkiewicz (1998) defined a number of compatibility and outperformance relations to express the comparative relationship between two solution sets as reported in [Knowles 2002].

Given A and B are two approximate solution sets,

- The solution set A ***weakly outperforms*** the solution set B if all solutions in B are covered (equal to or dominate) by those in A and there is at least one solution in A that is not contained in B. Formally, it is defined as follows: $AO_w B \iff ND(A \cup B) = A$ and $A \neq B$.
- The solution set A ***strongly outperforms*** the solution set B if all points in B are covered by those in A and at least one point in B is dominated by a point in A. Formally, it is defined as: $AO_s B \iff ND(A \cup B) = A \; and \; B \backslash ND(A \cup B) \neq \emptyset$.
- The solution set A ***completely outperforms*** the solution set B if each solution in B is dominated by a solution in A and is formally defined as: $AO_c B \iff ND(A \cup B = A \; and \; B \cap NDA \cup B = \emptyset$.

Assessing and comparing the performance of multi-objective algorithms is difficult because of the multidimensional nature of the Pareto fronts. As the outcome is a set of non-dominated solutions rather than the single best solution, it is hard to represent the solution quality on a plot against time. A number of performance metrics for multi-objective optimization algorithms have been reviewed in [Deb 2001] and have highlighted the fact that a proper comparison of results of these algorithms is a complex issue. The following are some of the measures taken from [Knowles 2002] to evaluate and compare the performance of multi-objective optimization techniques.

Generational Distance:

Generational distance is a measure of how close the current Pareto front is from the true Pareto front (based on the assumption that the true Pareto front is already known).

$$GD = \frac{1}{n}\sum_{i} D\left(z_{ref}^{i} - z_i\right) \qquad (2\text{-}5)$$

Where n is the size of the produced Pareto front PF_{prod}, z_{ref}^{i} is the individual member of the true or reference Pareto front PF_{ref}, z_i is the individual of the current Pareto front, and $D(z_{ref} - z_i)$ is the Hamming or Euclidean phenotypic distance between z_{ref}^{i} and z_i. If $GD = 0$ then $Pf_{prod} = Pf_{ref}$.

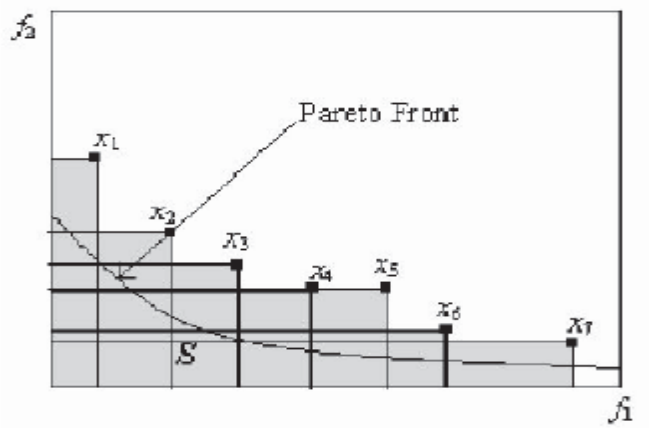


Figure 2-18: The example of the S metric in the case of two objective functions f_1 and f_2 with 7 decision vectors $(x_1, x_2, \ldots, x_7)$ for a minimization problem. [INRIA 2010]

Size of the Dominated Space:

The size of the dominated space metric (*S*-metric), which is generally recommended for assessing the quality of Pareto-based optimization techniques, is used to compute both the convergence and spread of the generated non-dominated solution sets with respect to a reference point. Figure 2-18 visually depicts the size of the dominated space. Depending on the selected reference point z^{ref}, the *S* value of two different non-dominated solution sets (Pareto front) can be different, therefore the selection of reference point is critical. The following equation taken from [Knowles 2002, page 98] is used to compute the *S* metric:

$$S = \sum_{i \in 1..|A|} \left|z_1^i - z_1^{ref}\right| . \left|z_2^i - z_2^{i-1}\right| \qquad (2\text{-}6)$$

Usually, the objectives functions have different range metrics, so the objective values are normalized before computing the *S* value. The maximum possible valid value of each objective is used as a component of the reference vector [Deb 2001]. In order to calculate the value properly when the optimal value is also the ideal value, a small positive ε-value is added to each component of the reference point [Hansen and Jaszkiewicz 1998].

Coverage

The coverage metric (*C*) is used to evaluate the performance of Pareto-based optimization techniques or their outcomes by comparing two sets of non-dominated solutions against each other. The following equation is used to compute the *C* metric:

$$C(A,B) = \frac{|\{b \in B | \exists a \in A : a \preccurlyeq b\}|}{|B|} \qquad (2\text{-}7)$$

The value $C(A,B) = 1$, means that all decisions vectors in B are dominated by A, whereas $C(A,B) = 0$ means that none of the points in B are dominated by A. As $C(A,B)$ is not necessarily equal to $1 - C(B,A)$, it is necessary to calculate $C(B,A)$ separately.

Spacing

The Spacing metric is used to determine if the points in the generated Pareto front are evenly distributed in the objective space. The Spacing metric is given as:

$$Space = \sqrt{\frac{1}{|PF_{prod}| - 1} \sum_{i=1}^{|PF_{prod}|} (\bar{d} - d_i)^2} \qquad (2\text{-}8)$$

where

$$d_i = min_j(|f_1^i(x) - f_1^j(x)| + |f_2^i(x) - f_2^j(x)|)$$

and $i, j = 1,2,\ldots,n$; n is the number of vectors in PF_{prod}; and $\bar{d}$ is the mean of all d_i. $Space = 0$ means that all the members of the produced Pareto front are evenly spaced.

Overall Non-dominated Vector Generation Ratio:

The Overall Non-dominated Vector Generation Ratio (ONVGR) metric is used to evaluate the convergence property of a produced Pareto front PF_{prod} and is measured the ratio of the total number of vectors found in the produced PF to the number of vectors found in the

reference Pareto front PF_{ref}. The following equation as reported in [Knowles 2002] is used to calculate it:

$$ONVGR = \frac{|PF_{prod}|}{|PF_{ref}|} \qquad (2\text{-}9)$$

The measure is calculated at a phenotype level and a larger ONVGR value indicates a better PF.

Error Ratio:

The Error Ratio (ER) metric is used to evaluate a produced Pareto front PF_{prod} in terms of the number of vectors in the PF_{prod} that are not members of the reference Pareto front PF_{ref}. ER is calculated according to following equation as reported in [Knowles 2002]:

$$ER = \frac{\sum_{i=1}^{n} e_i}{n} \qquad (2\text{-}10)$$

Where $e_i = 0$ when the vector i is an element of PF_{ref} and $e_i = 1$ otherwise; *n* is the number of vectors in the PF_{ref}. The measure is calculated at phenotype level and a lower ER valued PF_{prod} is considered as a better PF. If the $ER = 0$ then PF_{prod} is same as $PF_{ref}$and if the $ER = 0$ then none of the points in PF_{prod} and $PF_{ref}$are common.

2.4 Search Based Software Testing

Incorporation of metaheuristic techniques has started a new direction for automated software testing and according to a recent survey, it accounted for the largest proportion of publications (70%) in search based software engineering [Harman, Mansouri and Zhang 2009]. Metaheuristic techniques also known as search based techniques are usually applied to problems where the solution space is very large and no known exact algorithm can produce good solutions (global optimal) in a reasonable amount of time. Software testing is complex, expensive but forms an integral part of the software development process. The efforts to reduce the cost of testing through automated test generation and execution have been around for more than two decades now. Initial attempts on automating the test generation process were primarily focused on using random and systematic (i.e. goal and path oriented) approaches [Roger and Bogdan 1996]. The random technique can produce a large number of test cases, however, it can never guarantee complete coverage and is prone to produce redundant test cases. The systematic approaches enable generating test cases according to given a criterion.

However, they are also prone to produce infeasible test cases [Edvardsson 1999]. Several attempts have been made to improve both test case generation approaches by incorporating heuristic techniques. The test optimization techniques can be classified along several dimensions, such as source or reference-based (e.g. code and specification), purpose-based (e.g. mutation, temporal and stress), objectives-based (e.g. single and multi), test stage-based (e.g. optimal generation and optimization), optimization mode-based (e.g. online and offline) and heuristic-type based (e.g. local and global search) dimensions. Table 2-8 presents the overview of two orthogonal dichotomies of test optimization techniques. The following is the summary of some of the test optimization techniques and related work.

Table 2-8: Taxonomical overview of test optimization techniques

	Optimal Test Suite Generation	**Test Suite Optimization**
Code-based	• Structural testing, • OO class testing, • State-based testing, • Aspect oriented testing	• Test case prioritization • Test case selection • Test suite minimization
Speciation-based	• Test sequence generation • Domain testing	Same as code-based test suite optimization

2.4.1 Optimal Test Suite Generation

Test generation is one of the most extensively investigated topics in software engineering [Bertolino 2003]. Search based software engineering is no exception and several search based test generation techniques have been reported.

Exhaustive testing of non-trivial software is impractical as the number of possible test cases can be astronomical due to the infinite test space (e.g. input space, state space and path space). Typically, there are two types of approaches, sampling and folding, which are followed to address the test case explosion problem [Young and Taylor 1989]. The first approach is based on using random sampling to use only part of the infinite test space (e.g. random testing [Hamlet 2006]). The second approach is based on reducing the test space by folding or abstracting away some detail of inputs, paths or states. For instance, the equivalence class testing requires input data from each equivalence class. The subsumption hierarchy of test adequacy or selection criteria indicates the folding relationship of their test space. For example, test space of edge coverage criterion (graph-based criterion) includes paths that satisfy node coverage criterion as

well. A test criterion that subsumes another criterion in a particular criteria family (i.e. graph-based criteria) means a larger test space and more test cases needed to cover it.

Generating an effective test suite which efficiently covers the complete search space of candidate test cases according to a given criterion for the software under test is a computationally hard problem. Producing a test suite with minimum number of test cases and complete coverage of a given criterion (e.g. all transitions and basic paths) is analogous to Set Cover problem which is NP-hard. It means that no algorithm exists that can resolve such a problem exactly in polynomial time. Heuristic techniques often used for problems where the polynomial-time algorithms are not feasible due to the size of the search space or where the search space grows exponentially with the increase in problem size and the conventional exhaustive search algorithm does not scale well.

In order to generate test cases using a heuristic technique, the test adequacy or selection criterion is formulated as a fitness function and the search space of possible test cases is encoded in the appropriate representation. Several metaheuristic based test generation techniques have been reported and these can be classified along a conventional testing dichotomy of structural and functional testing.

2.4.1.1 Structural Test Generation

The aim of structure based test generation is to generate test cases with maximum coverage for a given structural coverage criterion. A heuristic technique uses the fitness function to evaluate individual candidate test case for fitness according to the given coverage criterion. Generally, the fitness function is defined as a distance value between a test case and the target program predicate that it needs to execute [Sthamer 1996; Wegener, Baresel and Sthamer 2001]. For example, consider the selected criterion is branch coverage, which requires that a test suite contains at least one test case which causes each feasible branch of the program under test to execute. The condition to execute a branch is x==y. With the fitness function |x-y|, that branch will be executed when this function is evaluated to a minimum [McMinn 2004]. For extensive review of the structural test case generation using metaheuristic, readers are referred to [McMinn 2004] and [Harman, Mansouri and Zhang 2009]. Formally the optimal structural test generation can be defined as follows:

Given a program P and a set of program elements $r_1, r_2, \ldots, r_n$ that must be traversed to provide the desired test coverage of the program, test suite generation is to find a test suite T that satisfies all of the r_i.

2.4.1.2 Functional Test Generation

The aim of functional test generation is to find test cases that fulfill the functional coverage criterion. The heuristic based functional test generation techniques can be categorized into test data and test sequence generation. Test data generation techniques are used to generate input data from the functional specification. Domain testing and combinatorial testing are test data generation techniques. In combinatorial testing, all possible t-way data combinations for input parameter are tested. Shiba, Tsuchiya and Kikuno [2004] proposed two test data generation algorithms based on GA and ant colony optimization algorithm (ACO) to produce a small t-way test set for combinatorial testing.

The test sequence generation techniques derive test sequences from the functional or behavioral model, such as Statechart and Markov Usage Model (MUM). State-based testing is a functional testing technique for generating test cases to identify faults in the implementation of classes or components modeled in logical states. Li and Lam [Li and Lam 2005] proposed an ACO technique for generating minimal test suite covering all states from the UML Statechart. MUM is usually used in software testing for identifying the relative importance of various transition and states of the software and prioritizing associated test cases accordingly. Doerner and Gutjahr [2003] demonstrated the usage of ACO in extracting the test sequences for functional testing from a MUM. Formally the optimal functional test generation can be defined as follows:

Given a specification S and a set of specification elements $r_1, r_2, \ldots, r_n$ that must be traversed to provide the desired test coverage, test suite generation is to find a test suite T that satisfies all of the r_i.

2.4.2 Test Suite Optimization

During ongoing software maintenance, the test suite grows and requires regular maintenance. New test cases are added wherever needed to exercise the modified part of the program. However, the changes may render some of the existing test cases invalid or redundant and which are usually difficult to identify and remove. Executing invalid test cases may produce false test results. Redundant test cases may create test suite maintenance problems as well as wastes testing time and resources. Moreover, due to the limited time and resources available for testing, executing the entire test suite is not always feasible. In such circumstances, test cases with higher effectiveness and/or lower cost are selected or prioritized for execution. Approaches to find feasible and economical sets of test cases are usually classified into three types: test case selection, test case prioritization and test suite minimization.

2.4.2.1 Test Suite Minimization

Test suite minimization techniques are used for maximizing the effectiveness of a test suite at minimal cost by removing those test cases that owing to code or requirement modifications over time have become redundant or obsolete with respect to the original test objectives. Generally test cases produced for a test requirement fulfils other requirements as well which in other terms means that a requirement might be satisfied by more than one test case. The premise of reducing a test suite is that running the minimal set of test cases that give the same coverage as all of the test cases can significantly reduce the cost of execution and maintenance with negligible effect on the fault detection capability. Several empirical studies have confirmed the advantages of test suite minimization [Chen and Lau 2001; Wong et al. 1997]. Formally, it is defined by Harrold, Gupta and Soffa [1993] as follows:

Given a test suite T, a set of test case requirements $r_1, r_2, \dots, r_n$ that must be satisfied to provide the desired test coverage of the program, and subsets of T, $T_1, T_2, \dots, T_n$ one associated with each of the r_i such that any one of the test cases t_j belonging to T_i can be used to test r_i. Test suite minimization is to find a representative set of test cases from T that satisfies all of the r_i.

2.4.2.2 Test Case Selection

Many testing techniques have been proposed to selectively choose test cases according to a predefined test objective. However, from the regression testing perspective, test case selection is about finding test cases to target only the changed code or functionality. The basic idea of selective testing is to execute a relatively small and effective set of test cases and to reduce extraneous testing. Generally, test cases are selected using specification or code oriented dependency analysis which allows testers to choose only those test cases that target the modified code or functionality. Simplistic dependency analysis or more sophisticated techniques such as data-flow analysis can be used. Many studies have shown the effectiveness of dependency analysis and heuristic based test case selection [Binkley 1995; Bogdan and Ali 1998]. It is defined by Yoo and Harman [2007] as follows:

Given a program P, its updated version P' and a test suite T, test case selection is to find T'such that $T' \subset T, (\forall t \in T)[t \text{ is modification\_traversing} \implies t \in T']$.

A test case t is considered modification_traversing if it covers at least one part of the updated code or functionality in P'.

2.4.2.3 Test Case Prioritization

Prioritizing test cases has become increasingly important in the past two decades and has also attracted significant research attention. The basic idea of test prioritization is that given the effectiveness (e.g. coverage, fault detection capability) of each test case one can prioritize the execution of the test cases such that the testing achieves the maximum test objectives at a faster rate. For example, if you know the coverage of each test case, you can prioritize the tests in a way that by executing them in that order you can achieve the highest coverage in the least amount of time [Rothermel and Elbaum 2003]. Several empirical studies and industrial experiences appear to confirm this approach [Rothermel et al. 1999; Srivastava and Thiagarajan 2002]. Rothermel et al. [1999] defined the test prioritization as follows:

Given a test suite T, the set of permutations of T, PT; a function from PT to real numbers f, test case prioritization is to find $T' \in PT$ such that $(\forall T'')(T'' \in PT)(T'' \neq T')[f(T') \geq f(T'')]$.

The set PT represents all possible prioritization orderings and the function f quantifies and assigns values to these orderings according to a given objective.

2.4.3 Code and Model-based Test Suite Optimization

Existing work in test suite optimization that are similar to test generation techniques can be categorized into code-based and model-based techniques. Code-based optimization techniques use source code based coverage information collected during the execution of the software under test (SUT) with test cases as a fitness value to guide the optimization process. Model-based optimization techniques rely on models of the SUT for test suite optimization.

Optimization of test suite with model-based techniques is relatively inexpensive in comparison to code-based techniques [Bogdan et al. 2005]. Therefore, the optimization overheads of model-based techniques are smaller than that of code-base techniques which make it a more feasible option for system level test suite optimization. The code-based techniques are language dependent and are effective for unit level testing mostly. Moreover, they do not scale well to system level testing. A recent study [Korel and Koutsogiannakis 2009] found that model-based prioritization techniques are superior to code-based techniques in terms of the fault detection capability at system level testing. Furthermore, incorporating requirement information in optimization gives model-based techniques an extra advantage which can be utilized to eliminate many overlapping test cases without sacrificing the integrity and thoroughness of the test suite.

2.4.4 Multiobjective Test Generation and Optimization

Most of the conventional testing techniques derive a large number of test cases to achieve a given coverage objective and often without considering the potential or allocated cost and time. Typically, software projects are constrained by limited time and resources, so a trade-off is often sought between the testing objectives. Selecting a small but adequate subset of test cases according to a given objective can potentially save significant test resources. Moreover, studies have shown that although more testing improves the productivity (satisfying test objective i.e. code and requirement coverage) initially, it eventually reaches a point of diminishing returns where more testing stops improving the productivity [Chen and Lau 2001; Wong et al. 1995].

Several researchers have incorporated heuristic techniques in the testing process in order to generate or select a minimal number of test cases while fulfilling the multiple test objectives [Lakhotia, Harman and McMinn 2007; Oster and Saglietti 2006; Yoo and Harman 2007]. Patton et al. [2003] proposed a multiobjective GA and usage profile based approach to provide appropriate debugging information (i.e. fault nature and location) during system testing. Kiran and Harman [2007] incorporated multiobjective evolutionary algorithms to generate test data for branch coverage while maximizing the dynamic memory allocation. Zhang and Harman [2007] proposed a multiobjective evolutionary algorithm for test case selection.

2.4.5 Test Suite Minimization Approaches

To minimize a test suite, there are two approaches that are usually used to identify and handle redundant test cases. One approach is iteratively selecting test cases to maximize the coverage of artifact /feature of interest according to a given criterion. The other approach is selecting test cases to maximize the range of behavior exhibited by the original test cases.

2.4.5.1 Coverage-based minimization

With coverage-based minimization, test cases are selected to maximize the proportion of the program artifacts /features that are covered according to a given criterion (e.g. statement or branch). It simply tries to cover as many program artifacts of a given type as the original test suite with a minimum number of test cases. For example, in the minimization of a test suite generated for a branch coverage criterion using a greedy algorithm, at each of the iterations it will select a test case that covers the largest number of branches that were not covered by the previously selected test cases. In general, coverage-based techniques have the following key components:

1) A test suite from which the minimal set is selected.
2) A set of selection criteria and an associated set of artifacts/features
3) A feasibility function to determine if a test case contributes to the solution
4) The solution function to determine when the solution is final and complete.

2.4.5.2 Distribution-based minimization

With distribution-based minimization, cluster analysis techniques are used for selecting test cases based on the way they are distributed over the program artifact /feature space according to the criterion [Dickinson et al. 2001; Masri et al. 2007]. At the start, k clusters of test cases are identified on the basis of their execution profile and then tests are selected using a random or guided sampling strategy. For clustering a dissimilarity metric (e.g. coverage of program artifacts or execution count) is used and calculated by an n-dimensional Euclidean distance between each pair of test cases in terms of coverage of program artifacts or features. It indicates their degree of dissimilarity. For example, in order to minimize a test suite generated for branch coverage criterion, test cases are first sorted into various clusters according to the difference in the branches they cover. Secondly, test cases are selected from each of the clusters either randomly or using some guided selection mechanism. The main steps involved in distribution-based minimization are as follows:

1) Sort test cases into clusters.
2) Define similarity/dissimilarity criterion
3) Select test cases from each cluster (e.g. one-per-cluster sampling or adaptive sampling).

A number of studies have been reported that enhance the efficiency and effectiveness of testing by eliminating the redundant and obsolete test cases without compromising the fault detection capabilities [Binkley 1995; Harrold, Gupta and Soffa 1993; Jeffrey and Gupta 2007; Jennifer et al. 2004; Offutt et al. 1995; Yanping et al. 2007]. Harrold, Gupta and Soffa [1993] formulated the test suite minimization as a hitting set problem and proposed a heuristic for finding the smallest test suite. Offutt et al. [1995] proposed a minimization heuristic and conducted an empirical study to reduce a regression test suite with respect to mutation and statement coverage criteria. Chen and Lau [2003] proposed a divide-and-conquer approach to minimize the size of a test suite generated through a random technique. It is based on an exact algorithm which is usually considered infeasible for industrial scale applications. Xie et al. [2004]

have developed a framework for the optimization of object oriented unit tests by eliminating redundant test cases. They also proposed a number of redundancy detection approaches and applied it in detecting and removing redundant test cases. Jeffrey and Gupta [2005] proposed a technique to minimize a test suite with selective redundant test cases. Tallam and Gupta [2005] adapted the greedy algorithm to minimize a test suite by removing redundant test cases. They used the Concept Analysis technique to identify the groups of objects and attributes and their implications and then exploit that information for test suite reduction. Jeffrey and Gupta [2007] proposed a test suite reduction approach while selectively retaining redundant test cases in order to improve the fault detection capability. Wong et al. [1999] and Rothermal et al. [1998] have empirically studied the effects of test suite reduction on its fault detection capability. According to Wong et al. [1999], the reduction in test suite size has no or negligible effect on it fault detection capability. However, Rothermal et al. [1998] found that the reduced test suite can compromise the fault detection capability of a test suite. The conflicting results of both studies have rendered the test suite reduction a controversial topic. Other studies [Chen and Lau 2001; Heimdahl and George 2004] on this topic found the similar opposite results. So far, the proposition about the compromised fault detect-ability of an optimized test suite is limited to the code-based regression test suite. Few studies in the category of functional testing regarding the effect of test suite minimization on its fault detection capability were conducted [Chen and Lau 2001; Chen and Lau 2003; Heimdahl and George 2004]. Chen and Lau's [2003] study is related to domain testing and reported no difference in the fault detection capability of the reduced or the original test suite. The other study by Heimdahl and George [2004] was related to model-based testing and they found that test suite minimization can compromise the fault detection capability of a test suite. However, due to the enormous differences between modeling languages and the associated test generation mechanisms, these results are not necessarily applicable to other model-based techniques. Nevertheless, this fact highlights the need for further study.

Chapter 3 Model Transformation

The previous chapter provides the review of the literature related with the study. This chapter reports the issues related to transformation of an AD model into a CPN model and describes a rule-based transformation methodology developed in this study. It also describes four case study AD models and reports an experiment to evaluate the proposed transformation methodology using the case models.

Model transformation is a core mechanism in Model Driven Architecture (MDA) for defining an automatic, valid and consistent transformation between source and target models. The transformation requires a set of production rules or mapping patterns for translating one or more elements in the source model into one or more elements in the target model. Transformation is the application of production rules to the source model until no more production rules are applicable and which finally yields the target model. Various model transformation techniques have been developed to produce models in the same or different technological spaces i.e. Common Warehouse Metamodel (CWM), XML Metamodel Interchange (XMI), eXtensible Stylesheet Language Transformation (XSLT), graph transformation and graph grammar. The Object Management Group (OMG) has approved various standards to facilitate model interchange. CWM, XMI and MOF QVT are the OMG's adopted meta-model, data and model level transformation respectively. XSLT is the World Wide Web Consortium (W3C) endorsed standard for interchanging documents in XML format.

UML provides various modeling formalisms to specify different views of a system in models. These formalisms are classified into structural and behavioral types based on their capabilities to reveal different aspects of a system. Diagrams associated with these formalisms provide graphical projection of the models depicted in them. The Interaction Diagrams (Sequence and Collaboration) are suitable for depicting inter-object behavior. However, they are not suitable for representing what happens inside an object. The Statechart Diagrams are excellent for showing the internal behavior of an object but they are not useful in depicting the inter-object interaction. The ADs are excellent in depicting both inter and intra-object behavior. It has wide application scope, ranging from modeling embedded hardware and software to business process modeling for distributed and concurrent systems. Flow-oriented modeling in

ADs do not require new skills as it is very similar to flowcharting, and is deemed intuitive for depicting program and process flow logic. The provision of hierarchical activities enables the development of large and complex models in a top-down, bottom-up or a combination of both top-down and bottom-up ways. In software modeling, it can be used to model the detailed flow-oriented specification of a method or operation for a class or use case. In the case of a business process or workflow modeling, it can be used to model the detailed specification of complex use cases involving many actors or business organizations.

In UML version 2.x (UML2), AD has gone through a major revision, resulting in several syntactical and semantical changes. Although, its new multilayered syntax is complex, it is quite rich in notations and expressiveness. This makes it suitable for modeling both high level conceptual diagrams at the earlier stages to low level detailed diagrams at the later stages. The syntactic elements with graphical notations such as action, activity, object and control nodes allow modeling both simple control and data flow behavior to complex behaviors such as hierarchies, synchronous or asynchronous communication, concurrency and exception handling. These features serve to reduce the modeling overhead [Bock 2003; OMG 2007]. Its new token-flow oriented semantic enables simulating the intended or implemented system behavior of the model. The action and activity firing rules, token insertion and removal rules on control nodes allow model navigation and status monitoring at any time during a simulation run.

As this study is focused on the application of AD in software testing, the following sections only describe software modeling related aspects of AD.

3.1 AD based Software Models: Notations, Syntax and Semantic

An AD is suitable for modeling the dynamic behavior of a system in terms of the computational steps connected by actions, data and control flows. They can be used to model the methods of a class or the detailed specification of a use case. In the following section, some of the basic notations and convention used to build behavioral software models using AD as defined in [OMG 2007], are presented. Although the notations are readily understandable, the syntax will be elaborated upon. Given the importance of semantics, some of the vital points of AD semantics in UML2 are also covered in the following sections.

An AD model consists of diagrams compose of a small set of graphical artifacts. Figure 3-1 depicts the basic set of artifacts that are provided for modeling with AD in UML2 (based on Enterprise Architecture [EA 2008] modeling notation).

3.1.1 Functional Artifacts:

There are two kinds of basic functional artifacts that take some input, execute (perform some operation on control or data input that they receive) and produce some output.

1) Activity: An activity represents a complex behavior or functionality provided by a program, which at a detailed level, can specify the sequence and condition for execution of low level behaviors (also referred to as actions). The initiation of an activity is depicted by an Initial node which is then linked to other artifacts by directed connections, thereby indicating an ordered flow of execution. The execution of an activity stops at an Activity Final node which concludes the behavior. However, a thread of execution within an activity ends when the flow reaches a Flow Final node. A Structured Activity node allows the specification of complex logic in multiple levels of sub-activities and may be represented as a composite activity as depicted in Figure 3-1. Although an activity can invoke (call) other activities (through call-behavior-action) at run-time and can create call-hierarchies, it is different to Structured Activity node. A sub-activity in a structured (composite) activity starts executing as soon as the super-activity commences execution. The super-activity remains active until all sub-activities have completed execution.

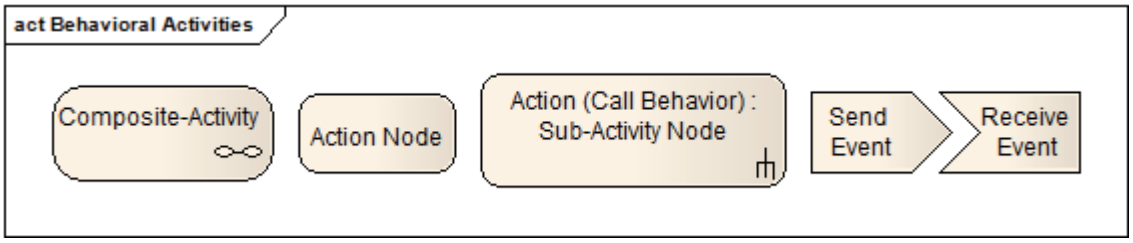


Figure 3-1: Activity Diagram Behavioral Artifacts

2) Action: An action node is used to represent a primitive or a low level behavioral or functional aspect of the system which can be atomic as well. Typically, an action node has some inputs and outputs and is active when it receives tokens (data) on its inputs and while it consumes those tokens. At the start of the its execution, the tokens are consumed (removed from the inputs) and an operation is performed on them; ultimately producing some output which in turn goes on to become the input for the next action node in the sequence. The rounded-corner rectangle notation as shown in Figure 3-1, represents an action node. UML2 specification has various types of primitive actions defined (e.g.

calling other behaviors (activities/ actions)). For instance, the *Call-Behavior* action represented by an action node with a trident symbol in the lower-right corner (see Figure 3-1) depicts calling other activities (behaviors/functions). The *Send Signal* action, shown as 'Send Event' node in Figure 3-1, creates a signal instance on receiving an input. It transmits the signal token to the target activity/ action which may trigger the execution of that activity asynchronously. Whereas, the *Receive Signal* action ('shown as Receive event') creates a token immediately on receiving the signal and passes it on to a successor node.

3.1.2 Control Artifacts:

Another set of artifacts in AD is the control nodes as shown in Figure 3-2. Control nodes pass the control and data tokens through the activity and coordinate the flow between nodes.

1) Initial node: An *Initial* node receives token (control) when the enclosing activity is invoked or started. It indicates the starting point for executing that activity. A control token placed in an initial node is forwarded to all outgoing edges. An initial node is not required for an activity and a direct link from the activity border (or input pins) to the start node can indicate the start of flow. However, for the sake of design clarity an explicit start node will be used as a convention.
2) Final node: The Final node also referred to as Activity Final is used to represent the termination of execution for an activity. A token reaching the Final node stops all action nodes in execution regardless of their state (e.g. waiting for event) and all of the synchronous flows (threads) in the activity. However, it does not affect the execution of any asynchronously invoked flows or separate instances of the same activity.
3) Flow Final node: The Flow Final node represents the termination of a flow in the activity. Arrival of a token at the Flow Final node only indicates the completion of a particular flow and does not affect the execution of other flows.
4) Fork/ Join node: A Fork node splits the flow into multiple concurrent flows. The incoming tokens at a fork are duplicated and offered to all outputs. At least one input and two outputs are required for a Fork node. A Join node synchronizes multiple flows and combines multiple parallel flows into a single flow. It offers a token on the outgoing edge only after receiving tokens on all of its inputs. The notation for both fork and join nodes is a line segment. The functionality of both Fork and Join nodes can be combined into single node using the same notation.

5) Decision/ Merge node: Both Decision and Merge nodes have the same diamond notation. A decision node has an incoming edge and multiple outgoing edges. Each outgoing edge (branch) from a decision node carries a guard condition that is evaluated at runtime to determine if a token can continue along the edge. The guard conditions on outgoing edges are evaluated for each individual token arriving at the decision node to determine which edge the token will traverse. Each token can only follow one outgoing edge from a decision node. A merge node has multiple incoming edges and one outgoing edge and serves to combine multiple flows without synchronizing. All tokens arriving at a merge node are immediately passed over to the outgoing edge.

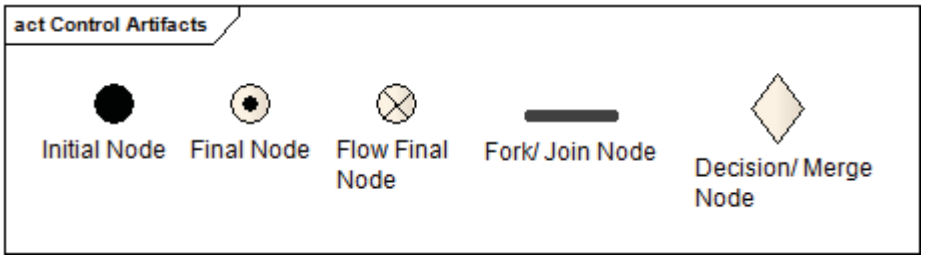


Figure 3-2: Activity Diagram Control Artifacts

3.1.3 Special Artifacts:

1) Objects: A rectangle in an AD means a data-object that can be accessed by actions during the program execution. Each of the data-object nodes in a model also has a unique identifier which is enclosed in braces as a prefix to the data-type name. All object nodes specify the type of value they can hold and if no type is specified, they can hold values of any type. Object nodes can hold more than one value at a time, and some of these values can be the same. Each object node has an upper bound which specifies the maximum number of tokens it can hold, including any duplicate values.
2) Partitions: Activity Partitions are provided in order to model the procedural flow and actions within an activity that has been grouped based on common grounds such as objects (e.g. classes, components, classifier or other responsible entities) that actually execute the action or provide the functionality. Partitions indicate what or who is responsible for actions grouped in a partition. An activity diagram may be visually divided into partitions each separated from neighboring partitions by solid vertical lines on both sides. In UML2, although the partitions

can be hierarchical, they do not affect the token flow of the model as there is no execution semantic defined for Partitions.

The visual notations for both object and partition artifacts are shown in Figure 3-3.

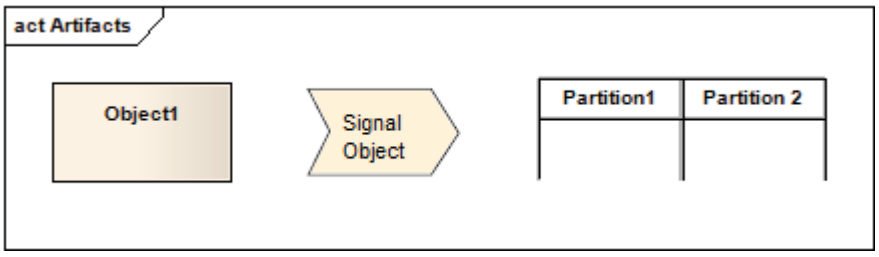


Figure 3-3: Object and Partition Notations

3.1.4 Connection Artifacts:

Activity nodes are connected by two kinds of directed edges:

1) Control flow edges connect control and behavioral nodes (activities and actions) and indicate that the node at the target end of the edge cannot start until the source node finishes. Only control tokens can pass along control flow edges.
2) Object flow edges connect objects nodes with behavioral types of nodes. The directed link between an action and an object node depicts the flow of data. If an action only reads from the object, then the arrow is directed toward the action node representing the flow from the object to the action only. If the action updates data in the object node, then the data flows from the action to the object node. Only objects and data tokens can pass along object flow edges.

The visual notations for the control and object flow edges are shown in Figure 3-4.

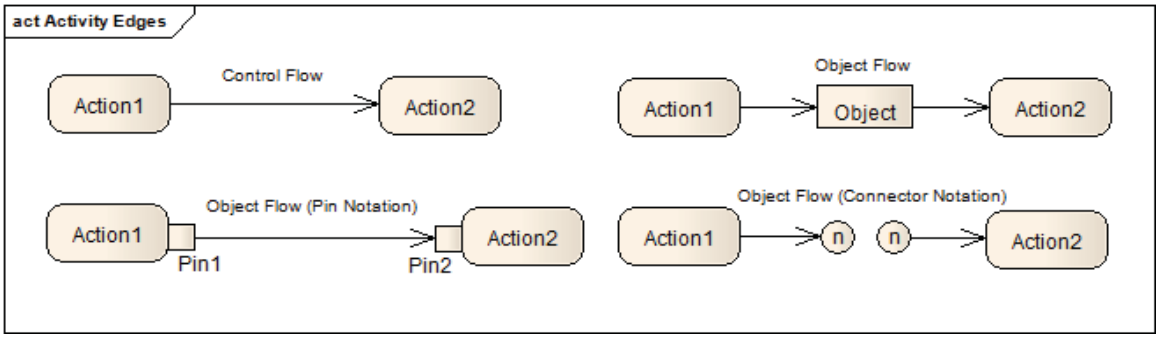


Figure 3-4: Activity Edge Notations

3.2 Case Studies

For the validation of the study, AD was chosen as a surrogate modeling formalism for depicting the case studies in all of the experiments. In the following sections, four case studies featuring models at various levels of complexity and composition are described.

3.2.1 Enterprise Customer Commerce System (ECCS)

The AD model shown in Figure 3-5 is adapted from [Koehler et al. 2005]. It describes an Enterprise Customer Commerce System (ECCS) and depicts the process of online purchasing of products that is comprised of two sub-processes, viz. the authentication process and the shopping process. The authentication process allows a customer to sign-in into the system. In the case of a new customer, system allows him or her to register first and then get back to the login screen. The authentication sub-process operates on user-specific data object for authorization. The shopping process facilitates the user to order the selected products and is also used for the account configuration by the user. The shopping sub-process operates on the data object that is related with shopping and user-specific information.

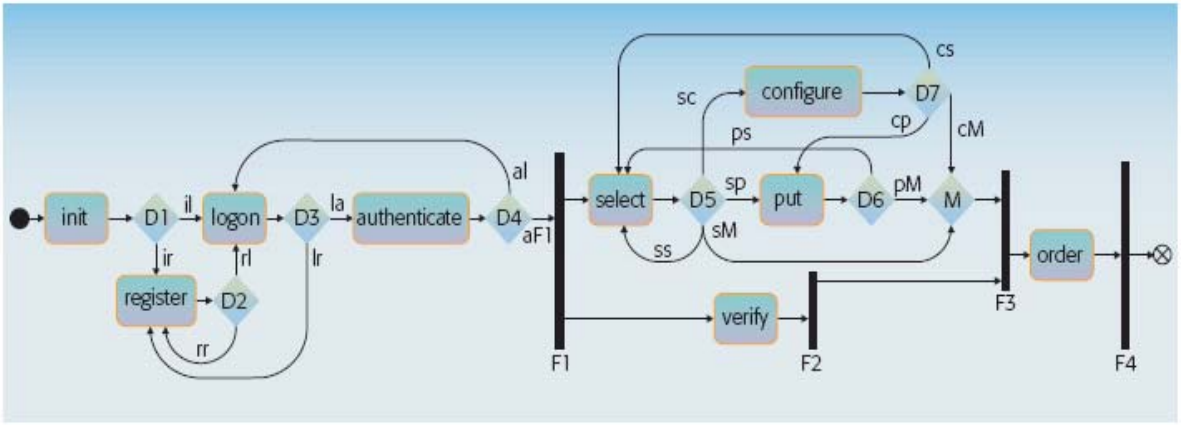


Figure 3-5: Enterprise Customer Commerce System [Koehler, Hauser, Sendall and Wahler 2005]

The guard conditions on the edges leaving decision nodes are denoted with mnemonic names according to the source-target actions. For example, the conditions for logon action from init, register and authenticate actions are denoted as *il, rl* and *al* respectively in the diagram. Overall, the ECCS model shows the order in which the data objects can be changed and the conditions when a specified action can operate on them.

The authentication sub-process comprises of *init, register, logon* and *authenticate* actions and D1, D2, D3 and D4 decision nodes. The *init* action initializes the process by creating an authentication data object for a new customer or loads it from the system for an existing

user. Existing users can log on straight to the purchasing system; while new users are requested to register first and then proceed to the shopping system after a successful authentication. Registration remains active for a specific period according to the system's security policy. Thereafter, the authorization may fail and the user with an expired registration needs to reregister (activate). On the successful authentication of a user, all shopping data related with the authorized user, such as payment information, open shopping sessions, previous and existing orders and shipping addresses, is made available for subsequent shopping processes.

The shopping sub-process comprises of *verify, select, configure, put* and *order* actions. The process has two concurrent threads, one for product selection allowing the user to select and possibly configure products, and a verification thread containing only a single '*verify*' action that ensures the security and integrity of the shopping process. If a user has an uncompleted shopping session with items placed in the shopping cart but not ordered, the cart will be load with previous session data by default and available to the user for modification. New products can be selected and configured until they are finally added to the shopping cart. The process is quite flexible and the user may go back and forth between the various actions. In order to place an order, the user is required to complete the product selection process via the *order* action where orders placed on products in the shopping cart may be submitted or cancelled. A *verify* action also needs to be completed successfully for the *order* action to be executed. No order can be submitted, and the process cannot be exited in a valid way without this information.

3.2.2 Automated Teller Machine (ATM)

The Automated Teller Machine (ATM) model is one typical case study in software engineering research. For this study, it has been adapted from a report [Chandler et al. 2006]. The ATM model describes the flow of control between a customer, the ATM and the Bank. This is a simple, high level view of these activities where a customer typically interacts with the system by performing some transaction. The transactions offered to a customer of the ATM such as withdraw cash, deposit funds, transfer funds and balance enquiry are depicted as separate activities and invoked using call-behavior actions. A transaction requires that a customer be authenticated via a bank card in order to access his/her account. One user session may involve one or more transactions and on the completion of a successful session, the ATM will eject the card from the card reader.

The top level ATM activity diagram as shown in Figure 3-6 depicts the possible actions at the system level and provides options for a customer to access specific functions. The specific selection functions are depicted by 'call-behavior' actions labeled with the activity name that it is

designed to invoke (e.g. the '*Deposit*' call-behavior action invokes the *Deposit Funds* activity as shown in Figure 3-7.

In the first interaction with the ATM, a customer inserts the ATM card into the machine. In response, the machine validates the card, and if it is recognized by the system, it prompts the customer to enter a PIN (Personal Identification Number) using the machine keypad, otherwise the card is ejected. After a successful authentication within three attempts, the system prompts the customer to select a transaction option. However, if the customer fails to enter a correct PIN in three attempts, the system retains the card and the customer is asked to contact the bank for new card. On the other hand, after each transaction the system prompts the customer to perform another transaction. A negative response to one of these prompts causes the system to close the session and to eject the card from the card reader.

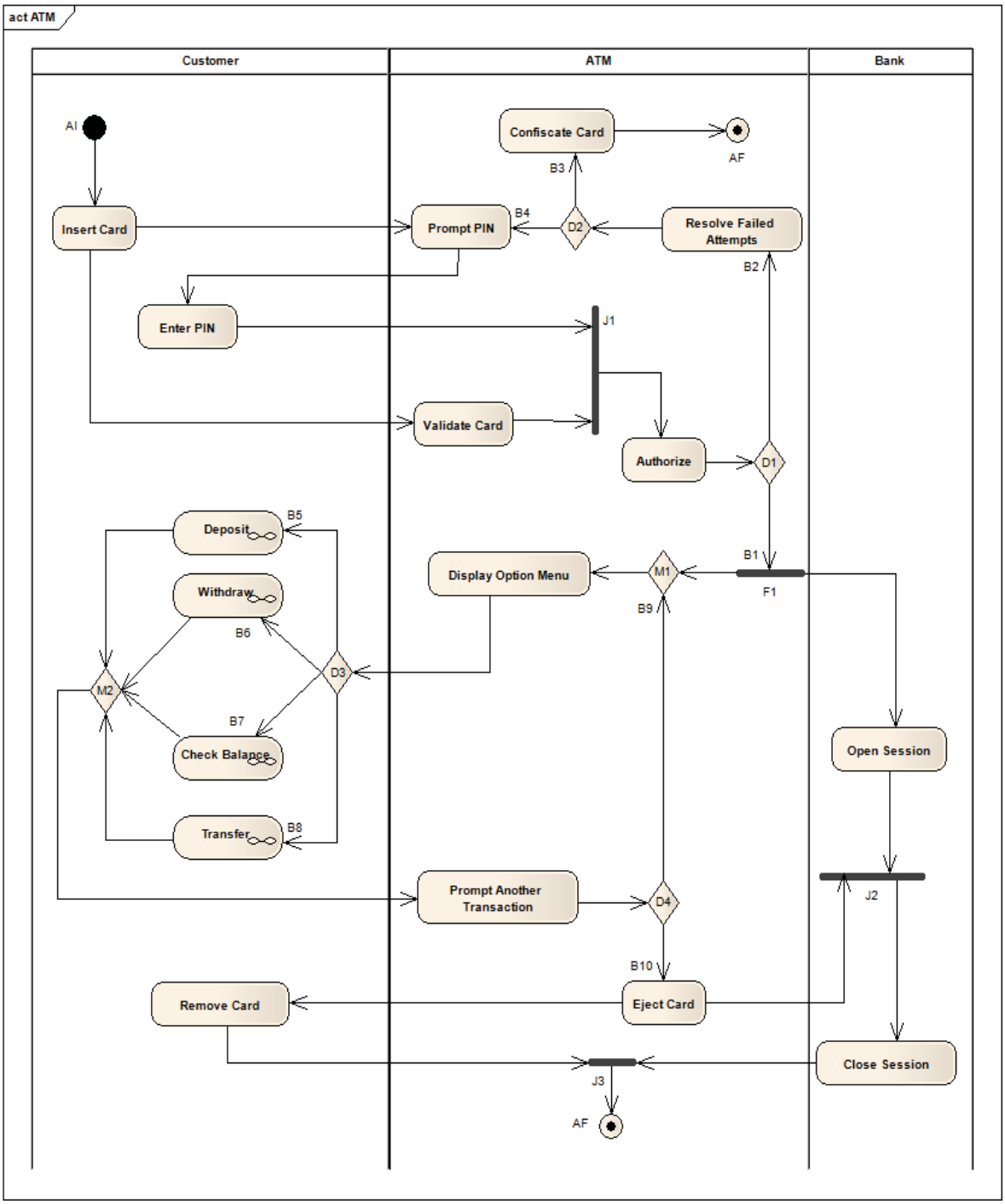


Figure 3-6: Automated Teller Machine Activity Diagram

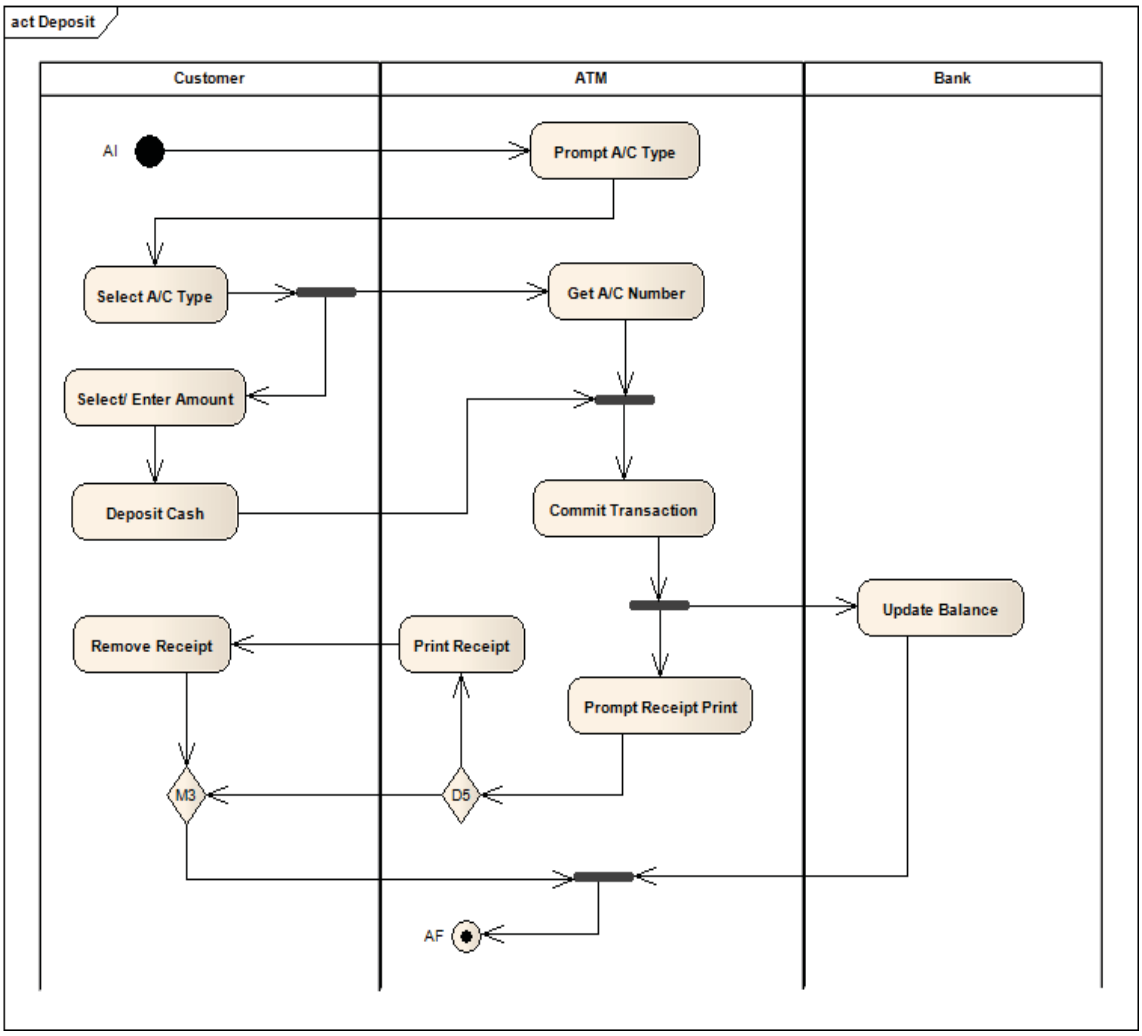


Figure 3-7: Sub-process Deposit

Deposit Funds: The *Deposit Funds* activity in Figure 3-7 illustrates the process flow of a deposit transaction using the ATM system. From the ATM system level activity, if the guard condition with 'B5' edge evaluates to true then control reaches the '*Deposit*' call-behavior action which will execute the *Deposit Funds* activity. The control flows through the activity, prompting the customer to enter the amount and deposit the cash, printing the receipt if needed and updating the account balance by tracing a route from nodes to edges. An interesting segment in this diagram is the concurrent region between the Fork2 and Join2 nodes. The upper thread in this region includes the *Update Balance* action which updates the '*balance pending*' field in the '*Account*' table at the bank while in the lower thread, the customer is asked if a receipt is required, involving a decision made within a concurrent region. In this activity, if the customer does not opt to print a receipt, then the lower thread branches past the print receipt action.

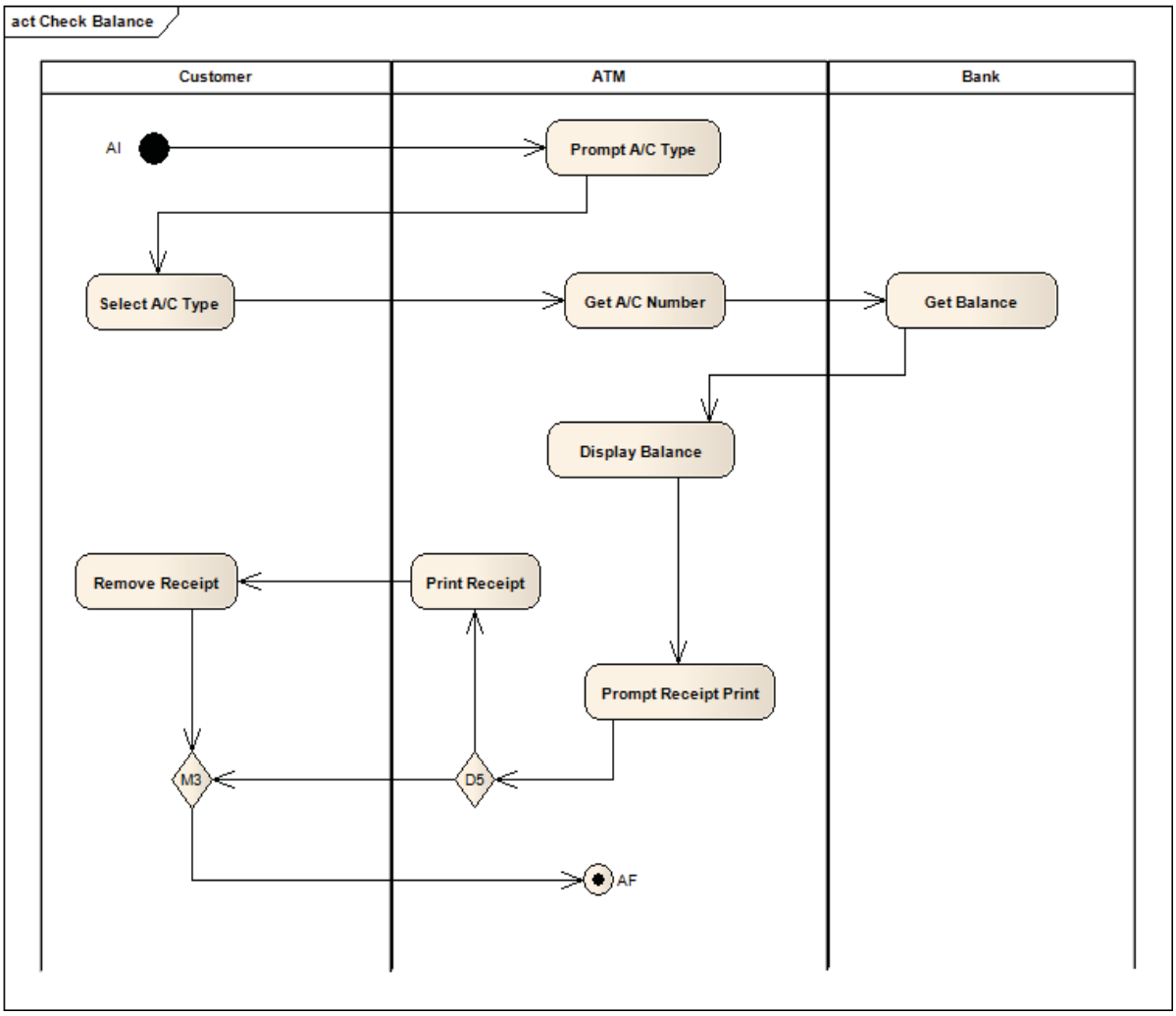


Figure 3-8: Sub-process Check Balance

Balance Enquiry: The *Balance Enquiry* activity in Figure 3-8 illustrates the functionality of finding and displaying the balance of the account linked to the card using the ATM. For this activity, the system prompts the customer to select the account type and then displays the account balance on ATM screen. The customer is then asked if account balance is required to be printed and prints a balance receipt is printed if the answer is affirmative. From within the ATM activity, the *Check Balance* action invokes the balance enquiry activity if the guard condition on the 'B7' edge is evaluated to true.

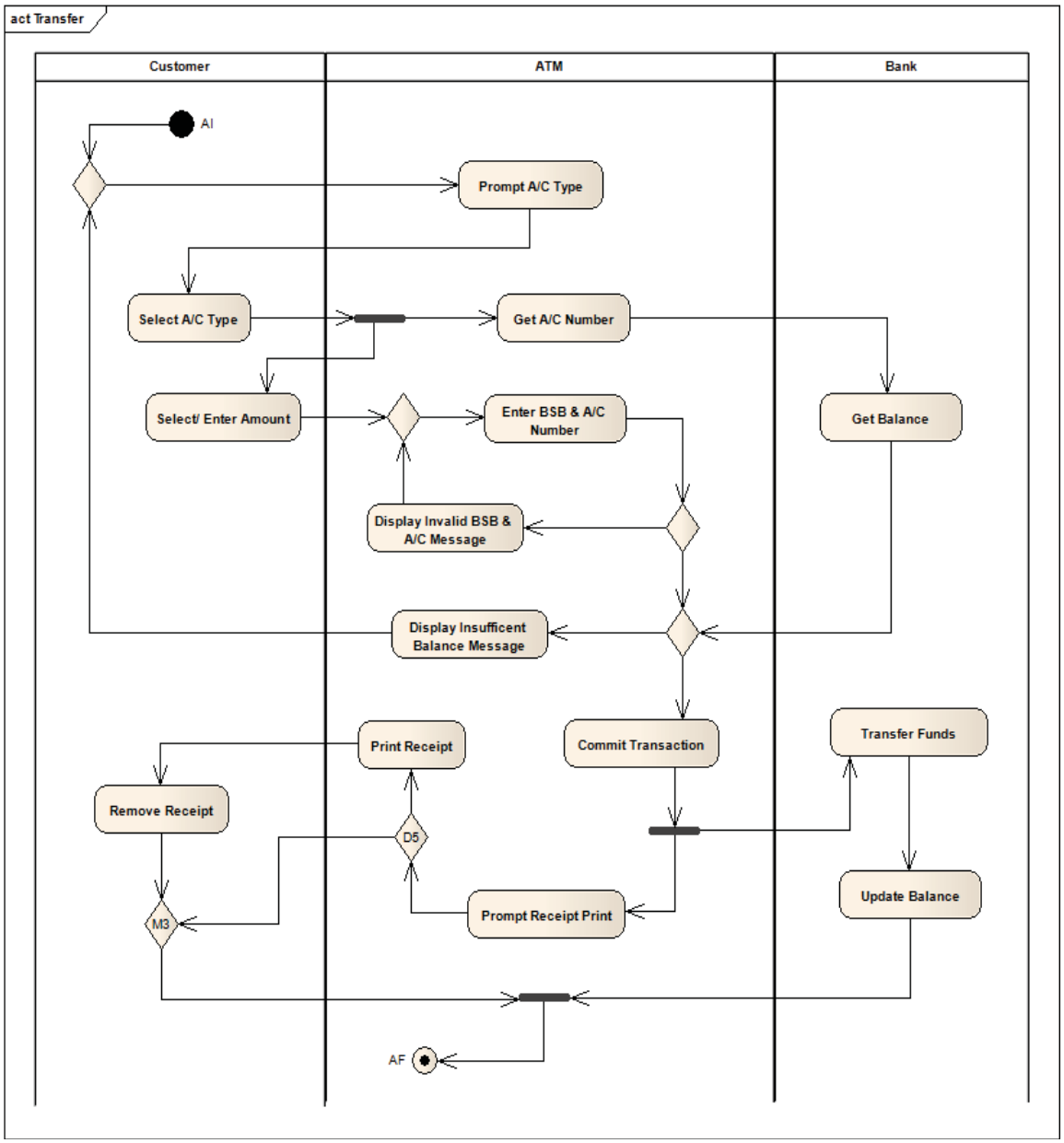


Figure 3-9: Transfer Funds Sub-process

Transfer Funds: Figure 3-9 illustrates the expected behavior of the *Transfer Funds* activity. It is triggered when the guard condition with the 'B8' edge in the ATM activity is evaluated to be true. On execution, it prompts the customer to select the account type, the transfer amount and to enter the BSB and the account number. The system then checks if the target account is valid, after verifying that the transfer amount is within the account balance.

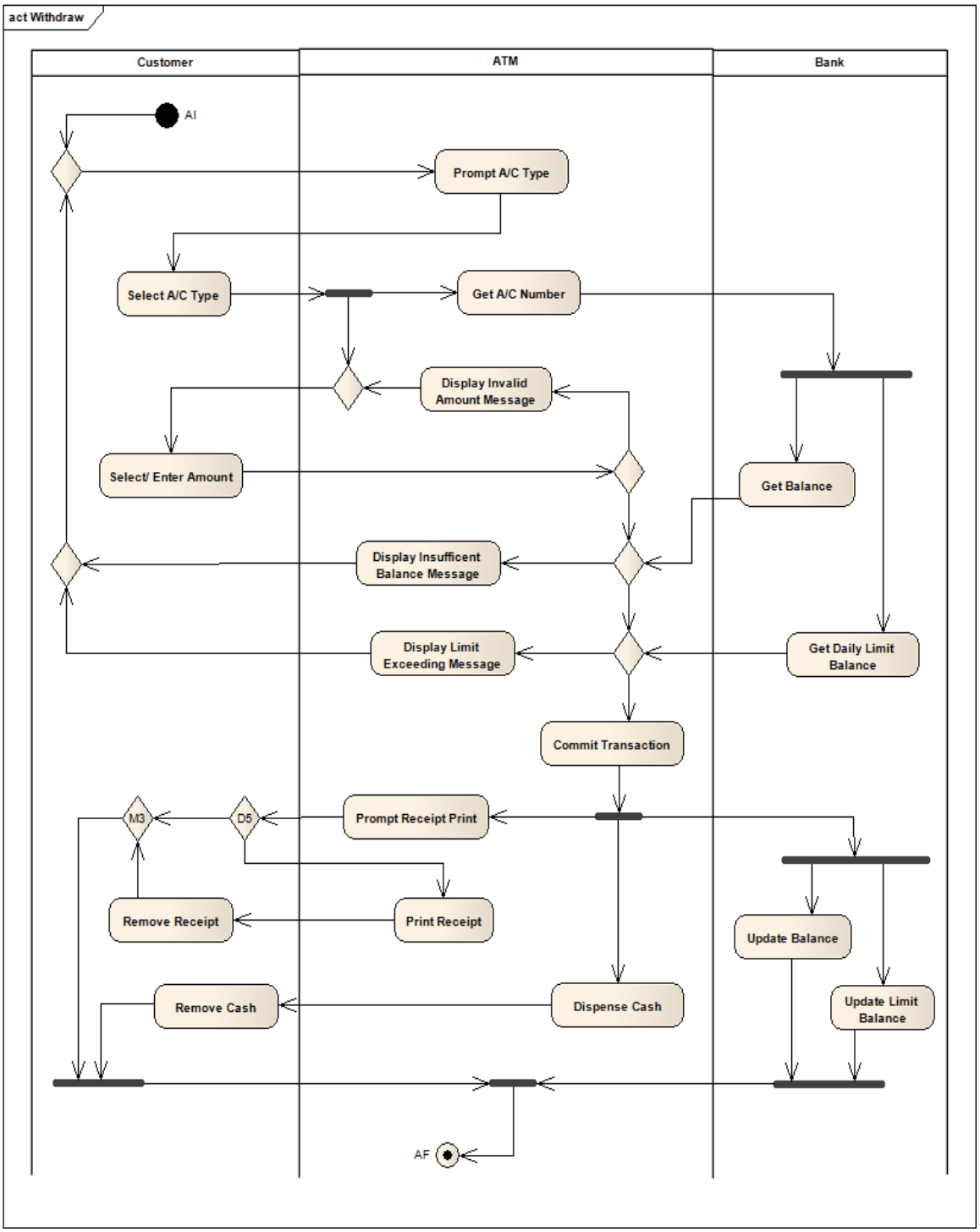


Figure 3-10: Withdraw Cash Sub-process

Withdraw Cash: Figure 3-10 depicts the actions that occur during a withdraw cash transaction in the ATM system. The activity is invoked from the ATM activity if the guard condition on the 'B6' edge is evaluated to true. This activity, similar to other low level activities,

performs various actions such as prompting the user for information like account type, withdraw amount and then updates the account balance on completion of the transaction request.

3.2.3 Two Models from Software Industry

The study also includes the two industry related models, namely Edit Trend Properties (ETP) and Delete Trend Properties (DTP) of a Trending Subsystem in an Intelligent Transport System (ITS). These two models were developed by the researcher while working for a software and permission was given to include them in this project. The Trending subsystem is responsible for the display of both historical and real-time trend of travellers, traffic flow and equipment operation in particular area or route. Basically any numerical data that is collected by the system can be displayed in a trend. A trend can display multiple plots for visual comparison of data. Two modes, time-based and source-based, are supported for comparative analysis and a trend can be configured to plot one or more data sources or a data source against two different time period simultaneously. A trend can display real-time data as well as archived data in multiple 2-dimensional plots. The x-axis typically represents time, while the y-axis represents the data value at a point in time. Historical trends are used to display archived data in a chronological plot for a given time period. Whereas, real-time trends with a start time set to the current date and time are continuously updated with new data at configurable sample rate. Data displayed in a historical trend can be scrolled forwards and backwards in time.

Trends are typically pre-configured and a user usually selects one or more pre-configured trends for display during system operation. However, a user with appropriate access privileges would be able to configure or remove a custom trend during system operation. Both models ETP and DTP describe the step by step editing and deletion of existing trending reports from archived or real-time data respectively. The product related terms in the models have been obfuscated for commercial neutrality purpose.

3.2.4 Edit Trend Properties (ETP)

The Trending subsystem allows the operator to graphically see the data trend by creating a new trend (graph) for a particular data point. The system comes with various trend templates that can quickly be attached to the new trend. A template specifies various trend properties such as graph type, sampling rate, scale and trend type. The system also allows the user to create new or modify existing trend properties (templates), either while viewing the trend or from the displayed list of saved trends. When a user selects a trend template to modify, the system verifies that the user has sufficient privileges to modify trend templates. On successful authorization, the system loads the details of selected trend template (Figure 3-12)

and displays it in the 'Trend Properties Dialog'; otherwise it shows a message, indicating insufficient rights for performing the requested operation. The user modifies the template details and selects to save. The system verifies the changes and stores the modified template. In case of invalid changes (e.g. the input data falls out of the acceptable ranges), the system advises the user of the error and allows him/her to correct it. The system also advises the user of the error if the name of the template is not unique or valid. On successfully saving the template the system loads the trend template (Figure 3-13) and refreshes the display. The system advises the user of the appropriate error if it fails to load the template or refresh the display.

Figure 3-11 to Figure 3-17 illustrate the control flow model of the Edit Trend Properties functionality in Trending subsystem.

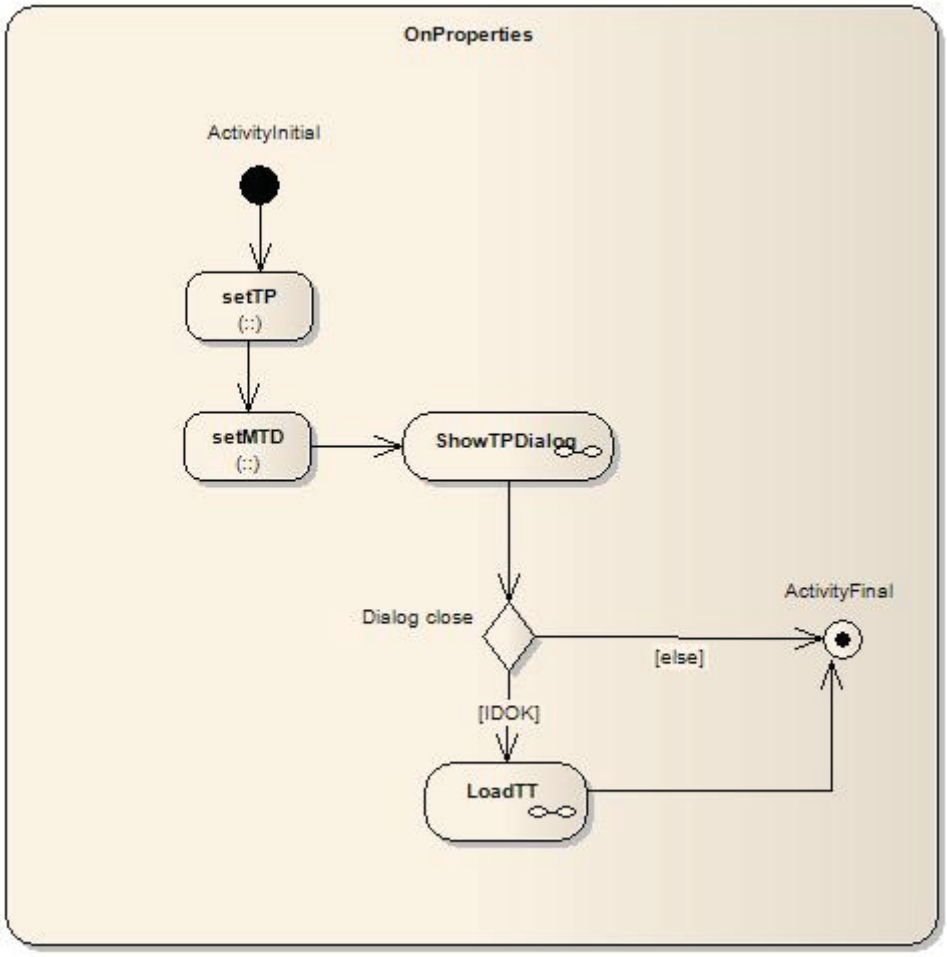


Figure 3-11: Edit Trend Properties (ETP)

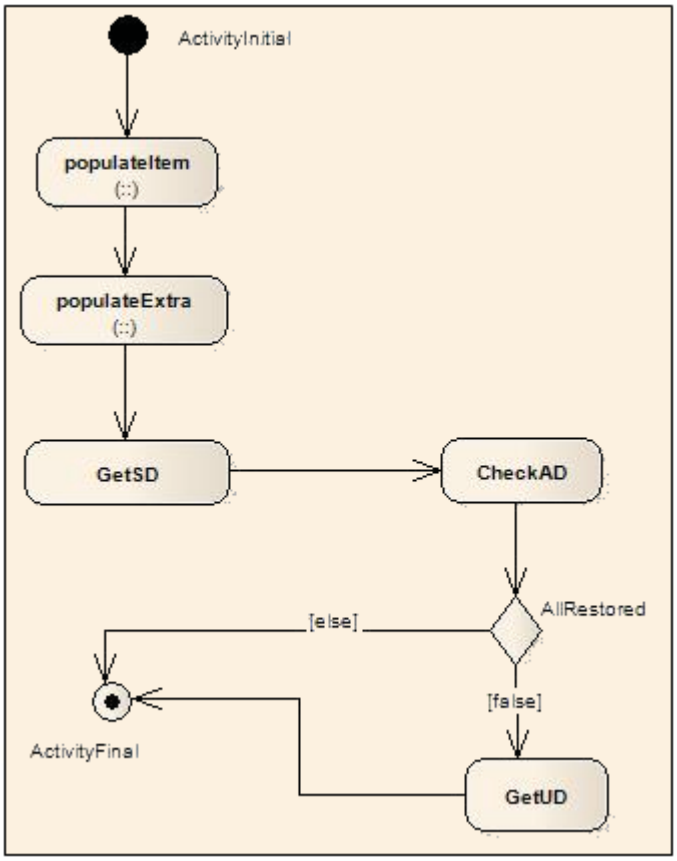


Figure 3-12: Show Trend Properties Dialog

In order to modify or display a trend template, its details need to be retrieved from the database. For that the system generates an appropriate SQL query and then executes it. If the database connection or SQL query fails then the system generates an appropriate error message detailing the problem. Otherwise the template details are uploaded from the database.

In order to reflect the changes in template properties into the loaded template, the system reloads the trend template. It calculates the data points applicable to the template and read associated data values from the database. New plots are created according to the given type and update the current display.

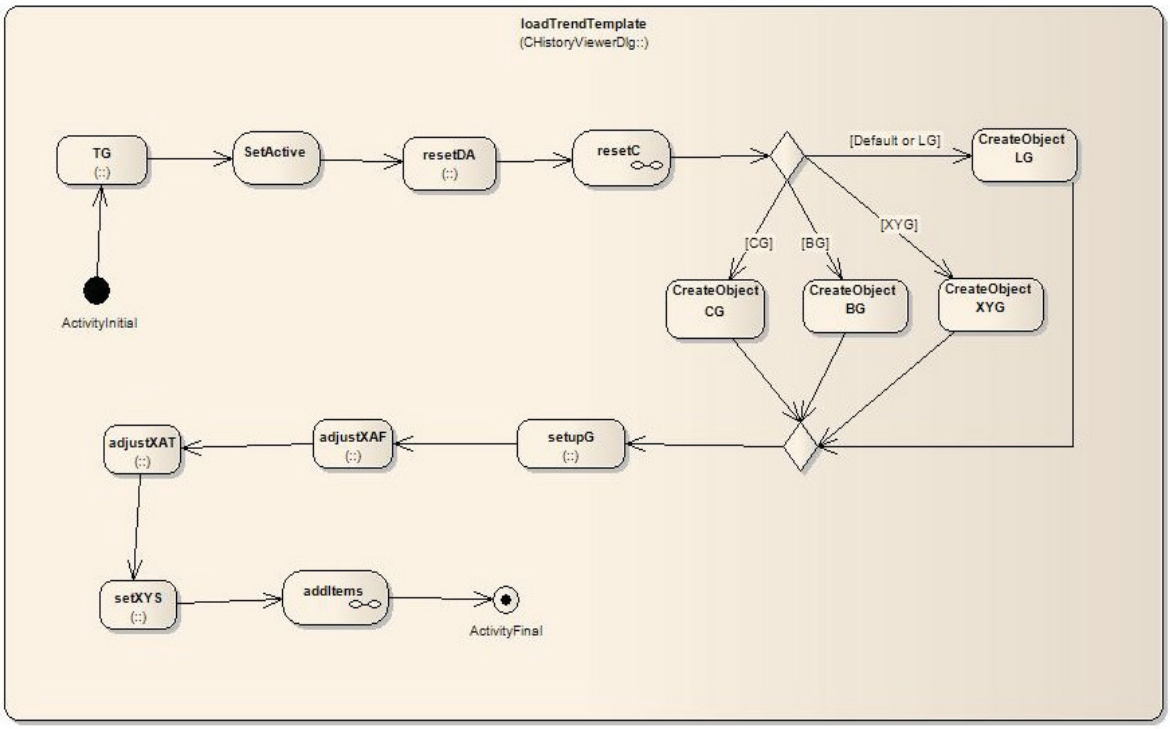


Figure 3-13: Load Trend Template

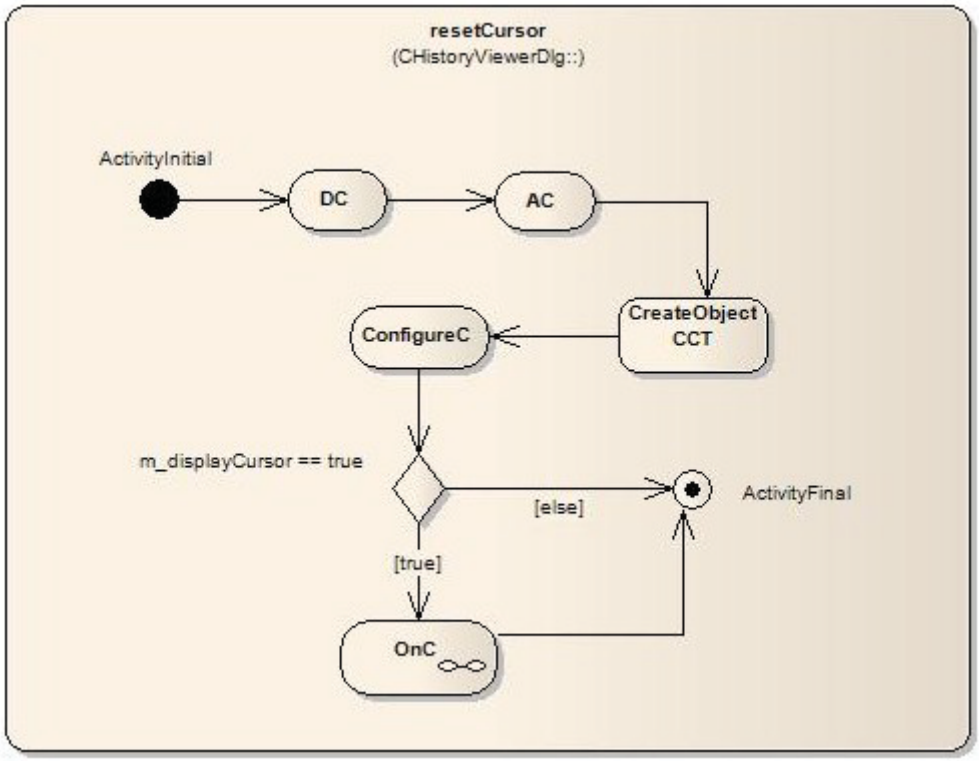


Figure 3-14: Reset Cursor

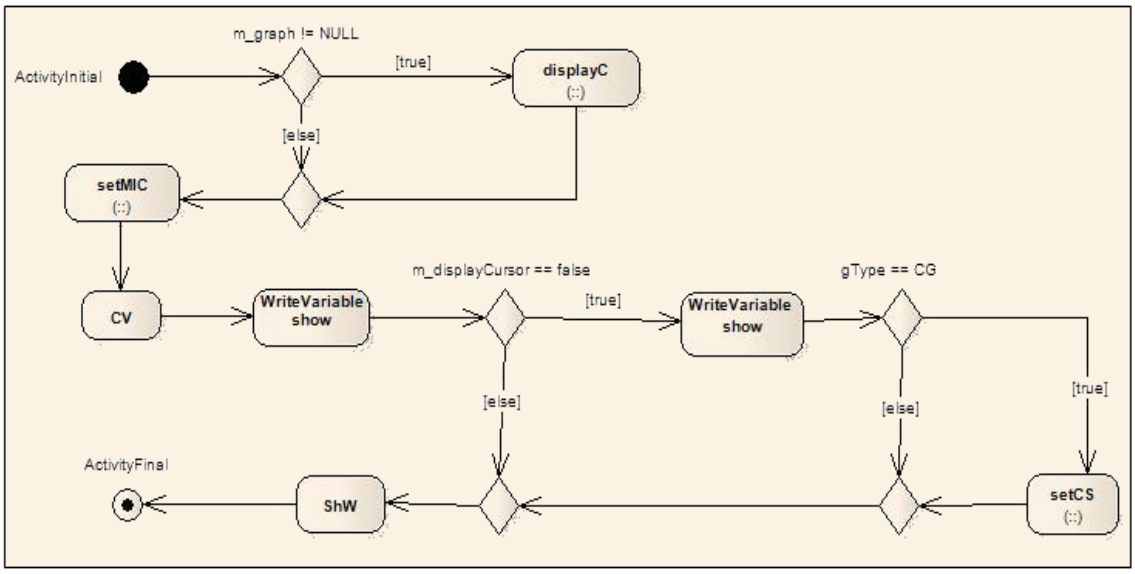


Figure 3-15: On Cursor

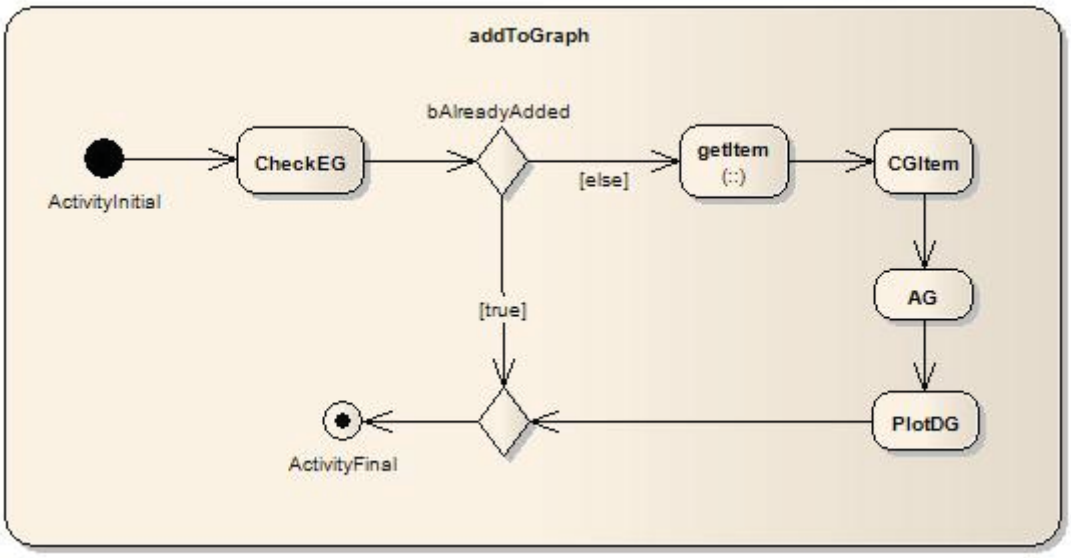


Figure 3-16: Add to Graph

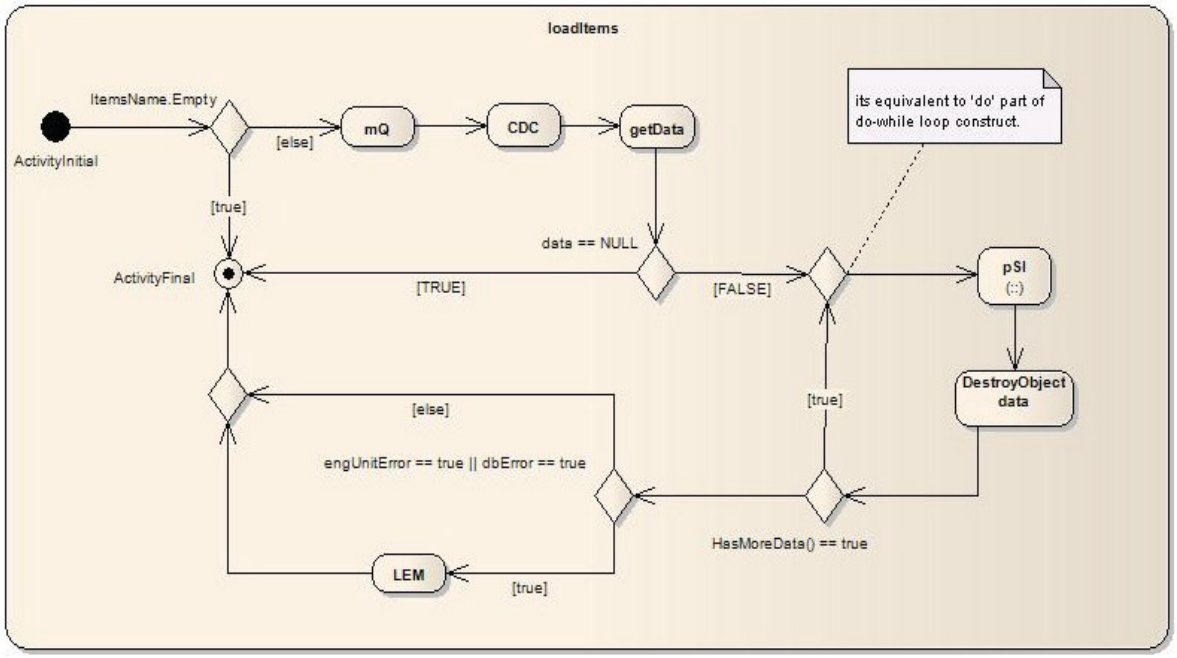


Figure 3-17: Load Items

3.2.5 Delete Trend Properties (DTP)

The Trending subsystem also has a provision to delete the existing trend templates. The user selects one or more trend templates and selects the “Delete Trend Template” functionality. The system verifies the user has sufficient privileges to delete trend templates. On successful authorization, the system first loads the details of selected trend template (Figure 3-19 and Figure 3-20) and then creates the query to load the associated trend data. The system prompts the user for confirmation. After getting confirmation from user, the system removes the trend data, plots and the selected trend templates. On successfully deleting the template the system refreshes the display. The system advises the user of the appropriate errors (e.g. database connection or query execution failure) if it fails to delete the template.

Figure 3-18 illustrate the control flow model of the Delete Trend Properties functionality of the Trending subsystem.

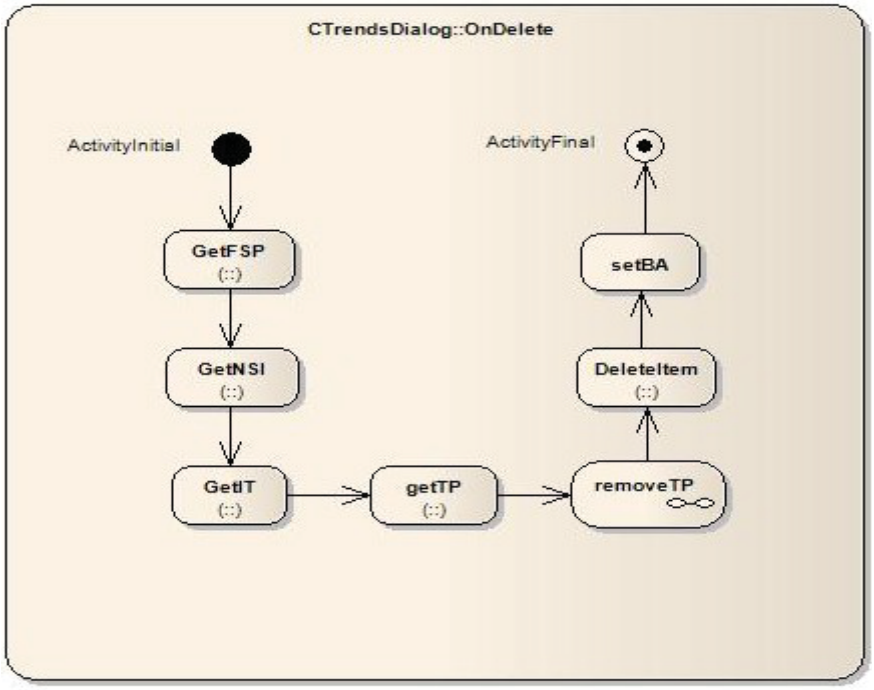


Figure 3-18: Delete Trend Properties

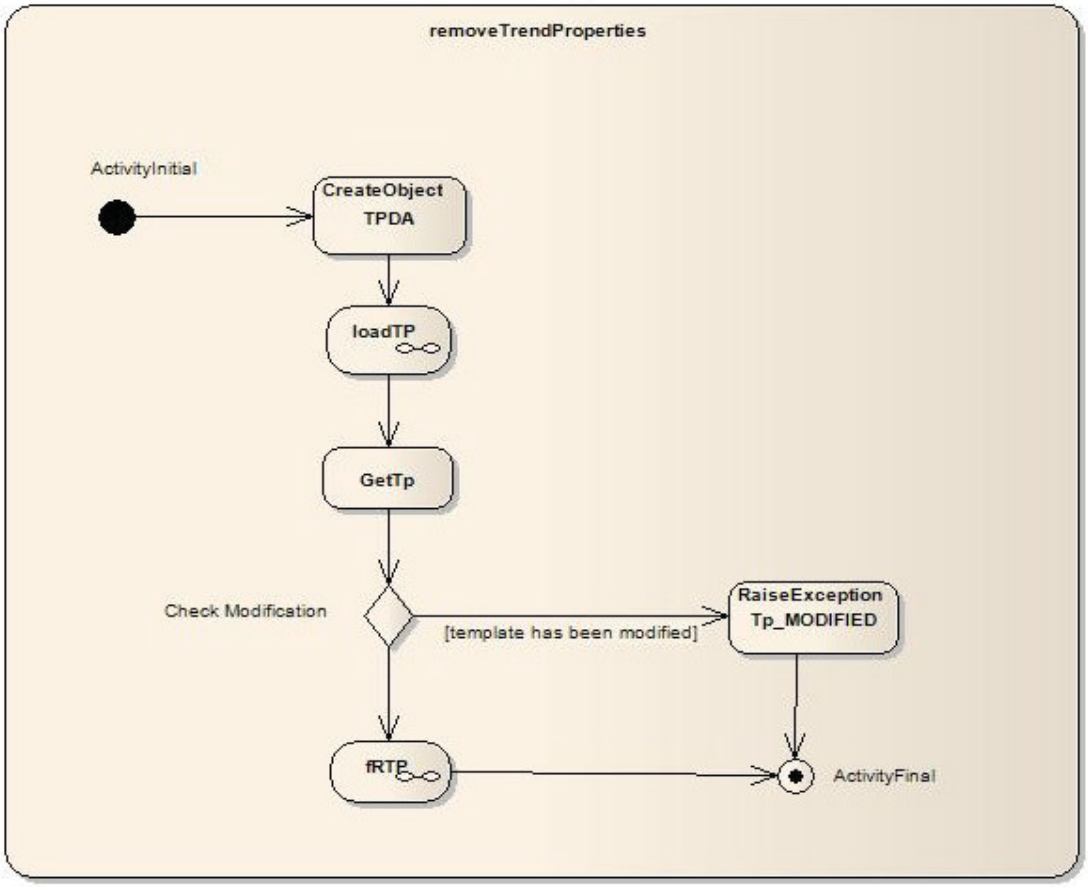


Figure 3-19: Remove Trend Properties

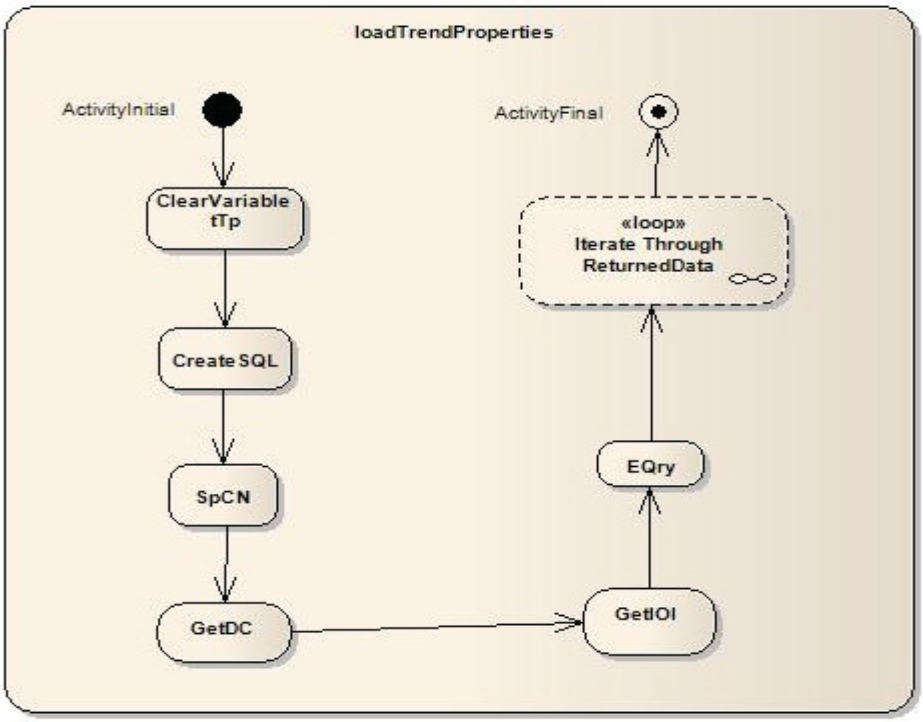


Figure 3-20: Load Trend Properties

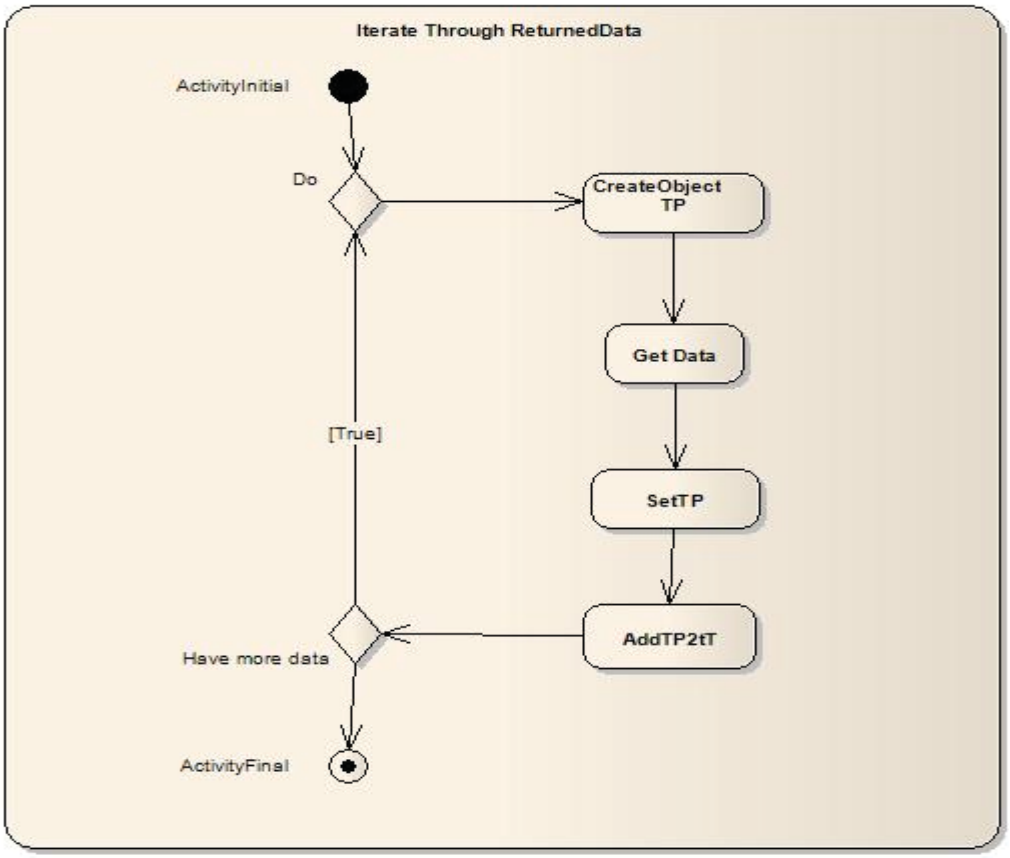


Figure 3-21: Iterate Through Data

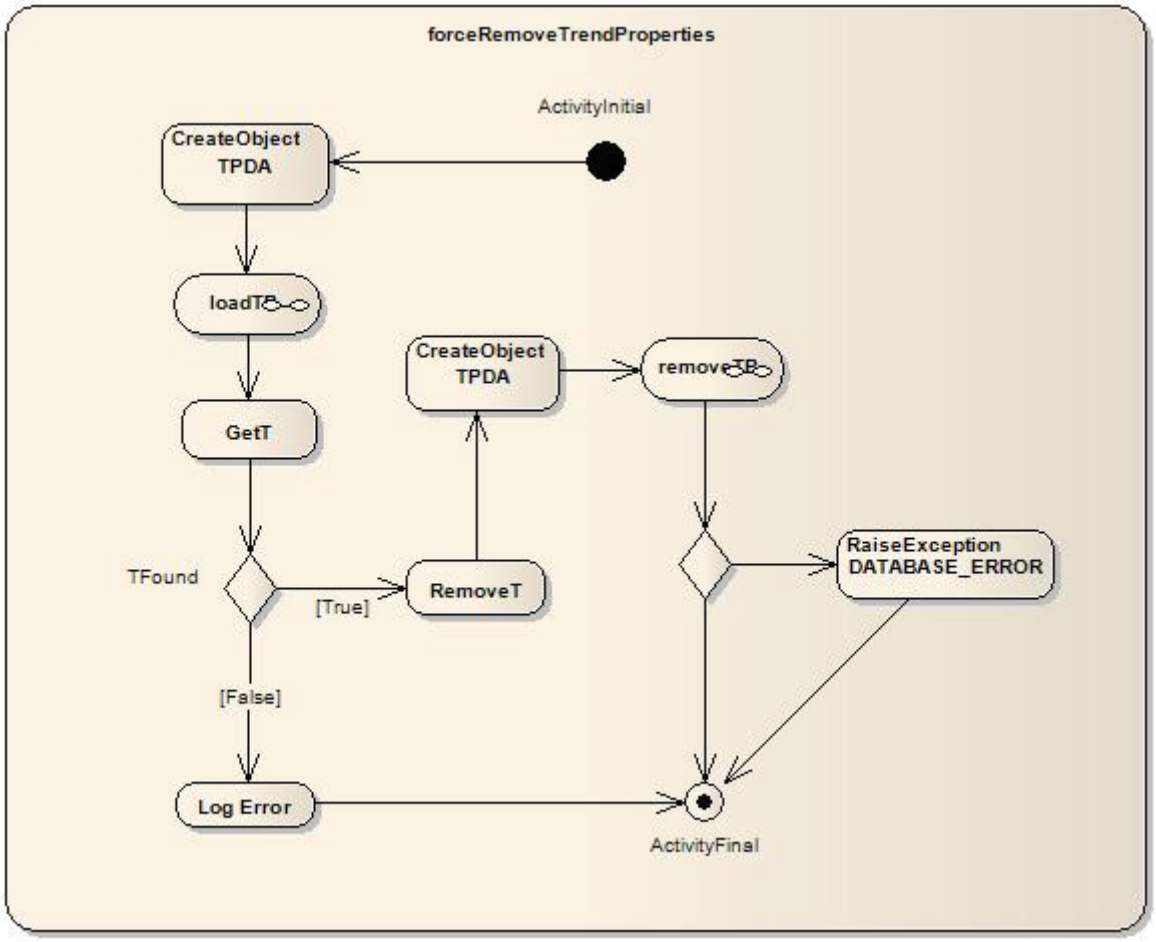


Figure 3-22: Force Remove Trend Properties

The summary of the key characteristics of these models is presented in Table 3-1. The column 'Nodes' represents the number of nodes in the model which is analogous to statements in a program. The 'Branches' column represents the number of edges immediately following predicate nodes. The complexity column is based on the cyclomatic complexity of the model.

Table 3-1: Summary of the key characteristics of case study models

Models	Nodes	Branches	Complexity
ECCS	23	17	11
ATM	135	28	16
ETP	77	26	14
DTP	52	37	21

3.3 Model Transformation

In typical model driven development (MDD), models are developed to depict the structural and behavioral aspects of a system under development from a particular viewpoint and at a particular level of abstraction. Developing a model in this way not only helps to focus on various specific aspects of the system, but also allow better handling of complexities of the system specification with the use of multiple models. Models refinement or decomposition into other models, enables the tackling of the complexity and enormity of system detail in layers. MDA is an emerging OMG standard for model-driven software development and it defines a layered framework for system specification. At the highest level of MDA is a computation independent model (CIM) where requirement models of a system are developed in the application domain at a conceptual level. In the next layer, the conceptual models are transformed into implementation technology independent models and are referred to as Platform Independent Model (PIM). At the lowest level they are referred to as Platform Specific Model (PSM). The platform independent models are transformed into models specific to a particular technology (e.g. J2EE and .Net). It is generally understood that for MDA realization, automated model transformation is a core mechanism as it is necessary for valid and consistent transformation, verification and synchronization between source and target models (from CIM to PIM and then PIM to multiple PSMs or vice versa).

Automatic transformation requires a set of production rules or mapping patterns between two modeling languages for translating a source model depicted in one language into another language. Transformation is the application of production rules on the source model until no more production rules are applicable and it finally yields the target model.

3.3.1 Transformation Types

Model transformation can be distinguished based on the way it transform the source model into the target model as shown in Table 3-2. Usually, models are initially developed or available at a particular level of abstraction and are then incrementally refined. Horizontal transformation involves transforming a source model into target model at the same level of abstraction e.g. transforming an AD model into a FSM model. On the other hand, vertical transformation involves transforming a model from one level of abstraction to another e.g. refining a design model into a development model or transforming a model from a CIM to a PIM level. As models are expressed in a particular modeling language, a transformation is called endogenous when the modeling language of both source and target models is the same and exogenous when both source and target models are expressed in different modeling languages.

Refactoring and simplification of a model are two examples of endogenous transformation while code generation and reverse engineering are examples of exogenous transformation.

Table 3-2: Orthogonal dimensions of model transformations (taken from [Mens et al. 2006])

	Horizontal	**Vertical**
Endogenous	Refactoring	Refinement
Exogenous	Migration (Porting)	Code Generation

3.3.2 Transformation Techniques

Various model transformation techniques have been developed to produce models in the same or different technological spaces. There are various classifications (e.g. imperative or declarative techniques, generic or dedicated transformation techniques) defined for transformation techniques [Mens and Grop 2006]. In the following section, the classification based on the underlying technology used in the transformation is presented. Further details on the classification of transformation techniques can be found in [Mens and Grop 2006].

3.3.2.1 Graphical Transformation Techniques

Graphical transformation techniques treat source and target models as graphs. *Graph Transformation* is a well studied graph-based model transformation approach and has been applied in variety of application domains such as pattern recognition, database specification, programming languages, distributed systems, optimization etc.

Graph Transformation is a formal mechanism used to generate, evaluate and manipulate graphs [Varro et al. 2002] and as the name suggests, provides visual transformation rules that when applied rewrite the graphs. A transformation rule consists of two graphs, a LHS graph to match and a RHS graph to replace (or add). Typically, these transformation rules play a pivotal role between the source and the target model and they provide a general and flexible mechanism for expressing a mathematical relation between them. In a source model, the LHS of a transformation rule can hold true for several artifacts and in that case the transformation rule executes for each instance iteratively. The transformation algorithm (application) executes the relevant transformation rule(s) for each artifact in the source (input) graph and adds the corresponding RHS artifact to the intermediate graph which finally yields the target (output) graph.

The Attributed Graph Grammar (AGG) [Ermel et al. 1997], PROgrammed GRaph REplacement System (PROGRES) [Schürr et al. 1997], Graph Rewriting and Transformation

Language (GReAT) [Agrawal et al. 2004] and VIsual Automated model TRAnsformations (VIATRA) [Csertán et al. 2002] are some of the well known examples of *Graph Transformation* systems. AGG is a general purpose algebraic approach based graph transformation tool. AGG support transformation within a single domain only. Although, AGG uses graph grammar for describing transformation rules, it does not support programmed graph transformations. PROGRES allows not only specifying the transformation rules but also defining the sequencing of these rules. Similar to the PROGRES, GReAT also supports describing both transformation rules and execution ordering rules. Both PROGRES and GReAT allow heterogeneous transformations.

Even though, graphical transformation techniques offer quite expressive mechanism or language for transformation rules, the execution control rules have limited usability in specifying complex transformation logic. Various visual control-constructs including sequencing, recursion and conditional branching are supported in this context but writing complicated transformation algorithms is very difficult.

3.3.2.2 Textual Transformation Techniques

Text based transformation techniques as the name suggests process models as textual data. Output models are produced by taking an input model from a text-based file or data stream and with the application of text-based pattern matching. For example, XSLT, ATL (ATLAS Transformation Language), YATL (Yet Another Transformation language) and AWK are some of the known textual transformation techniques.

XSLT is an industry standard for model interchange and it is specifically defined for describing transformations of XML-based models. It is a declarative technique that lacks execution sequencing rules. As a consequence, it requires experience and considerable effort to define even simple model transformations in XSLT. However, a solution 'MTrans' has been proposed to address the problem of describing the sequencing rules for XSLT-based transformations [Peltier et al. 2000].

XSLT is also a most widely used transformation technique for XML-based artifacts. Many UML tools support the facility to export and import models in XMI which is a XML-based standard for interchanging UML models. Once a model is exported or available in XML format, it is possible to use existing XML tools, such as XSLT to perform model transformations. Even though XSLT was proposed to describe transformations in general, it is tightly coupled to the XML.

Although textual or lexical languages are better in handling complex and detailed transformations and offer a finer control over transformation description, the construction of

model transformations using such languages is time consuming, costly and mostly defined semi-formally or informally. Another disadvantage of textual transformation approaches is that the transformations are performed in batch mode and offer little control over execution of transformation.

3.3.2.3 Hybrid Transformation Techniques

Hybrid techniques offer a pragmatic approach to address some of the shortcomings of graph-based transformation languages. These techniques mostly enhance graphical mapping with the accessibility and expressive power of programming languages (e.g. Java and OCL), for describing complicated transformation algorithms. For example, Sendall et al. [2002] proposed VMT (Visual Model Transformation) which is a visual declarative language similar to GReAT. However with the use of OCL to indicate the selection, creation, modification and removal of model elements, it provides greater control on the transformation. The main features of VMT are the abstraction with visual notations in the specification of transformation rules (which makes the comprehensive mapping of rules easier), and the strength of OCL.

Milicev [2002] proposed a graphical language for describing model transformations with the features of both imperative and declarative languages. It uses the UML object diagram as a notation for expressing the transformation rules. The notations are extended to support the constructs for conditional, repetitive, parameterized, and polymorphic model element creation. These concepts provide built-in features to specify and control model transformations. However, the approach needs a specifically built tool to interpret the extended graphical notations and run transformations. This is because the extended graphical notations make heavy use of stereotypes and common UML elements, such as, packages, in ways that are not typically seen in UML-based languages. Although, the approach has the advantage of strong expressive powers due to the extended notations, it is very likely that the selection conditions, in contrast to OCL, would become complex and hard to maintain in the case of complex selection criteria.

Rational XDE (eXtended Development Environment) is a specialized model transformation platform [IBM 2010]. The transformations rules are defined as model templates called patterns. A pattern can take and return parameters and contain arbitrary procedural code in Java, VB or C#. The transformation engine executes the pattern binding parameters with arguments automatically or manually, to yield the target model. Although, the proprietary XDE approach has a strong transformation engine, the main drawback is its limited capability to compose patterns.

3.4 AD to CPN Transformation

As mentioned earlier, AD2 (AD in UML2) has gone through major changes and acquired Petri-Nets like token flow semantics. Unlike former UML versions, where the semantic of AD1 (UML version 1.x) was based on State Machines (another UML behavioral modeling language). Since UML State Machines are synchronous, AD1 had certain limitations, such as, it could perform distributed computational steps only synchronously and only one step per concurrent region could be activated at a time. On the other hand, the execution of distributed steps in AD2 need not be synchronized. Thus AD is now an ideal tool for modeling asynchronous control flow logic required in systems like multithreading and agent-based systems.

The new token flow semantics of AD2 has made it executable and can simulate the control or object flows in the system models. In terms of token flow, the UML2 specification state that "by flow, we mean that the execution of one node affects, and is affected by, the execution of other nodes…" [OMG 2005 - page 308]. Other researchers also consider that the new AD semantics provides the basis for the execution of models [Bock and Gruninger 2005] as it enables the simulation of the behavior of the system. Ironically, the lack of simulation and analysis tools has restricted the potential advantages of using the new AD semantics. The semantics of UML generally and AD specifically is still defined informally in natural language, and is therefore largely prone to misuse and inconsistent implementations. Cook pointed out that despite the popularity of UML in the software development industry, the developed models in practice are often inconsistent with the standard defined semantic [cited in Henderson-Sellers 2005]. He ascribed a number of reasons for this problem including the lack of standard conformance procedures and a general practice among modelers to ignore the UML semantics. The imprecision and ambiguity of natural language make it difficult, not only for precise analysis but also for the detection and correction of subtle errors, incompleteness, and inconsistencies in models [Broy et al. 2007]. In their seminal work, France et al. [1998] identified the need to formalize the UML semantics and initiated the Precise UML (PUML) project. They argued that the semantics, defined with a mixture of meta-models and natural language, is though good for its abstract syntax but is not precise enough for the clear meaning of its constructs (notations). Clear semantics of a language is a prerequisite for the verification of consistency, formal analysis and execution of a model that it described. To address this issue, two types of approaches have been reported in literature; the first is defining the formal semantics for UML [Broy, Crane, Dingel, Hartman, Rumpe and Selic 2007; Evans and Kent 1999] also known as operational formalization [Heckel 2003], and the second is mapping the elements of a UML diagram into an existing formal language [Baresi and Pezzè 2001; Shen et al. 2001] a.k.a. denotational

formalization [Heckel 2003]. For AD, various formal languages such as Finite State Process [Rodrigues 2000], Abstract State Machine [Borger et al. 2000], Labelled Transition System [Eshuis and Wieringa 2001], π-calculus [Yang and Zhang 2003], Process Specification Language [Bock and Gruninger 2005] and Petri Nets [Gehrke et al. 1998; Störrle and Hausmann 2005] have been used to define its formal semantics.

Gehrke et al. [1998] used Petri Nets (PNs) as a target formalism for automatic derivation of Collaboration diagrams (CD) from AD. They describe a rule based transformation methodology for AD to Collaboration diagram but it was based on earlier version of AD. Rodrigues [2000] used Finite State Process (FSP) to define Transition System (LTS) based semantics for AD. Börger et al. [2000] proposed Abstract State Machine (ASM) based semantics for AD. Eshuis and Wieringa [2001] used hypergraphs to define LTS-based formal semantic for AD. Although, they also developed a real time execution algorithm for it, the proposed syntax and semantics were tied to intended use for workflow modeling and analysis. Yang and Zhang [2003] used π-calculus for formalizing AD. Although, the proposed formalism is intended for workflow modelling, the syntax and semantic of π-calculus is very difficult. Lopez-Grao, Merseguer and Campos [2004] introduced formal semantics for AD by transforming it into Stochastic Petri Nets. Merseguer [2003] described the three stage transformation of an Activity diagram into Stochastic Petri Nets. Accordingly, the early stage involves model simplification operations such as merging control nodes and removing bad design constructs. The second and last stage includes replacement and removal of pseudo artifacts from the final PN model respectively. Ironically, in some cases the proposed model simplification is not helpful, as it not only adds more ambiguities (shown later in Figure 3-29), but also makes the transformation erroneous (e.g. in case of three successive merge or decision, fork or join). Störrle [Störrle 2004; 2004; Störrle and Hausmann 2005; Störrle and Hausmann 2005] proposed formal semantics for AD2 over a series of papers and pointed out various deficiencies in the semantics of AD2. While addressing the artifacts relating to control and data flow respectively, he described the PN semantics for a number of overlapping AD artifacts using several Petri nets dialects such as Place-Transition Nets (P/T Nets), Procedural Petri Nets (PPN) and CPN. Like others, Störrle suggested the simplification of the model by merging the multiple levels of control nodes which adds ambiguities into the models. Bock and Gruninger [2005] proposed formalization of AD through Process Specification Language (PSL). Table 3-3 summarizes the key characteristics of the existing works.

Most of these studies focused on defining and formalizing AD, but did not address the pragmatic model transformation issues (i.e. undefined link and execution semantic for invoked

activities, label and predicate inscription, ambiguous specification and hierarchal activities). The volume and complexity of real world system models make manual transformation extremely difficult, expensive and unpredictable. Moreover, most of these studies were based on previous versions of AD and due to a major revision in UML2 these studies are now outdated except for those of [Bock and Gruninger 2005] and [Störrle and Hausmann 2005] . The following section described the approach where a methodology to automatically transform an AD model into another formal model was devised to address the gap between the formal semantics defined in [Störrle 2004; Störrle and Hausmann 2005] and the practical model transformation. The motivation for translating an AD into a CPN stems from the fact that CPN is a mathematically defined modeling language. As such it provides unambiguous, visual and executable specification which is backed by several formal verification and validation techniques and tools. Moreover, CPNs are ideally suited to AD models as a target formal language because of the similarity between AD2 and CPN semantics [OMG 2007 - page 326].

Table 3-3: Comparison of existing work to the define formal semantic for AD

	AD Version	Formalism	F.T.	Asynch. Invocation	Hierarchy	Domain
Merseguer [2003]	1.x	LGSPN	D	Yes	Yes	Performance analysis
Börger et al. [2000]	1.x	ASM	O	Yes	Yes	Program
Ehusis [2001], Rodrigues [2000]	1.x	LTS	D	No	Yes	Workflow
Gehrke et al. [1998]	1.x	PN	D	No	No	Program
Yang et al. [2003]	1.x	π-calculus	O	Yes	Yes	Workflow
Störrle [2005]	2.0	PPN, CPN	D	No	Yes	Program
Bock et al. [2005]	2.0	PSL	O	Yes	Yes	Process

- Denotational (D), Operational (O), Formalization Technique (F.T.)

3.4.1 Colored Petri Nets

PN is a formal modeling technique for consistent and clear description of the behavioral aspects of a system. It has a well-defined syntax and semantics which supports the explicit

description of both states and actions. The states are represented by places and each place is assigned a number of tokens (also known as marking). The transitions depict the behavior and arcs provide the links between places and transitions. The execution of a behavior is depicted by a transition firing that changes the state of the system. An inward arc shows that the token from the corresponding input-place can be removed and the outward arc implies that the token can be added to the next place. A transition gets enabled only when all the preconditions for the actions have been satisfied such that there are enough tokens available in the input places. A transition can only be fired when it is enabled. The exact number of tokens to be removed/ added is determined by the arc expression [Jensen 1993].

CPN is a popular variant of PN that exploits the synergy of PN and high level programming constructs. In CPN, the colored tokens were introduced to improve the process modeling capabilities of PN [Jensen 1992; Jensen 1993; Jensen 1996] with data definition and the processing of programming languages. With the synthesis of PN and programming language, CPN enables the modeling and verification of very large scale systems in PN formal semantics. It is widely used in academia and industry due to its intuitive visual notation and extensive tool support for design, simulation and system verification. $CPN\ Tools$ is a CASE Tool for editing, simulating and analyzing CPN models. It has a built-in syntax checker which validates the model before doing simulation or formal analysis. It features a simulator for handling both timed and untimed models. For model data declaration and model inscription it uses the *CPN ML* language which is a adaption of the Standard ML [CPN-Group 2010].

Despite the fact that AD has PN-like token game semantics, the transformation from AD to CPN is not straightforward owing to several syntactical and semantical differences. In this context, there are a few important questions that need to be addressed. Each of the following subsections considers a particular issue, and describes a number of objective criteria to be taken into consideration in developing a transformation methodology.

3.4.1.1 Syntactic Variation

Since precise syntax is critical to correct language interpretation, visual notations (e.g. lines and boxes) and their composition (e.g. connection and partitioning) are critical to visual modeling languages. CPN has only two types of nodes and connections (arcs) between the nodes is restricted (source and target nodes must be of different types). However, AD2 has a complex hierarchy of nodes and connections between these nodes are somewhat flexible. In terms of visual notations, AD2 is provided with nine notions for ten control flow related nodes types [OMG 2007, page 415-419], up to the intermediate level of activities. To elaborate the

differences further, the meta-model descriptions of AD2 and CPN are shown in Figure 3-23 and Figure 3-24. The analysis is limited to modeling elements of Activities up to the *Intermediate* package as it inherently supports modeling similar to PN [OMG 2007, page 297].

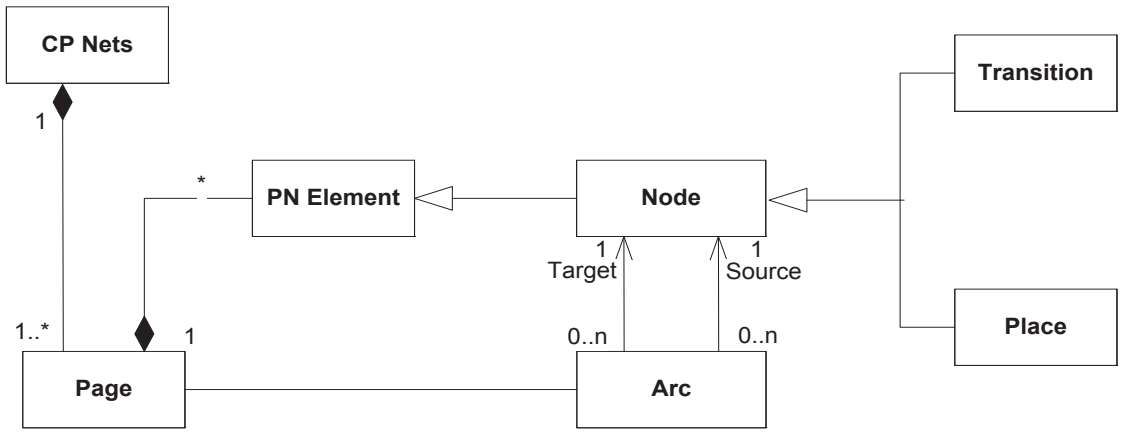


Figure 3-23: CPN meta-model adapted from [Hillah et al. 2006]

3.4.1.2 Semantic Variation:

Interestingly, the syntax of AD in UML2 is formally defined using class diagram (generally referred to as meta-model) but its semantics is still in informal natural language. A summary of the new token game semantics of AD2 up to the Intermediate package is presented in Table 3-4. It is important to note that the aggregated semantics of control nodes types in AD is almost similar (but not equivalent) to the semantics of the two types of CPN nodes. Furthermore, in some cases, AD nodes seem to be a specialized version of CPN nodes (e.g. Decision & Merge nodes in AD2 and Place node in CPN). Therefore, they can be mapped directly to the places and transitions in CPN.

In other cases, such as, Activity-Final, translation can be performed by adding some explicit contextual conditions or extra code to the CPN node to get the required AD behavior. The difference in both cases is that the first type of translation is implicit translation as the semantics of AD nodes is inherently supported in CPN and in the second case, it is an explicit translation as the semantics of the Activity-Final node is achieved by overloading the inherently behavior. A node-level comparison between the token game semantics of both AD2 and CPN was undertaken as part of this study and is summarized in Table 3-4.

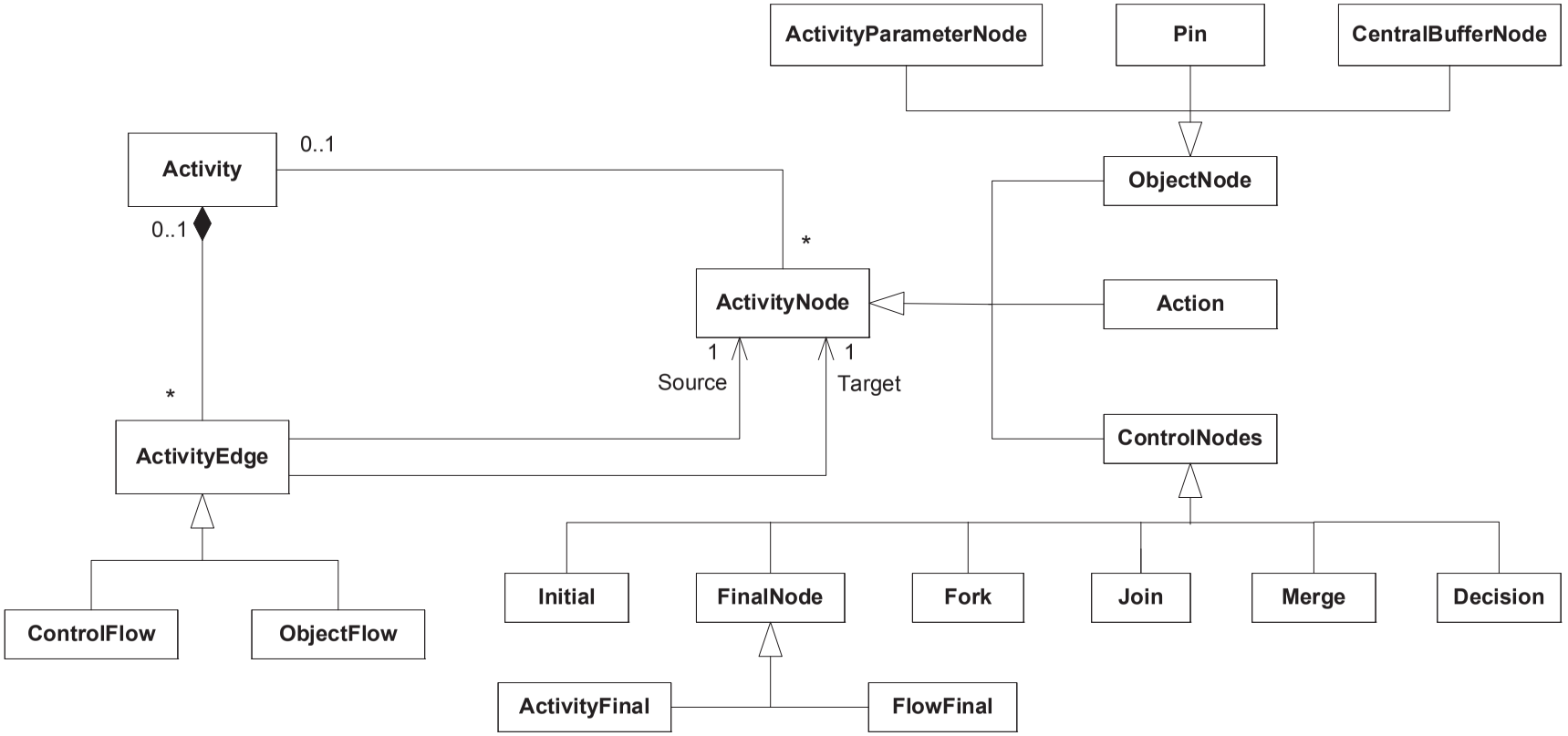


Figure 3-24: AD meta-model (Adapted from [OMG 2005])

Table 3-4: A Summary of token game semantic for AD2

AD2 Node	Description	Semantic
Action	An atomic executable element representing a single step within an activity	▪ An action can only start execution when all the incoming edges and pins have tokens. ▪ When an action starts execution it consumes all the tokens available on the incoming edges and pins. ▪ On completion of the action execution, tokens are offered on all outgoing edges and pins.
Initial	A node where the flow starts in an activity	▪ A control token is placed on it whenever the enclosing activity is invoked. ▪ An initial token is offered to all outgoing edges but only one of them will receive the token.
Activity Final	To stop all flows in the activity	▪ When a token arrive at this node, the enclosing activity will be terminated; particularly, all executing actions are stopped, all other tokens are destroyed and all flows are terminated, except ○ Token in activity output parameter node ○ Asynchronously invoked actions
Flow Final	To terminate a flow – indicates the termination of a flow path	▪ All tokens arriving on it are destroyed.
Fork	To split the inflow into multiple concurrent outflows	▪ Incoming tokens are duplicated to all outputs
Join	To synchronize multiple inflows into single outflow	▪ All incoming tokens are joined according to the following strategy and offered to the outflow: ○ If all incoming tokens are control token then only one control token will be offered ○ If the incoming tokens are mixture of control and data tokens then only the data tokens will be offered.
Decision	To split an inflow into alternate outflows	▪ Each incoming token can traverse only one outflow. ▪ No token duplication is allowed so although a token will be offered to the all outflows but only one outflow will accept it.
Merge	To combine multiple inflows into single outflow	▪ All incoming tokens are forwarded to the single outflow without synchronizing the inflows and joining the tokens.

Table 3-5: Mapping between AD and CPN nodes with token game semantic of CPN

AD2 Node	CPN Node	CPN semantic
Action	Transition	▪ A transition can only start execution when all the input arcs have tokens. ▪ When a transition starts execution it consumes tokens from all inputs. ▪ On completion of execution, transition offers tokens on all outputs.
Initial	Place	▪ An available token is offered to all outgoing arcs but only one of them will receive the token. ▪ A token can stay on a place until it is consumed by a transition.
Activity Final	Place	▪ All token arriving on it are destroyed. A token in a place with no outgoing arc have no effect on the model state and represents the end of flow behavior.
Flow Final	Place	▪ All token arriving on it are destroyed. A token in a place with no outgoing arc have no effect on the model state and represents the end of flow behavior.
Fork	Transition	▪ Incoming tokens are duplicated to all outputs.
Join	Transition	▪ All incoming tokens are joined according to the specific strategy and offered to the outflow.
Decision	Place	▪ Each incoming token can traverse only one outflow. ▪ No token duplication is allowed so although a token will be offered to the all outflows but only one outflow will accept it.
Merge	Place	▪ All incoming tokens are handled separately and forwarded to a outflow without synchronizing the inflows and joining them.

3.4.1.3 Modular Modeling

Modular modeling is quite useful for large complex systems. The modular creation of models makes the development and maintenance of these models easier. Moreover it supports the reuse of model components as well. The provision of model libraries in UML 2.0 enables the development and reuse of standard or user-defined models [OMG 2007]. The high level models in AD2 specify the processes at the abstract level. The precise description of each task involved is relegated to the lower level detailed activities described as behavior which in turn ultimately resolves to the individual action or sub-activities.

In AD2, two mechanisms are provided for modular decomposition of complex logic through multiple activities or sub-activities. Firstly, the specific type of actions i.e. Call-Behavior-Action and Call-Operation-Action are provided for reuse of other already defined activities (sub) and to invoke them during the execution of invoking activity (super). The usage of Call-Behavior-Action is comparable to a calling function in programming languages and requires having a linking mechanism for combining the invoking activity with various other activities it uses. The

Call-Behavior-Action is a direct invocation of called activities whereas the Call-Operation-Action is an indirect invocation of called activities. Secondly, a structured activity is provided for decomposing the behavior in one or more subordinate activities. Although, an activity can invoke (call) other activities (through the call-behavior-action) at run-time and can create call-hierarchies, this is different from Structured Activity. With a structured activity, a sub-activity (subordinate) commences. The super-activity start executing and super-activity remains active until all sub-activities have completed their execution.

For modular modeling, PN supports the abstraction (refinement) mechanism. Formally, the abstraction is explained with the concept of bordered sets where the nodes (transitions or places) which are directly connected within a border or locality are called Transition or Place Border Set. Therefore, in a net, a Transition Bordered Set can be abstracted by representing the transition bordered set with a single transition. On the other hand, the refinement of a transition will yield a net structure. Similarly a Place Bordered set can be abstracted by replacing it with a single place which on refinement results in a net structure.

In CPN, the modular model structure is defined through the pages mechanism [Jensen 1992] where the whole system model is comprised of several corresponding page modules. With the help of the module linking mechanism, the page (also called super-page) with references to other pages are linked to the referred pages (also named as sub-pages). A page can be both referring (super) and referred (sub) page simultaneously, hence it can be reused many times in the model.

In order to describe a task precisely which is represented by a transition node at an abstract level, the detailed description needs to be relegated to the sub-page. The integration of these detailed specifications (sub-pages) of particular transition nodes are accomplished by a corresponding transition substitution marked by a sub-page name-tag. After attaching a page module to a substitution transition, a couple of places on the sub-page need to be constituted as port places for communication with the environment. The place which is set to receive tokens from the surroundings is called the input port and marked with an in-tag, whereas the place for sending tokens to the surroundings is said to be the output port and marked with an out-tag. A place which can mimic both input and output ports simultaneously is tagged as I/O. At the super-page, the input and output places of the substitution transition are called socket places and are linked to the corresponding input/output ports on the sub page.

According to the UML specification [OMG 2007], the direct or indirect invocation of a behavior or an activity can either be synchronous or asynchronous. In the case of synchronous invocation, the execution of the invoking action does not complete until the invoked behavior or

activity is completed and returned an output (token). On the other hand, with asynchronous invocation the invoking action will complete as soon as the invoked behavior commences. In an asynchronous mode, the invoking action will copy the input token to all the outputs and the flow continues while the invoked behavior will not return any output on its completion. The substitution-transition mechanism provides the synchronous mode of communication and is similar to synchronous invocation. The implementation of asynchronous invocation is not intuitive and straightforward in CPN and therefore requires further investigation.

In CPN, fusion of places or transitions, termed as place fusion and/or transition fusion are used for composition of Nets (modular modeling) [CPN-Group 2010]. The example in Figure 3-25 is taken from [Khadka 2007]. The first three diagrams on the left side depict various scenarios of place fusion whereas the first three nets shown on the right side depict various scenarios of transition fusion. The last diagram on each side is the final resulting net for any of the three cases.

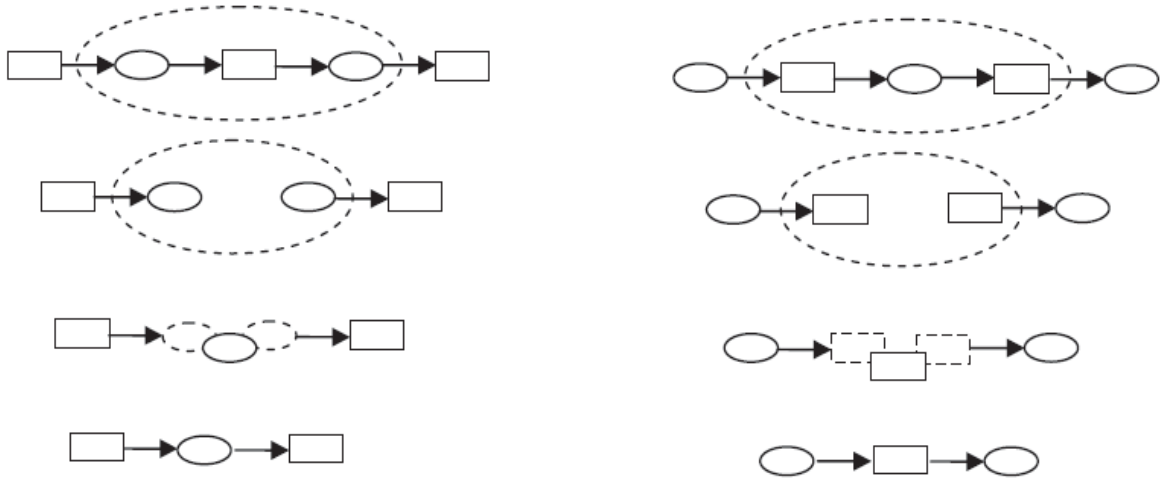


Figure 3-25: Scenarios of Place and Transition Fusion (taken from [Khadka 2007])

3.4.2 Transformation Methodology

The transformation methodology developed in this study comprises of three steps: pre-translation, translation and post-translation as shown in Figure 3-26. The first stage is the refinement of the model and is called pre-translation, as it is performed on an AD before transforming it into a CPN. In consideration of the informal semantics of AD and the different levels of abstraction, it is important to refine the source model by identifying and resolving the ambiguities in the model before transforming it into a CPN model. The transformation of AD models into CPN models is defined by rules and patterns based manipulation of models.

Therefore, the second stage is translation of an AD into a CPN via the execution of transformation rules. Given the fact that CPN is a high level PN and needs auxiliary data for displaying and simulating the model, enhancement to the transformed model is performed. In the third stage, the post-translation step, where the auxiliary data associated with the CPN is added to the transformed model. For illustration of the methodology, a simple example of an AD as shown in Figure 3-27 and the ECCS case study model (Figure 3-5) is used.

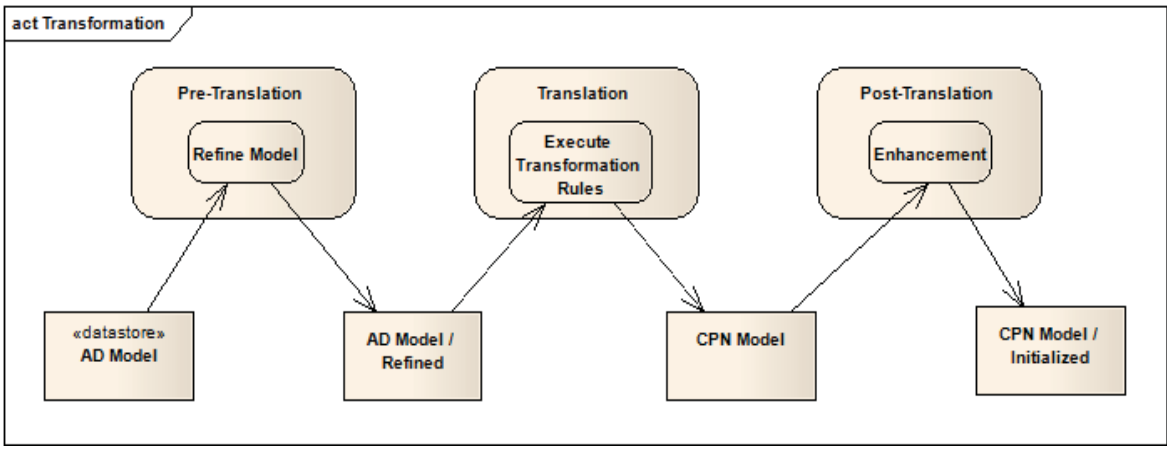


Figure 3-26: AD to CPN Transformation Methodology

3.4.2.1 Pre-translation step

Abstraction is a key approach to deal with the complexity in modeling [Booch 2010]. Typically, in the beginning a high-level model is developed with a simplified view of the system through the use of abstraction. However, an unplanned abstraction can leave ambiguity in the developed model. In model-based development, the usage and effectiveness of a model largely depends on the accuracy and level of information available in it. A model at a higher level of abstraction may provide a simplified view of a very complex system but it may omit vital and useful details resulting in ambiguities in the specification. On the other hand, a low level model with concrete information not only limits the usage of the specified model, but also increases the cost and effort needed to develop and maintain it.

In order to develop complete and consistent models, it is important to understand the application of abstraction. Pretschner and Jan [2005] noted the omission and encapsulation of details as basic techniques for abstraction and simplification in modeling. However, unplanned omission is problematic as Bock and Gruninger [2005] argued that *abstraction* is a deliberate and clearly identified omission of unnecessary and redundant information, whereas *ambiguity* is an

unintended and unidentified omission of useful information. In the case of unavailability of required information or when the intended system is not fully defined or understood, the modeler may need to make assumptions. Therefore, Prenninger and Pretschner [2005] pointed out that the ambiguities may arise from erroneous or partial behavioral assumptions. Bock and Gruninger [2005] also suggest that imprecise and implicit functional assumptions are the main cause of miscommunications and ambiguities.

Although the rich and layered syntax of UML languages empowers flexibility in expressing very complex systems at various levels of abstraction, it often causes error and ambiguities in the specification depicted in it. According to Heckel [2003], the obscure semantics of a modeling language is another source of ambiguities in a model. In a recent study, Lange and Chaudron [2005] noted that UML models in practice often contain several syntactical and semantical defects when used. They further stated that some of these issues stem from the inability or failure of the modeler to correctly and precisely specify the system in the model. Imprecise and implicit functional assumptions in a model can lead to missing critical aspects of a system and implementing a behavior which has undesired characteristics. Frisch et al. [2002] ascertain that the assumptions made while developing models at certain levels of abstraction are responsible for several pitfalls in models and they suggested refinements of the model for addressing these problems.

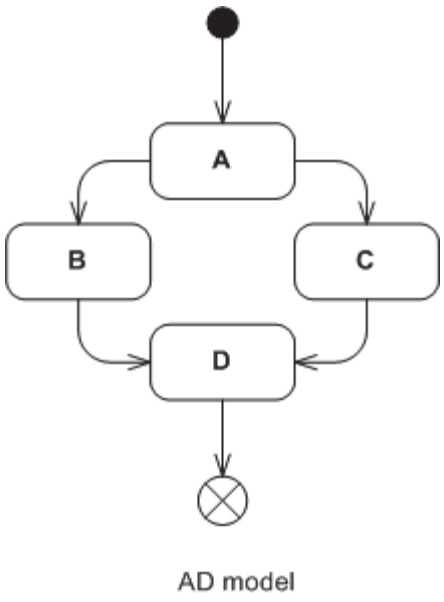


Figure 3-27: Example Activity Diagram

As one of the objectives of this study is the automatic generation of test cases from AD, the implicit behavior to generate a complete test suite needs to be expanded. Binder [1999]

argued that “implicit behavior is no less a requirement than explicit behavior”. Test cases generated for explicit behavior will not necessarily exercise the implied behavior. Therefore, in order to generate an adequate test suite it would be necessary that an implicit behavior be expanded and replaced with an explicit behavior.

In order to get an unambiguous and semantically valid model, it is necessary to refine the model by transforming it to the next lower level of abstraction. The fundamental steps of refinement are the identification of implicit (control-flow) behavioral assumptions in the model and replacing them with explicit control nodes, thus it is not viable until the purpose and usage of the implicit behavior is well-understood.

During software model development, a modeler may need information about the various aspects of the system such as control/data flow, functionality and operational environment depending on the projected viewpoint of the system. In the case of unavailability of required information, the modeler needs to make various types of assumption such as the data related assumptions (e.g. type, variable or value), functional assumptions (e.g. parameters, functional logic, procedure calls and error handling) and flow related assumptions (i.e. sequencing, loop and concurrency) [Dye 2002]. Analogous to the assumption types, the ambiguities in the model can be categorized according to these assumptions such as functional ambiguities, flow ambiguities and so forth.

Here in this chapter, only flow related ambiguities that may stem from the implicit flow related assumptions in the model are addressed. For instance, in the ECCS model (Figure 3-5), the action ‘*logon*’ has three inward edges which imply that it needs exactly three tokens to begin its execution. However, a further analysis could reveal that in any scenario it could not receive more than one token at any one time and any attempt to execute (simulate) this model would end in a deadlock eventually. Similarly, there are three other actions, such as *Register*, *Select* and *Put* that also have ambiguous specifications. As with reference to UML2 specification for action semantics [OMG 2005, page-301], “an action can only begin execution when all incoming control edges have tokens, and all input pins have tokens.” Hence it is reasonable to infer that the model (Figure 3-5) is designed with various assumptions for sequential control flow that are inconsistent with the semantics defined in the UML2 specification.

In AD2, several Activity artifacts with layered syntax and semantics, allow the depiction of a system model with ambiguous behaviors. The ambiguity in the behaviors of syntactically valid AD2 constructs can be seen in the examples in Figure 3-28. The diagram (A) in Figure 3-28 does not clearly specify whether activity-B (activity notation labelled B) and activity-C (activity notation labelled C) are concurrent or branch activities. Similarly, the diagram (D) in Figure 3-28

implicitly assumes that the execution of activity-A, follows the execution of activity-B and activity-C, but fails to specify whether the execution of activity-A will wait for the tokens from both preceding activities, or start execution soon after receiving a token from either one of them. The two possible but conflicting behavioral interpretations of diagram (A) are shown in diagrams (B) and (C), and highlight the ambiguity in the specification. Therefore, it is necessary to unmask the necessary control flow detail and replace the implicit control behavior with the explicit control artifacts. For instance, replace each implicit initial/ flow-final/ Activity-final node with an explicit initial/ flow-final/ Activity-final node, replace each implicit decision/ merge with an explicit decision/ merge and replace each implicit fork/ join with an explicit fork/ join.

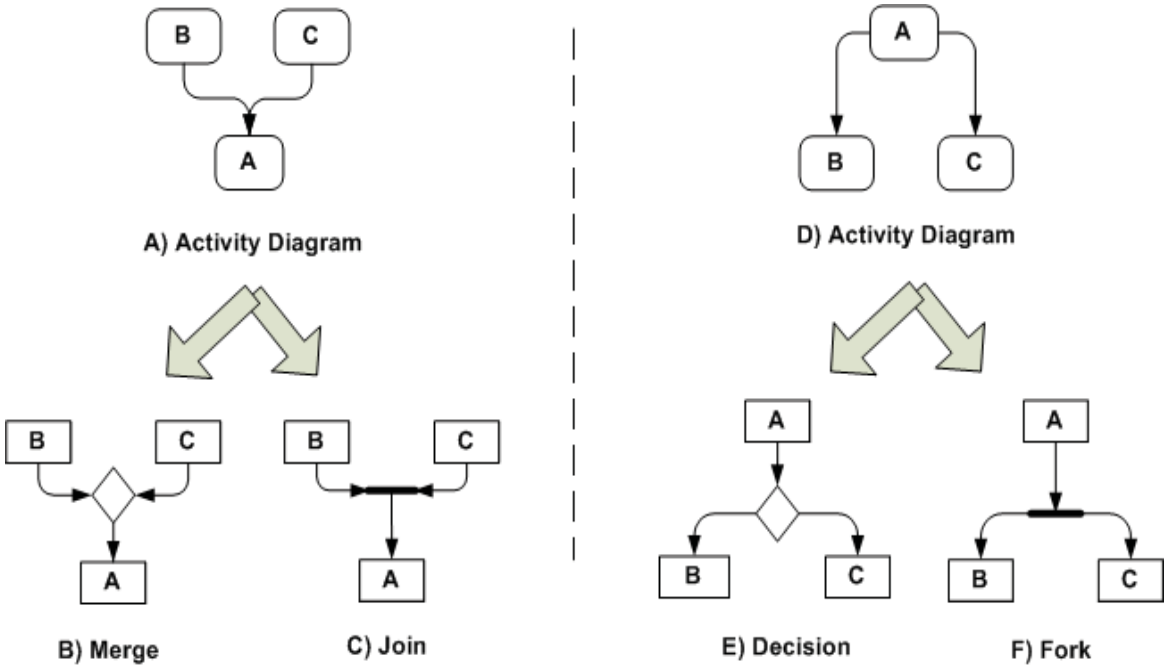


Figure 3-28: Ambiguous control flow behavior

Although AD2, allows the combination of decision and merge, or fork and join functionality in one node symbol to get a more concise and simplified model, as shown in Figure 3-29 (a, b, e and f), it must be done with careful consideration. As Störrle [2004] pointed out, the simplification of a control node in some cases may result into some unwanted behavior, such as the simplification of Figure 3-29 (c) into Figure 3-29 (d) does not produce the same behavior. Although this proposition does not seem to hold true in Figure 3-29 (g and h), in fact it also implies that the overuse of control nodes does not elucidate the process logic. Nevertheless, another similar but pragmatic scenario in Figure 3-29 (i) reveals that a straightforward simplification will not work here as both Figure 3-29 (i) and Figure 3-29 (j) mimic different behavior.

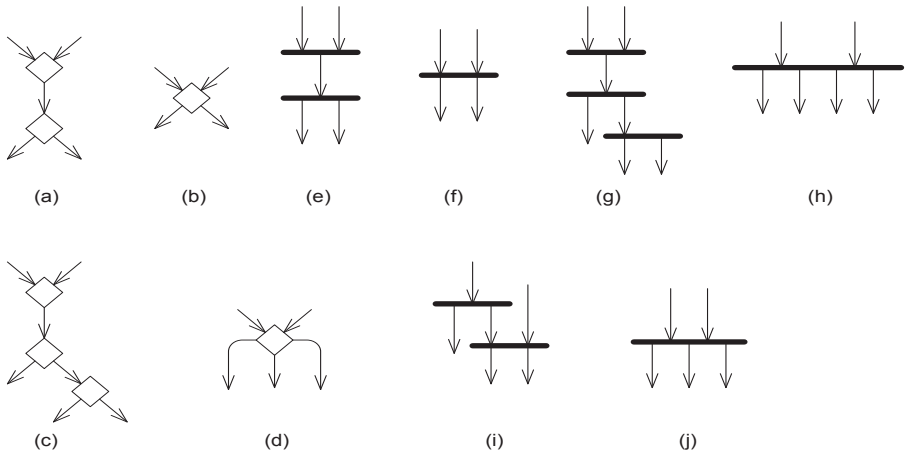


Figure 3-29: Node simplification

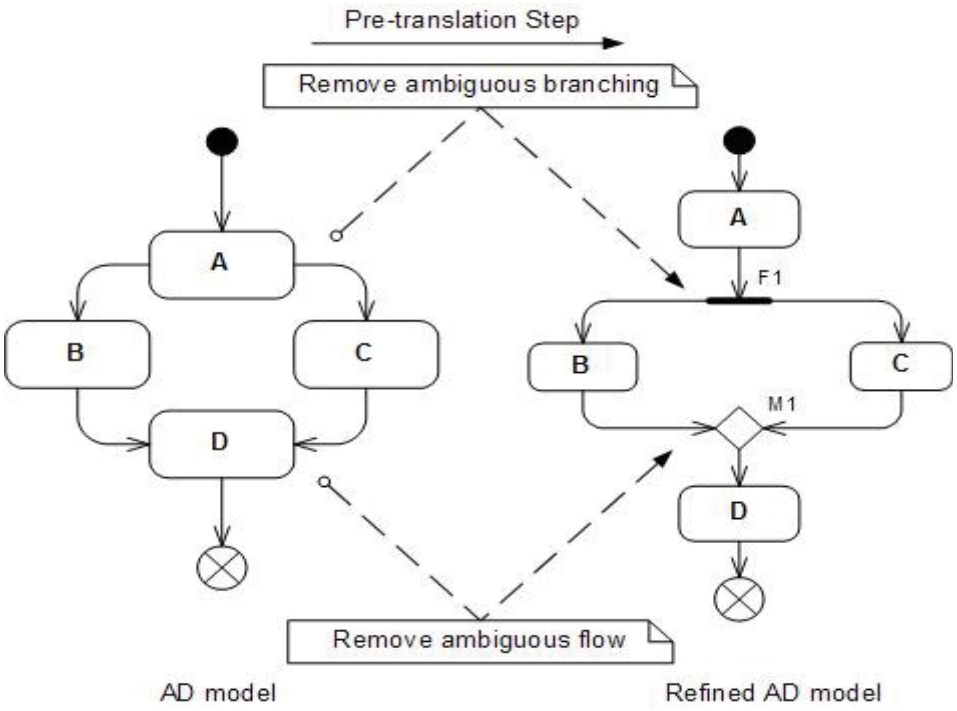


Figure 3-30: Illustration of refinement strategy

As a pre-translation step, a two pronged decomposition strategy of integrated control elements for addressing two sources of ambiguities in the control flow is proposed. As shown earlier, the implicit control flow behavior at an action node and simplification of control nodes can lead to a more precise and accurate behavior. Therefore, the strategy is to unmask the implicit control flow behavior from the action (executable) nodes and then combine the multiple levels of decision-decision, decision-merge, fork-fork and fork-join nodes provided that they are directly linked and no other node is linked between them. For illustration of the refinement step, see Figure 3-30, where a fork and a merge node is added to indicate that both activities A and B will execute concurrently but will not be synchronized.

The case study ECCS model (Figure 3-5) seems syntactically correct according to the UML2 specification but further analysis reveals that it is in fact semantically inconsistent. For detailed analysis, every node in the model is evaluated according to the UML 2.x specification which is given in Table 3-6. Based on the observations as given in column 2, 3 and 4 in Table 3-6, the following nodes are disambiguating by explicitly expressing the implicit behavior:

Action *logon* has three inputs, according the UML2 specification, the *logon* action finally enters into deadlock as it will wait for input from D1, D2 and D4. Therefore, it is essential to replace the implicit merge with explicit merge to make it executable.

Table 3-6: Ambiguous behavior

AD Nodes	Type	ID,OD	UML2 Behavior	Assumed Behavior	Refinement Action
Init	Initial	0,1	Implicit Decision	Nil	
Init	Action	1,1	Implicit Fork/Join	Nil	
D1	Decision	1,2	Implicit Merge	Nil	
Logon	Action	3,1	Implicit Fork/Join	Implicit Merge	Explicit Merge
D3	Decision	1,2	Implicit Merge	Nil	
Register	Action	3,1	Implicit Fork/Join	Implicit Merge	Explicit Merge
D2	Decision	1,2	Implicit Merge	Nil	
Authenticate	Action	1,1	Implicit Fork/Join	Nil	
D4	Decision	1,2	Implicit Merge	Nil	
F1	Fork	1,1	Implicit Join	Nil	
Verify	Action	1,1	Implicit Fork/Join	Nil	
F2	Fork	1,1	Implicit Join	Nil	
F3	Fork	1,1	Implicit Join	Nil	
Select	Action	4,1	Implicit Fork/Join	Implicit Merge	Explicit Merge
D5	Decision	1,4	Implicit Merge	Nil	
Put	Action	2,1	Implicit Fork/Join	Implicit Merge	Explicit Merge
D6	Decision	1,2	Implicit Merge	Nil	
Configure	Action	1,1	Implicit Fork/Join	Nil	
D7	Decision	1,3	Implicit Merge	Nil	
M	Merge	3,1	Implicit Decision	Nil	
Order	Action	1,1	Implicit Fork/Join	Nil	
F4	Fork	1,1	Implicit Join	Nil	
Final	Flow final	1,0	Implicit Merge	Nil	

ID: Incoming degree of a node,

OD: Outgoing degree of a node.

- Action *Register* has three inputs. Even if it receives a token from D1, it will essentially wait for input from D2 and D3 as well resulting in an eventual deadlock. Thus it is essential to replace the implicit merge with an explicit merge to make it executable.
- Action *Select* has four inputs, where the control flow will stall because the *select* action finally enters into deadlock as it will wait for input from F1, D5, D6 and D7. Replacing the implicit merge with an explicit merge will resolve this problem.
- Action *Put* enters into deadlock as soon as it receives an input from D5 but cannot proceed until it receives an input from D7. Therefore, it is essential to replace the implicit merge with an explicit merge to make it executable.

The refined ECCS model is shown in Figure 3-31.

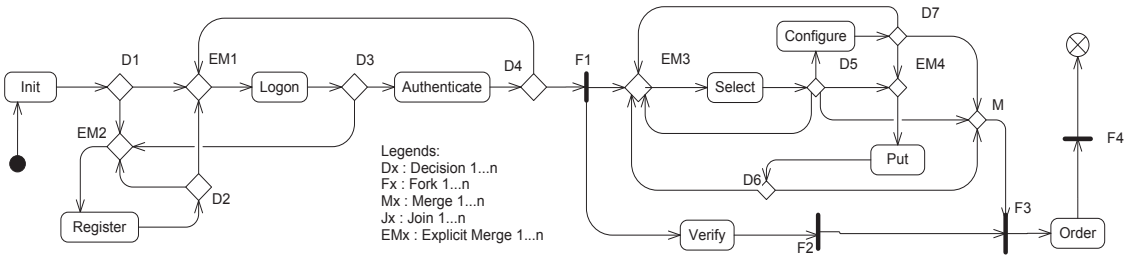


Figure 3-31: ECCS Model (after refinement)

3.4.2.2 Translation

In model transformation, production (transformation) rules play a pivotal role. The production rules provide a general and flexible mechanism for expressing a formal relation between source and target formalism. Formally, a production rule is an ordered pair $P = (A, C)$ where both A and C are a finite set of artifacts in source and target models respectively. The Left Hand Side (LHS) of a production rule represents the artifact(s) in a source model where as the Right Hand Side (RHS) is used for artifacts in the target model. In a source model, the LHS of a production rule could hold true for several artifacts, in which case the production rule will be executed iteratively for each instance. The transformation algorithm executes the relevant production rule for each artifact in the source model and replaces it with the corresponding RHS artifact for the target model.

3.4.2.2.1 Production rules

For AD to CPN model transformation, the production rules are defined for three types of AD artifacts, action nodes, control nodes and activity edges corresponding to *Intermediate*

Activities (IA). The rules illustrate the conditions and the transformation of AD elements into CPN artifacts. In AD, *Action* is a basic atomic artifact that becomes ready to execute when token(s) are available on all inputs and all preconditions are satisfied. It consumes all input tokens when it starts execution, holds them until completion and then offers these token(s) to some or all of its outputs according to the given condition [Bock 2003]. Figure 3-32 illustrates the translation of the action node in an AD into transition node in a CPN. *Call-Behavior-Action* is a special kind of action in an AD, provided to invoke another activity at run time. In a CPN, *Substitution-Transition* represents the other net as a subnet and has semantics similar to that of *Call-Behavior-Action*. The transformation rule *create-substitution-transition* given in Figure 3-33 indicates the replacement of call-behavior-action node in an AD with a substitution transition in a CPN.

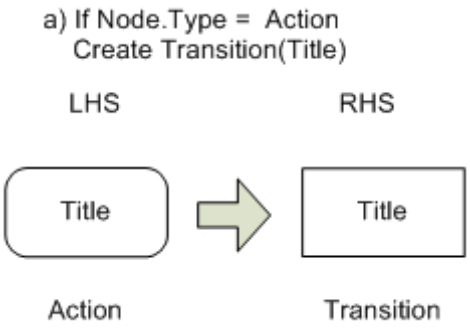


Figure 3-32: Transformation rule for Action to transition

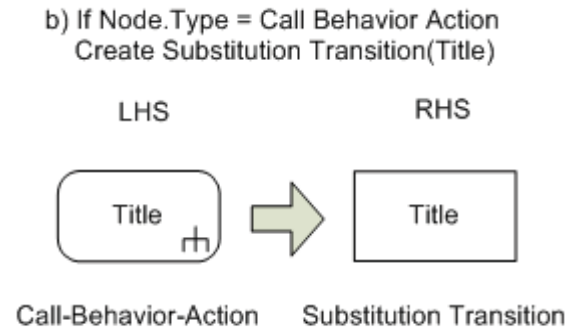


Figure 3-33: Transformation rule for Call-behavior action

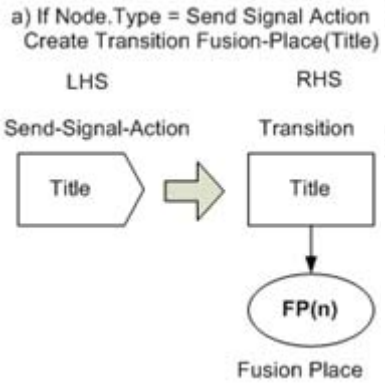


Figure 3-34: Transformation rule for Send-Signal action

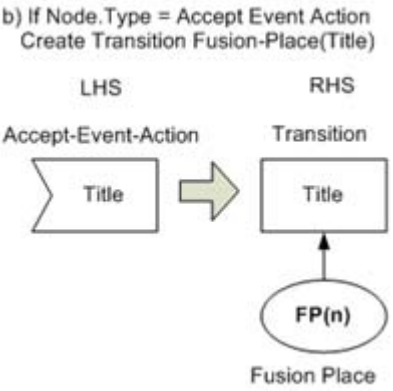


Figure 3-35: Transformation rule for Accept-Event action

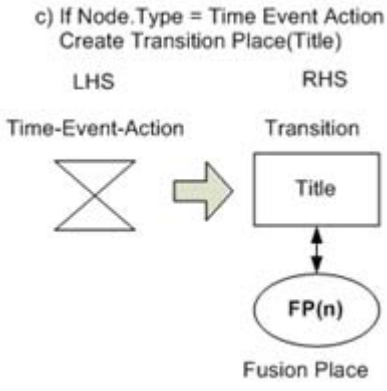


Figure 3-36: Transformation rule for Time-Event Action

In AD, *Send-Signal-Action* and *Accept-Event-Action* are defined to trigger and handle asynchronous communication. Similarly the rules *create-transition-fusion-place* in Figure 3-34 and Figure 3-35 describes CPN patterns for the *send-signal* and *accept-event* action nodes. For a

send-signal-action, a transition with an output fusion place is used to send signal token to a group of fusion places where one of them can randomly process this token. In case of accept-event-action, an input fusion place with a transition in CPN mimics the required behavior. A place with a bidirectional link to a transition node as defined in Figure 3-36 indicates that the given *accept-time-event-action* remains enabled until the containing activity is active.

Control nodes only route the token flow along the outgoing edges within an activity and does not operate/modify them. The transformation rules for five control nodes are presented in Figure 3-37, Figure 3-38, Figure 3-39, Figure 3-40 and Figure 3-41. The *create-place*, suggest replacing the initial, decision, merge and final types of control nodes in an AD with place nodes in a CPN. With *create-transition* rule as illustrated in Figure 3-42 and Figure 3-43, the transition nodes in CPN will substitute both fork and join types of nodes.

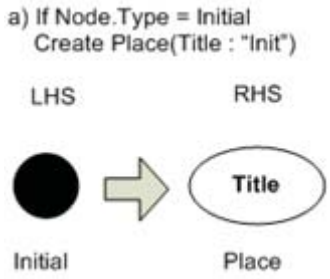


Figure 3-37: Transformation rule for Initial node

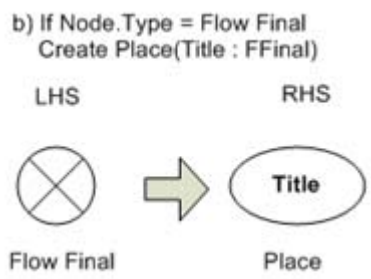


Figure 3-38: Transformation rule for FlowFinal node

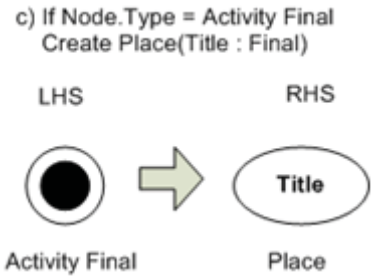


Figure 3-39: Transformation rule for ActivityFinal node

In AD, ActivityEdge is a directed link between two activity nodes and the connection between source and destination node types is quite flexible. However, a CPN model is a bi-partite, directed graph where two disjointed type of nodes: namely places and transitions are linked by arcs. An arc represents the flow between a transition and a place or vice versa and always connects two different types of nodes (i.e. place and transition). The arc between a place and a transition is called Input arc, whereas an arc which connects a transition to a place is known as Output arc. The translation of an AD model into a CPN without considering the arc restrictions in the CPN model would generate a model which is inconsistent to CPN semantics. Thus it is necessary to include ActivityEdge related rules as well. A synthesis of these rules is given in Figure 3-44 depicting the edge transformation in different scenarios. Application conditions indicate the execution of a rule only triggers when the edge have the specific types of source and target nodes, for example, when an edge links a fork node and an action node such as in condition-3 (con3) in Figure 3-44, it will be replaced by a place with a single input and output edge in a CPN model.

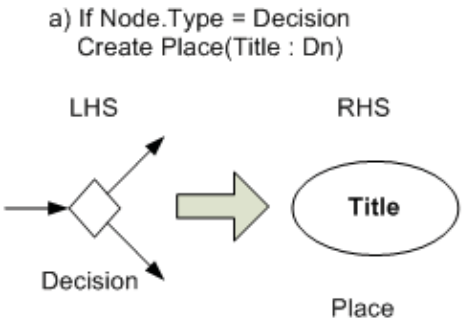


Figure 3-40: Transformation rule for Decision node

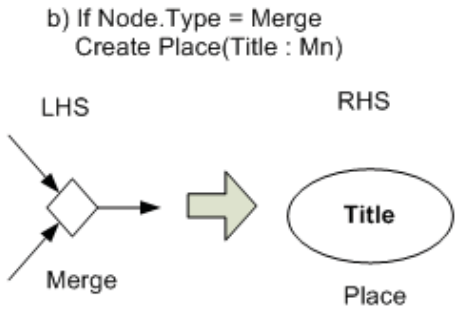


Figure 3-41: Transformation rule for Merge node

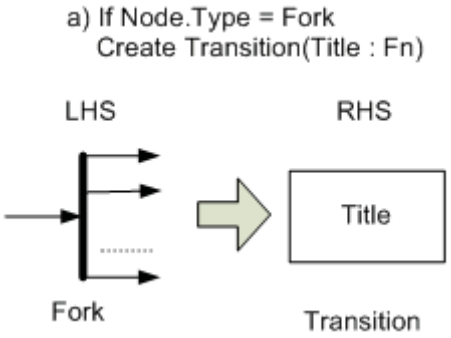


Figure 3-42: Transformation rule for Fork node

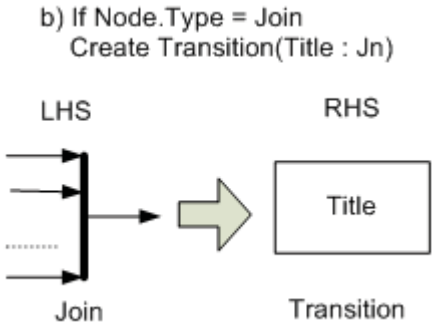


Figure 3-43: Transformation rule for Fork node

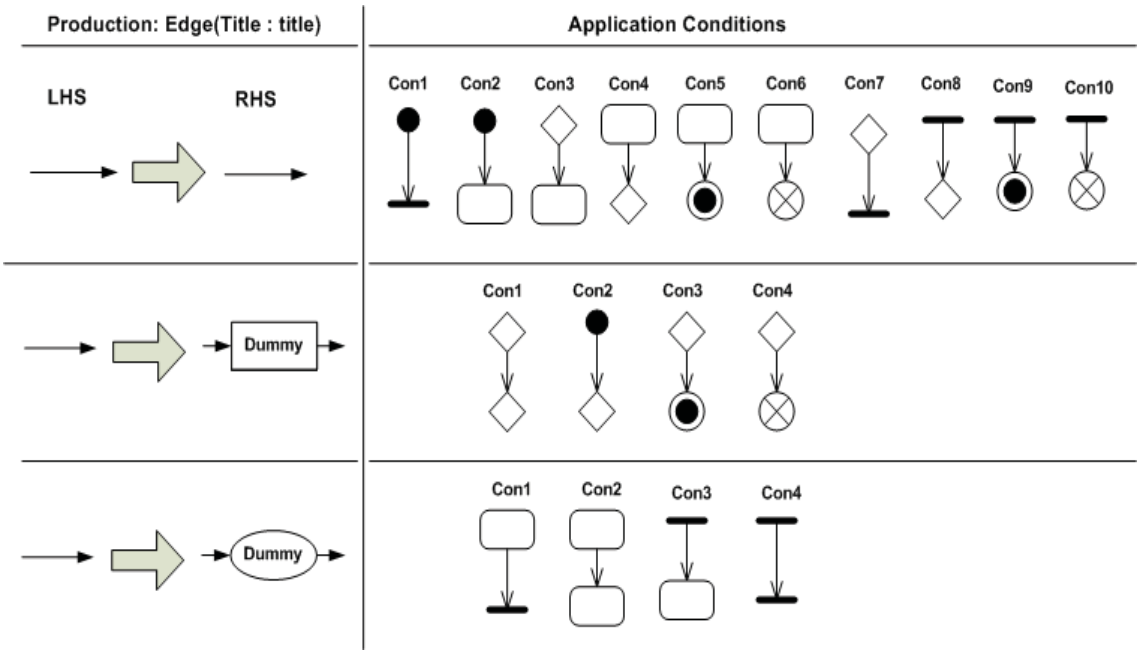


Figure 3-44: Production rules for control flow edge

Typically, an activity model comprises of several activities depicting a behavior or functionality in an associated diagram. Given the previously defined rules, each activity and associated activity diagram can be effectively transformed into a separate net. As in Section 3.4.1.3, it is noted that CPN support composition of multiple nets a.k.a. pages (sub-pages) through transition and place fusion mechanism [CPN-Group 2010]. In AD, the modular formation defined in terms of behavior invocation is structurally analogous to the page modulation in CPN. Accordingly, invoked-activities or sub-activities can be added as sub-Pages to the main CPN model and link them to the invoking actions/activities in the super-page or other sub-pages using *Substitution-Transitions* (ST). The *Substitution-Transition* also needs mapping it to a sub-page by defining and linking port and socket places on sub and super pages respectively.

AD is provided with two behavior invoking mechanisms; the *call-behavior* action and the *call-operation* action are direct and indirect invocation mechanism respectively. The UML2 specification is silent about the way the target method of *call-operation* action will be determined. However, at runtime the *call-operation* action will obviously be resolved so it is postulated that the call to the associated method is already resolved. Consequently regardless of whether a behavior is invoked using *call-behavior* action or *call-operation* action, the transformation would be the same. Moreover, as the invoking and invoked activities can be in the namespace of a single model or different models, it is assumed that all the referred and referencing activities are available for transformation.

Integrate behavior as subpage: Activities defined as behavior make up the modules of the system in the model and could be reused when required. Once the top level activity is translated into a CPN, the activities referred through *call-behavior* or *call-operation* actions will be accessed in a depth-first manner and nested into the CPN model as a subpage after translation into a CPN. In order to resolve the potential reuse of an activity to same corresponding page consistently, the name of each activity will be used as the label of the corresponding subpage.

Call-behavior/ Call-operation action as Substitution-transition: Similarly for *substitution-transition* in CPN, both the c*all-behavior* and *call-operation* actions represent a behavior as a module and link to it during execution when all prerequisites are satisfied [OMG 2007, page 248-250]. The *substitution-transition* will replace both *call-behavior* and *call-operation* actions in the CPN and attached to the associated subpage. In order to observe the design constraints, such as equal number of inputs and outputs, are given to establish the consistency between invoking actions and invoked behavior, the socket and port places will be referred whenever needed.

3.4.2.2.2 Production Rule Processing

Since UML models can be serialized as XML (eXtensible Markup Language) using the XML Metadata Interchange (XMI), and CPN model are serialized in a XML-based file, implementing model transformations using XSLT, which is a standard technology for transforming XML, seems very appropriate. XMI is an industry as well as an international standard framework endorsed by both OMG and ISO for defining, interchanging, manipulating UML models through XML based data and objects [ISO/IEC 2005; OMG 2007]. It provides production rules which enable the various UML modeling tools, applications and repositories to exchange UML models by serializing the models into XML documents. It can be used to exchange models between UML and non-UML tools using standard XML-based transformations mechanism.

The Petri Net Markup Language (PNML) is a part of the international standard on Petri Nets [ISO/IEC 2009]. The part 2 of the standard defines PNML, an XML-based structure of a Petri net file and interchange format for Petri nets. The main feature of PNML is that it supports both the general features of all types of Petri nets and specific features of a particular Petri net type. However, the specific features are handled through a separate Petri Net Type Definition (PNTD) for each Petri Net type. Although PNML is an ISO standard for interoperability among Petri Net tools, it is currently not supported by CPN Tools which was selected for to support further investigations in this study. However, the native model file of CPN Tools is XML-based and its format is well defined using a Document Type Definitions (DTD)-based schema [CPN-Group 2010].

XML is a meta-language for describing the markup of different types of documents and a considered as one of most widely used technique for data presentation and exchange. The key feature of XML is that it is a simplified subset of SGML (Standard Generalized Markup Language) so it is easier to learn, use, and implement than SGML while supporting full features like validation, structure, and extensibility. The XML specification stipulates the requirements for XML documents, data objects and XML processing programs for reading such documents and accessing their content and structure. XML documents are composed of entities, which are storage units containing text and/or binary data. Text is composed of character streams that form both the document character data and the document markup. Markups describe the document's storage layout and logical structure. A well-formed XML document is unambiguous, so that a browser or editor can read the tags and create a tree of the hierarchical structure without reading its schema. XML provides a standard way for information providers to add custom markup to information-rich documents, so that complex documents can be rendered and published in a dynamic way. Another key feature is the structure of XML documents or the

presentation of data in the document can be defined separately in XML schema which can be used to validate the document as well.

As described in Section 3.3.2.2, XSLT is a de facto standard language for describing the transformation of XML-based documents in one format into another format. Typically, transformation rules are specified in XML-based Stylesheet and then using a XSLT execution engine (processor) these rules are applied on an input XML document to create a new document in same XML or another format. There are several open-source and commercial XSL execution engines available for use; Saxon, AltovaXML and Xalan are examples of some of the freely available XSL engines. As it is a declarative transformation technique, the XSLT execution engine loads the input XML document and performs transformation rules in batch mode which makes it suitable for large models.

Using XSLT-based model transformation mechanism, a direct AD to CPN model transformation can be expressed through an automatic application of transformation rules described in a XSLT Stylesheet. Formally, the AD to CPN model transformation described by $T: M_{AD} \overset{R}{\Rightarrow} M_{CPN}$ is obtained by applying transformation rules $R: LHS \rightarrow RHS$ to an AD model by finding the occurrence of the transformation rule's LHS in the source model M_{AD} and applying the transformation rule *R* for each occurrence which finally yields the resultant model M_{CPN}. Therefore, the XSLT-engine primarily reads the input XMI file containing the AD model, parses and converts it into a document object model (DOM) tree. Then the sequential execution of the transformation rules that includes searching and matching the occurrences of LHS of a rule in the tree occurs. For each occurrence, the RHS of the rule is applied by creating new artifacts. Finally, results are structured and written into a CPN schema compliant XML file. The high-level description of the transformation is graphically shown in Figure 3-45.

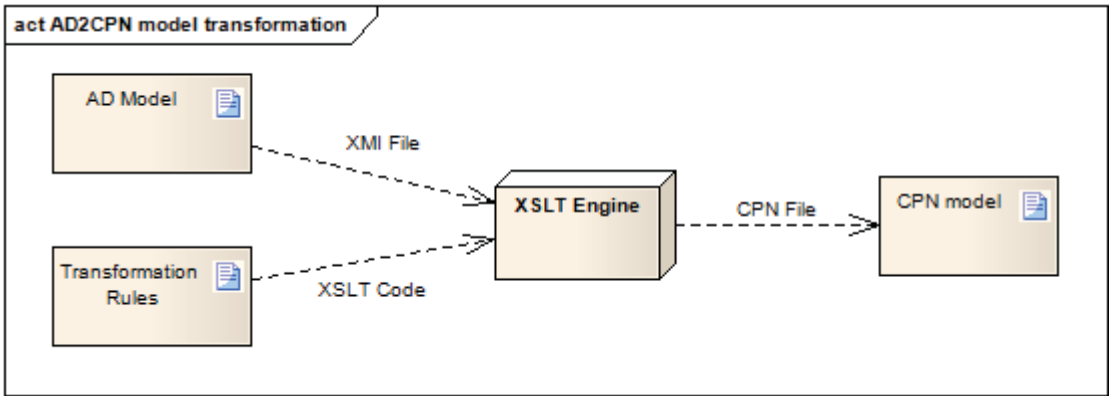


Figure 3-45: XSLT -based AD to CPN model transformation

3.4.2.3 Post-translation steps:

The implementation specific issues such as input, output, condition and annotation, are dealt with in the post-translation step. There are several CPN implementations and associated tools available (e.g. CPN Tools, PNetLab, ALPHA/Sim and CPN-AMI [2010]). Most of the available CPN modeling tools come with an inscription language but none of them uses the same language. Nevertheless, a specific tool and language needs to be selected to specify data, guard and annotation on the model. Due to the comprehensive toolkit and technical support, *CPN Tools* [CPN-Group 2010] which comes with CPN ML (an extension of Standard ML) was chosen for reference and demonstration purposes. Following is the specific auxiliary data that is needed for a CPN model in *CPN Tools*.

- **Data binding and Processing:** In CPN each place has a specific type and requires defining a color set. Similarly each arc needs binding with a variable of linked place type. For example in Figure 3-46, the STRING is the color type of the 'Initial' place and arc inscription 'str' binds the arc to a variable 'str' of STRING type.
- **Guard inscription:** Each guard condition in AD needs to define the equivalent conditions on the corresponding element using *Standard ML* which is a general purpose functional programming language. Figure 3-47 shows an example of guard inscription in *CPN Tools*.

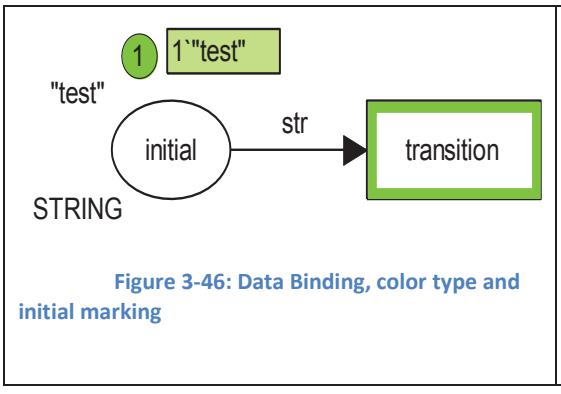


Figure 3-46: Data Binding, color type and initial marking

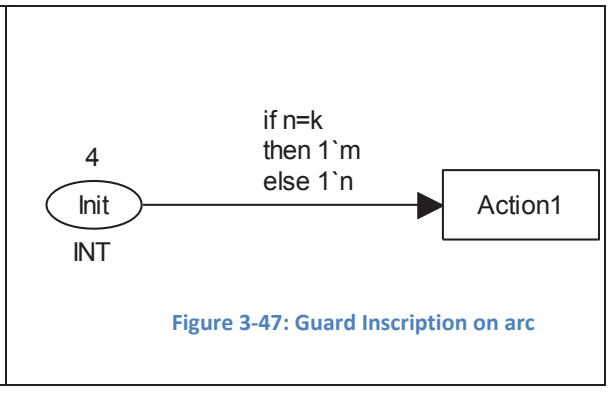


Figure 3-47: Guard Inscription on arc

- **Code segment and Transition binding:** For some specific behaviors, such as execution stoppage at an *Activity final* node and token merging at a *Join* node, such functionality using *Standard ML* expression in the model needs to be implemented. A breakpoint function is required to depict run-time behavior of an *Activity final* node. It is postulated that an activity execution will only

complete at the top-level and therefore the breakpoint monitor will only be attached to one or more *Activity Final* nodes on the super-page whereas all other *Activity Final* nodes on sub-pages will only return control back to the invoking activity. Similarly, a code segment is also needed for token merging behaviors according to the rules defined in [Bock and Gruninger 2005] for CPN transitions which are mapped Join nodes in AD. The code segment executes when a linked transition executes.

- **Initial Marking:** In order for a model to execute, it needs to define initial marking by placing initial tokens on the required initial places (see Figure 3-46). It is also necessary to provide initial marking in the case of multiple entry points, such as accept-time-event action, accept-event action and send-signal action.

3.4.3 Transformation Validation

In order to determine that the transformation of a source model into a target model, from one modeling formalisms to another modeling formalism or one abstraction level to another abstraction level, is valid and correct it is necessary to evaluate such transformations. Following are the four qualitative criteria have been defined to evaluate a transformation [Küster 2006].

- **Syntactic correctness:** The purpose of the syntactic correctness criterion is to determine if the output model conforms to the syntax of target modeling formalism.
- **Semantic equivalence:** In order to ensure that the transformation is valid, this criterion requires that the target model is semantically equivalent to the source model. A transformation is considered invalid if it yields target model which is syntactically correct but semantically not consistent with the source model.
- **Termination and confluence:** One of the important properties of valid and correct transformation is that the transformation always produces consistent results both syntactically and semantically. The termination and confluence criterion ensure that the transformation does not produce intermediate or inconsistent outputs.
- **Safety or liveness properties:** This criterion is used to determine that the transformation preserves the security or structural properties in the target model.

To put these criteria in perspective, the purpose of transforming an AD model into a CPN model is to get a valid executable model. The syntactic correctness is a basic requirement for an output model and for a CPN model, the checking for syntactic correctness can be achieved with built-in syntax analyser of *CPN Tools* [CPN-SC 2010]. The syntax analyser of *CPN Tools* automatically starts checking syntax of loaded model and highlights the syntax errors with colour coded auras. Aforementioned that the new semantic defined for AD is based on Petri Nets. However, there are some variations points which are pertinent to this study and addressed in Section 3.4.1.2. The transformation rules are defined to produce the semanticaly equivalent CPN model. The evaluation of semantic equivalence after transformation is either not achievable or not required at this stage for different reasons. First, the two often used techniques, trace equivalence and bisimulation, for determining similarity between two bahvioral models cannot be used due to the fact that the AD models are not executable. Second, one of the objectives of this study is to generate test suite from AD by transforming it into an executable CPN model. In the case the transformation is incorrect and both the source and target model are not equivalent, the test sequences generated from the target (CPN) model will not be adequate to the source (AD) based coverage criteria. The coverage criteria based analysis is addressed as test suite evaluation in next chapter. The other two transformation evaluation criteria are not performed for the same reasons.

3.5 Transformation of Case Study Models

For evaluating the proposed transformation methodology, an experiment with four case study models described in Section 3.2 was conducted. The pre-translation step as specified in Section 3.4.2.1 was performed manually to resolve the control-flow related ambiguity. For example, the refined ECCS model is shown in Figure 3-31. For transformation of case study models into CPN, the production rules defined in Section 3.4.2.2 were implemented with XSLT Stylesheet. The refined case study models were exported into XMI format. In order to execute the transformation Stylesheet, Saxon execution engine was used and the output CPN models were serialized to '.cpn' files. The XSLT transformation templates and output CPN models are presented in Appendix A. In order to validate transformation, syntactic correctness criterion was applied via two mechanisms. First, the document type definition (DTD) for *CPN Tools* was used to ensure that the output '.cpn' conforms to the required format. Second, the output '.cpn' files were loaded into *CPN Tools* and further syntactically validated with the built-in syntax analyser.

3.6 Summary

The chapter describes the case models and a methodology for transforming an AD model into a CPN model which is an executable model. Various issues related with the AD to CPN transformation were discussed. First, the common sources of ambiguity in AD models that may result into an incorrect transformation are elaborated and a model refinement solution is proposed. Second, the syntactic and semantic difference between the AD and CPN are detailed and a set of transformation rules are proposed for a rigorous and consistent transformation. Third, the CPN tool specific details that are needed for model simulation are identified. A three stage transformation methodology is developed to address each of these issues and evaluated with case study models. Resulting CPNs and related artifacts on this process are provided in Appendix A.

Chapter 4 Activity Diagram Based Testing

The previous chapter describes four case study AD models and a methodology developed as part of this study to transform an AD model into a CPN model. In this chapter, the potential of model based testing using AD is explored and a dynamic analysis based technique for deriving test sequences from AD models is proposed. To evaluate the adequacy of the generated test suites, two evaluation techniques have been developed, namely distribution analysis and mutation analysis which are coverage-based and fault-based techniques respectively. Furthermore, two types of coverage criteria were defined for evaluating the test suites through coverage-based analysis. In addition, this study defined a set of mutation operators to support mutation analysis of AD models and AD-based test suites. Lastly, a controlled experiment using case study models described in Chapter 3 was also conducted to evaluate the proposed test case generation and evaluation techniques and associated results were reported.

Concurrent software must be designed to take advantage of the multi-core and multi-processor systems in order to meet ever increasing user demands and leveraging of the recent hardware developments. Many formal languages like Petri Nets and Communicating Sequential Processes (CSP) have long been used for designing and analyzing concurrent systems. A number of studies [Kersten and Nebe 2004; Shousha et al. 2008] have also been conducted to exploit the concurrency characteristics of UML diagrams such as State-chart and sequence diagram in detecting concurrency faults (e.g. deadlock and race conditions). Among the UML diagrams, Activity Diagram (AD) is ideally suited for depicting parallel or distributed processing [Bock 2003]. In particular, it provides standard notations and semantics that can clearly represent multithreaded or multi-process design and readily supports the accurate description of concurrent work breakdowns. For example, the fork and join nodes are used to depict creation and synchronization of threads or processes[2] respectively. Furthermore, it supports a wide range of applications in the development cycle including process sketching and executable program

[2] Generally, the difference between process and thread is granular and a thread usually contains in a process. The implementation differences between them vary from one operating system to another. Usually, multiple threads can exist in a process and share resources whereas multiple processes have separate address spaces and they do not share resources.

specifications [Bock 2003]. The concrete syntax of AD is defined in multiple layers which allow modelers to use it for both high level abstract models (i.e. requirements or Computational Independent Model in MDA) and low level detailed models (i.e. program design or Platform Specific Model in MDA). For a complete description of a task (usage scenario) in the system, AD provides high-level notations and token-flow semantic for an accurate and concise depiction of the use case. In order to specify how a procedure or method should be accomplished, control and data flow notations support the detailed modeling of the implementation.

4.1 AD Based Test Sequence Generation

An AD model depicts logical paths in a program that can or should be followed in its implementation based on various conditions such as concurrent processing, data access and interruptions. These logical paths can be used to verify that the implementation is correct and is as expected. Similar to the execution paths in code based testing, it is the execution sequences of model artifacts that interest testers more than the execution of an individual artifact. Therefore, the execution traces of the model are recorded during its simulation and these traces then constitute test sequences for the implementation (of the model) under test. A test sequence is defined as a path through the model from its initial node to its final node. In AD-based testing, typical test-cases are sequences of operations or usage scenarios that reflect both the input-output relation as well as the possible sequences of interaction. The test cases generated from an AD determine the functional correctness of tasks or operations, order of execution and the dependencies among the various tasks or operations.

Investigations involving the application of PN in software testing can be categorized into three groups:

1. Test suite generation using typical state-space analysis techniques [Watanabe and Kudoh 1995]
2. Test suite generation using invariant analysis [Ramaswamy and Neelakantan 2002], and
3. Deriving test scenarios by simulating or executing the PN models.

Watanabe and Kudoh [1995] proposed two CPN based algorithms for the automatic test suite generation in conformance testing involving concurrent systems. Their CP-tree method requires the generation of a reachability tree from a CPN model and then test sequences are produced by traversing through arcs and nodes from the root to the leaf nodes of the CP-tree. In their CP-graph method, the CP-graph is generated from a CP-tree. The W_p method is applied to generate the test suite if the resulting CP-graph satisfies the pre-conditions associated with

the W_p method. The authors noted that the developed approach performs better than other existing approaches. The authors also claimed that the test suite generated from the CP-tree method has the same fault detection capability as that of the FSM-based method.

Ramaswamy & Neelakantan [2002] showed the application of a PN based invariant analysis technique in software design and testing. Their approach aimed to generate unique paths dubbed as "sub-flows" using the T-invariants obtained from a PN model. While the approach avoided the state explosion problem associated with model checking, it requires a high level of mathematical skills, thus inhibiting its applications at the industry level.

Zhu and He [2002] proposed four types of techniques, for testing Predicate Transitions Net (a variant of high-level Petri Nets), which is based on the general theory of testing concurrent software systems. They also defined separate classes of test evaluation criteria for each type of techniques. Transition and state-based techniques focused on testing all the transitions and their sequences, as well as all states and their sequences in the model. The third type, a flow-based technique focuses on token-flow testing which is similar to conventional data-flow testing. Using PN as a formal algebraic specification of the Net, the fourth approach aims to use the existing specification-based testing techniques (e.g. mutation testing and formal algebraic for verification and validation of both specification and implementation). Additionally, the authors proposed an observation based scheme to determine the system's dynamic behavior during concurrent testing. They found that observations in the form event sequences of various possible dynamic behaviors are more appropriate and pragmatic than test data adequacy criteria for testing concurrent systems.

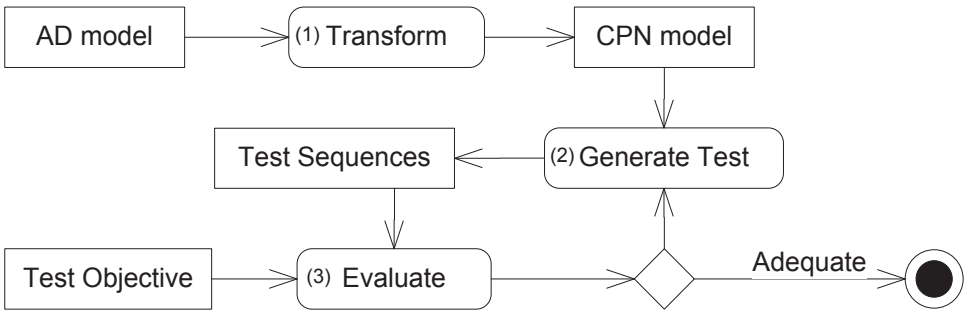


Figure 4-1: Three steps in an AD-based test sequence generation

Given the new token-game semantic defined for AD in UML2, a token can abstractly represent a control or stimuli in a system and the flow of these tokens in AD models can be used

to simulate the behavior of the intended or implemented system. However, the semantic of AD is informally defined and AD models are not suitable for static (formal) or dynamic (model execution) analyses [Störrle and Hausmann 2005]. Thus, in order to derive test suites from an AD model, a three step test sequence generation (TSG) process as illustrated in Figure 4-1, was developed as part of this study.

1. In the first step of the TSG process, the transformation of an AD model into a Colored Petri Nets (CPN) model using the approach developed in Chapter 3 was proposed and demonstrated.
2. In the second step, test sequences were generated through the execution of the CPN model and the control flow traces are recorded. A guided random-walk based stochastic algorithm was proposed for the random execution of the CPN model and details of this algorithm are described in the following section.
3. Finally the generated test suite was evaluated against a given test objective. If the test suite did not satisfy the required objective, more test sequences would be generated through another iteration of the guided random-walk algorithm. This process was repeated until the required criterion specified in test objective was satisfied, at which time the process terminates.

4.1.1 Test Sequence Generation Algorithm

Prior to the description of the proposed TSG algorithm, the concept of random-walk is briefly described. The random-walk algorithm is derived from the theory of probability and it refers to a trajectory where the path is initiated from a specific point and from which each successive step is made randomly. The trajectory of a random walk includes all visited nodes in a connected graph. In general, a random-walk is considered suitable for discrete problems and often needs adaptation for better results according to a particular application. Examples of adapted versions of random walk include a self-avoiding walk to generate non-intersecting paths [Madras and Slade 1996], a reinforced random walk tailored to exploit the information in weighted graphs [Renlund 2005] and the exploration process for state space analysis [Sivaraj and Gopalakrishnan 2003]. This apparently simple technique has received a fair amount of attention and has been applied in areas such as networking [Chen and Bhaskar 2008; Christos et al. 2006], image segmentation [Grady 2006], web search [Fortunato and Flammini 2007], state space exploration [Pelánek et al. 2005] and model checking [Sivaraj and Gopalakrishnan 2003]. In this study, it has been adapted for model-based test suite generation.

The application of the random-walk algorithm in software testing has been reported in a number of studies [Lee et al. 1996; Pelánek, Hanžl, Černá and Brim 2005; Sivaraj and Gopalakrishnan 2003]. Sivaraj and Gopalakrishnan [2003] proposed a random walk-based approach for model checking in parallel and distributed environments together with a breadth first search. They defined four heuristic-based algorithms with a configurable coupling between a random-walk algorithm and a breadth first search for state space exploration. The objective of their work was to explore the potential of heuristics with various combinations of a random-walk algorithm and a breath first search in detecting bugs that are difficult to find. Lee et al. [1996] presented the idea of using the random-walk algorithm for generating test sequences from Communicating Finite State Machine (CFSM) in conformance testing. According to the method, an adaptable random-walk algorithm is guided by classified transitions in a directed graph and visited states are sampled for test traces. Pelánek et al. [2005] presented an empirical analysis of a random-walk approach with various factors for state space exploration and proposed many performance enhancements to the random-walk algorithm.

The natural graph based representation of a CPN makes it an ideal candidate for the application of the random-walk algorithm. The random-walk algorithm is adapted for generating traces from a CPN. The formal definition and semantics of CPN which can be found in [Jensen 1997] are first described in the following section.

A Colored Petri Net is a tuple $N = (\sum, P, T, A, N, C, G, E, I)$ where

- $\sum$ is a finite set of non-empty types and called $color\ sets$,
- P is a finite set of places,
- T is a finite set of transitions,
- $A \subseteq (P \times T) \cap (T \times P)$ is a set of arcs,
- N is a node function $N: A \longrightarrow (P \times T) \cup (T \times P)$,
- C is a color function $C: P \rightarrow \sum$,
- G is a guard function $G: T \rightarrow \{expressions\}$ such that $\forall t \in T: \left[Type\left(G(t)\right) = true, false \wedge Type(Var(G(t))) \subseteq \sum\right]$,
- E is an arc expression function $E: A \rightarrow \{expressions\}$ such that $\forall a \in A: \left[(Type(Var(E(a))) \subseteq \sum) \wedge Type\left(E(a)\right) = C(p(a))_{MS}\right]$ where $p(a)$ is the place of $N(a)$ and
- I is an initialization function $I: P \rightarrow \{closed\ expressions\}$ such that $\forall p \in P: \left[Type\left(I(p)\right) = C(p)_{MS}\right]$.

For a given net, a marking is a mapping from P to natural numbers, indicating the number of tokens in each place such that $M: P \rightarrow \mathbb{N} \cup \{0\}$. A marking represents the state of the system in its CPN model and M_0 indicates the initial marking of the Net (state of the system).

For a transition $t \in T$, $\cdot t = \{p \in P | (p,t) \in A\}$ represents the input set of t, and $t \cdot = \{p \in P | (p,t) \in A\}$ represents the output set of t. Similarly, for a place $p \in P$, $\cdot p = \{t \in T | (t,p) \in A\}$ and $\cdot p = \{t \in T | (t,p) \in A\}$ represent the sets of inputs and outputs of p.

A transition t is enabled for execution if every input of t contains enough tokens (must be equal or more than what is required), and its execution conditions are satisfied. When a transition is executed the marking changes in a way that tokens from the input places are removed and added to the output places. The number of tokens removed from an input place equals the weight of the corresponding input arc. Two transitions execute concurrently if they are not in conflict. Conflicts are resolved non-deterministically (e.g. using a stochastic or a probabilistic function). The execution of an enabled transition is atomic. It is assumed that the set of all possible execution sequences E represents the behavior of a net and thus constitutes the potential test set. An execution trace $e \in E$ is a sequence of transitions that can be executed starting from the initial marking and each successive marking is obtained through a transition step (transition execution) in the order indicated. It can be denoted as $e: m_0 \xrightarrow{F_0} m_1 \xrightarrow{F_1} m_2 \xrightarrow{F_2} \ldots \xrightarrow{F_{k-1}} m_k \xrightarrow{F_k} \ldots$ where $F_0$is a transition execution sequence, and it represents the sequence of transitions that were executed to reach $m_1$from m_0, $m_0 \in M_0$is an initial marking, $m_i, i = 1,2, \ldots$, are marking such that m_i is obtained from $m_{i-1}$by executing transition sequence F_i.

Table 4-1: CPN nodes with corresponding AD nodes in brackets and observing token-game semantic

Node	Semantic
Transition (Action)	▪ An action can only start execution when all inputs have tokens. ▪ When an action starts execution it consumes tokens on all inputs. ▪ On completion, tokens are offered on all outputs.
Place (Initial)	▪ Initialize with a token whenever the enclosing activity is invoked. ▪ An outgoing token can follow only one edge.
Place (Activity Final)	▪ When a token reached in it, the enclosing activity will be terminated; particularly, all executing actions are stopped, all other tokens are destroyed and all flows are terminated.
Place (Flow Final)	▪ All tokens arriving on it are destroyed.
Transition (Fork)	▪ Incoming tokens are duplicated to all outputs.
Transition (Join)	▪ All incoming tokens are joined according to the rules given in (Bock, 2003).
Place (Decision)	▪ Each incoming token can traverse only one outflow.

Place (merge)	▪ All incoming tokens are forwarded to a single outflow without synchronizing and joining them.

Owing to the stochastic nature of the random-walk algorithm, it may generate traces (paths) which are illegal according to CPN semantics. In order to avoid this problem, the random selection process has been adapted to incorporate the predefined semantics for CPN in this study. In CPN, the flow of a token can represent the control flow in the model and therefore the random-walk algorithm will simulate the token-flow during the model execution. Moreover, as the technique is based on a pseudorandom exploration of the model, the model inscriptions such as conditions and data information are not used during the random-walk algorithm. However, it is important to note that the random-walk algorithm can further be adapted to handle model inscription during branch selection. Furthermore, as in the case of CPN, one of the many enabled transitions will eventually occur or fire so it is therefore postulated that the walker randomly selects a transition, and in visiting it, makes one step of the walk. Similarly, the traversal of the walker through a place node is also marked as a step of the walk.

At the start, the random walk begins from an initial place node dubbed as '*Init*' in the CPN model. The walker then randomly selects one of the outgoing arcs according to the corresponding semantics of the initial node as shown in Table 4-1 and the given token moves along the selected arc. After the occurrence of a transition, a token is passed to each output place. The walker continues as long as it is visiting nodes with nonzero outgoing arcs. Once it reaches a node without any outgoing arc, this implies that the *Activity-Final* node has been reached and therefore the walk terminates for the current iteration. Using the random walk approach, test sequences are automatically generated at the end of each iteration by recording the trace of the random walk starting from the initial node to the final node. The pseudo code for the algorithm is given in Figure 4-2. The random-walk approach is adapted to make the next move at a particular node according to the semantic specified in Table 4-1. In a conventional random-walk algorithm, all subsequent transitions become active and eligible for execution at each step of the walk. In this adapted version of the algorithm only those transitions that have satisfied the CPN semantics (i.e. all input tokens are available, all execution conditions for a transition are satisfied) are to be enabled.

```
If(Model is not initialized)
    initialize the model;
End if
Repeat until ActivityFinal place is not marked
    For all enabled transitions;
        If(all inputs and postconditions are satisfied)
            Fire an enabled transition randomly;
            /*Firing a transition will remove the input token
            and pass it on to the output places.*/
        End if
    End loop
    If(Fired transition is not dummy)
        Record the fired transition to token trace;
    End if
    If(New marked place is not dummy)
        Record newly marked place to token trace;
    End if
    For all transitions
        If(all inputs and precodotions are satisfied)
            Enable the transition;
        End if
    End loop
End loop
Output the token trace;
```

Figure 4-2: Pseudo Code for the Random-Walk Algorithm for TSG which is adapted from [Pelánek, Hanžl, Černá and Brim 2005].

As indicated earlier the output of the second step of the AD based test suite generation are test sequences which are subsequently evaluated to determine if the specified testing criterion are met. The evaluation process involves coverage-based and fault-based techniques.

4.1.2 Test Suite Evaluation

Measurement is a prerequisite to quality control and project management. DeMarco stated that you cannot control what you cannot measure [cited in Fenton and Pfleeger 1997]. A measure is used to quantitatively characterize an attribute of an entity under observation and constitute as a basic building block of any measurement system. A measurement for an attribute of an entity usually has a standard unit of measure, for example inch is a unit of length and gram is a unit of weight. Similarly, in software engineering, several specific measures such as mean time to failure (MTTF) and the number of faults found per KLOC (thousand lines of code) are being used for evaluating the reliability and quality of a program respectively.

As discussed in Chapter 2, a number of measures (e.g. node coverage, branch coverage and mutation adequate test suites) have been reported for evaluating the adequacy of a test suite [Zhu, Hall and May 1997]. Coverage criteria are generally used to determine the adequacy

of a test suite and are therefore considered an essential part of any testing method. A general rule of thumb in testing is that a test suite with a higher coverage is considered to be of a better quality. This is based on the fact that a test suite with a higher coverage can reveal more defects than a test suite with a lower coverage [Frankl and Weiss 1993; Wong et al. 1994]; which ultimately improves the quality and reliability of a software under test. The test coverage is measured in terms of the percentage of specific constructs that have been executed at least once during execution according to the defined coverage criterion. Comprehensive reviews of various code coverage based test adequacy criteria and UML model-based coverage criteria can be found in [Zhu, Hall and May 1997] and [McQuillan and Power 2005] respectively. Mutation testing, a fault-based technique introduced by DeMillo et al. [1978], provides an alternative measure for assessing test adequacy of a test suite.

Mutant programs are generated by introducing faults into the code of the program under test, with each mutant program containing a single fault. The test suite is then assessed in terms of how many mutants it distinguishes from the original program.

AD based testing would require the testing of ordered executions of tasks or operations in isolated control paths or threads as well as the coordinated execution of tasks or operations in synchronous or asynchronous parallel control paths or threads. In the following sections, coverage-based and mutation-based evaluation techniques for AD-based test suites are introduced. Two sequential and concurrent criteria, adapted from [Zhu, Hall and May 1997] and [Factor et al. 1996] respectively, as well as mutation operators developed in this study for generating mutants of AD models. The coverage-based criteria will allow the systematic analysis of the coverage information for AD-based test suites and for assessing its adequacy with respect to a given coverage criterion. The mutation operators will allow the generation of mutant models of an AD model and for assessing the adequacy of a test suite in terms of its fault detection capability.

The next section describes sequential coverage criteria developed as a part of this study. The proposed criteria has been adapted from those proposed in [Zhu, Hall and May 1997].

4.1.2.1 Sequential Coverage Criteria

An AD depicts transaction, control or data flow in a process or method of a system, depending on the level of abstraction. A start to end path in an AD is a sequence of actions or activities which starts and ends at the initial and final node respectively. Testing isolated control paths or threads is analogous to sequential control flow based testing. Control flow based testing has been extensively studied and a number of control flow based coverage criteria for

code-based testing have been proposed. As control based criteria are basically defined on a graph structure, they can be adapted for AD based techniques by exploiting the underlying graph structure and covering it. A sequential control flow based coverage criteria which will allow measuring and determining the adequacy of the AD based test suite, is presented in the following section.

4.1.2.1.1 Action/Activity Coverage

In AD, an action or activity node is executed when it offers a token onto a set of outputs $(O_1, O_2, \dots, O_n)$ that can be traversed after its execution. Action or Activity coverage would require the execution of each action or activity node in the model at least once. Therefore, the test suite includes a test case for testing each action or activity node at a minimum. This is analogous to node-coverage in state-based testing and could be considered as the elementary and minimal required testing criterion. It can be defined formally as follows:

***Definition**:* A test suite *T* satisfies the action or activity coverage criterion if and only if for each action or activity node *A* in an AD model there exists *t* in *T* such that t causes *A* to execute.

4.1.2.1.2 Branch Coverage

This is a control flow based criterion that measures the number of branches that have been executed at least once during testing. It ensures that all alternate paths have been evaluated during testing. For complete branch coverage, a test suite needs to have at least one test case for each branch which also includes the execution of all transitions. Therefore, branch coverage subsumes the Action/Activity coverage.

***Definition**:* Given a test suite *T* and an *AD* model, for each branch *b* in the model, *T* must cause each *b* to be taken at least once.

4.1.2.1.3 All Path Coverage

For complete testing, one would try to ensure that all possible executions sequences (Paths) in an AD-model have been executed at least once during testing. *All-Path coverage* criterion is the strongest. Unfortunately, analogous to all-path analysis in path-based testing, it is difficult to achieve for a non-trivial program. This is because the total number of all execution paths is usually very high, and in some cases it is possible to have an infinite number of all execution paths. The formal definition is as follows:

***Definition**:* A test suite *T* satisfies the all path coverage criterion if and only if for each path *P* in an *AD* model there exists *t* in *T* such that t causes *P* to be traversed.

The next section describes concurrent coverage criteria developed as a part of this study. The proposed criteria has been adapted from [Factor et al. 1996].

4.1.2.2 Concurrent Coverage Criteria

The development of concurrent software poses a set of challenges different from the development of sequential software. For example, deadlock and race conditions are two of the issues that can occur in concurrent software. Concurrency faults that lead to a deadlock or a race condition may only occur in a very small number of execution interleavings which means it is extremely difficult to detect via conventional testing. Conventional testing of sequential programs usually involves developing a set of test cases that provide a certain level of code coverage (e.g., path coverage). Furthermore, studies have shown that conventional test coverage criteria are inadequate for concurrent program testing due to the non-determinism of the execution of concurrent regions and the high number of possible interleavings [Factor, Farchi, Lichtenstein and Malka 1996; Tai 1989; Yang and Chung 1990].

An AD provides several basic primitives (i.e. fork, join, SendSignalAction and AcceptEventAction) for specifying concurrent designs. The fork node depicts the creation and invocation of new threads or processes that may execute concurrently. It splits the control into 'n' sub-processes and allows all of them to execute in parallel. The join node depicts the synchronization point for the concurrent threads or processes and is used for the scheduling of threads and for access to shared resources. SendSignalAction and AcceptEventAction are used to depict the communication between processes by sending and receiving messages or events in distributed processing configurations (i.e. client-server and interacting peers). Similarly, CallBehaviorAction is another AD construct which can be used to depict the invocation of remote operations in distributed computing. ExpansionRegion is particularly suitable for data or object level parallelism, however as the study is limited to control flow it is not covered here.

An AD supports the depiction of complex concurrency at various levels of granularity. Concurrency at instruction level is where multiple parts of a single instruction that may execute simultaneously is modeled with actions (e.g. Figure 4-3). Functional or operational level concurrency may be achieved through activities which are assigned to different threads and can execute simultaneously. Furthermore, objects created or assigned to a different thread or process may execute their methods concurrently and depict the object level concurrency. The execution of concurrent processes is an interleaving of actions or activities, depending on the granularity of the model. A particular execution of a concurrent program can be viewed as a trace of the sequence of actions or activities.

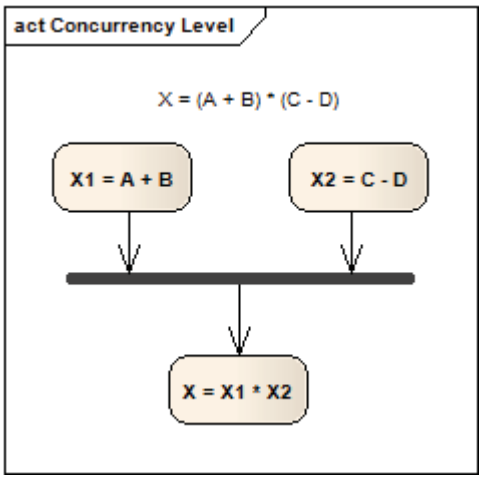


Figure 4-3: Instruction Level Parallelism (taken from [Hughes and Hughes 2003])

When executed, threads in a concurrent program work together to compute results and the interleaving space of a concurrent program consists of all possible thread or process schedules. A test suite that could reveal concurrency faults such as race conditions and deadlocks, must exercise these interleavings. The following sections describe the concurrent coverage criteria that has been adapted from [Factor, Farchi, Lichtenstein and Malka 1996] for the evaluation of the AD based test suite.

4.1.2.2.1 Synchronized path coverage:

The execution of the set of all possible interactions between concurrent threads or processes is required in order to satisfy the synchronized path coverage criterion. As the number of interleaving paths grows exponentially along with the number of threads or processes, attaining adequate coverage for this criterion is intractable.

***Definition**:* Let *T* be a set of test sequences for *AD* model *M*. *T* satisfies the synchronized path coverage criterion, if and only if for any feasible interleaving *I,* between concurrent threads of *M*, there is at least one $t \in T$ such that *I* is covered by *t*.

4.1.2.2.2 Interleaving edge coverage:

The selection of sufficient test cases such that all the n-wise permutated set of edges in $'n'$ synchronized threads or processes are executed at least once during testing is known as interleaving edge coverage. The percentage of the paired edges exercised during testing implies the adequacy level of the test suite.

***Definition**:* Let *T* be a set of test sequences for *AD* model *M*. *T* satisfies interleaving edge coverage criterion, if and only if, for any feasible edge sequence *E* of *M*, there is at least one *t* ∈*T* that covers *E*.

4.1.2.2.3 Interleaving node coverage:

The interleaving node coverage criterion requires the execution of n-wise permutated set of concurrent nodes in $'n'$ synchronized threads or processes. In AD, the execution of an action and activity node will make up the permutated set. The degree to which the permutated set has been exercised by a test suite implies the coverage attained according to this criterion.

***Definition**:* Let *T* be a set of test sequences for *AD* model *M*. *T* satisfies interleaving node coverage criterion, if and only if for any feasible node sequence *N* of *M*, there is at least one *t* ∈*T* that covers *N*.

4.1.2.3 Mutation Analysis of AD-based Test Suite

In the section, a novel mutation analysis based technique which has been developed for assessing and improving the fault detection capability of test suites generated from AD models is described.

One application of mutation analysis on AD model is the verification of the design correctness [Farooq and Lam 2008]. Here it is defined for the adequacy evaluation of a test suite generated from an AD model. Mutating an AD model is similar to mutating a program source and usually, a single syntactical change is introduced per mutant. Using the new token-game semantics of AD defined in [OMG 2007], a modeler can simulate the model and analyze the runtime behavior of the system. The test sequences are generated from the original AD model and then used to execute the mutant models. If a mutant model fails to execute these test sequences then it is considered dead otherwise it is deemed equivalent. In other words, if the generated test suite cannot kill all non-equivalent mutants then it is considered inadequate and more test sequences need to be generated.

The steps involved in the proposed approach for mutation analysis of an AD model are illustrated in Figure 4-4 and described as follows. At the “Mutant Generation” step, mutant models are generated according to a set of selected mutant operators. A mutant model is generated for each application of a mutant operator in the AD model. An initial suite of test sequences is then generated from the original AD model using the random-walk based test case generation technique at the “Test Generation” step. During “Test Execution” step, each mutant

model is transformed into CPN model and executed with all of the test sequences in the initial test suite. If the mutant fails in the execution of a test sequence then it is considered to be killed by that test sequence. If all of the mutants generated from the original AD model are killed then the generated test suite is considered adequate. If a mutant is not killed by any of the test sequences then it is necessary to determine whether the live mutant is an equivalent mutant. In the case where a mutant is found to be equivalent, it is separated and excluded from the mutation score. Otherwise, more test sequences are generated to kill the remaining live mutants. This process continues until all mutants are killed or separated.

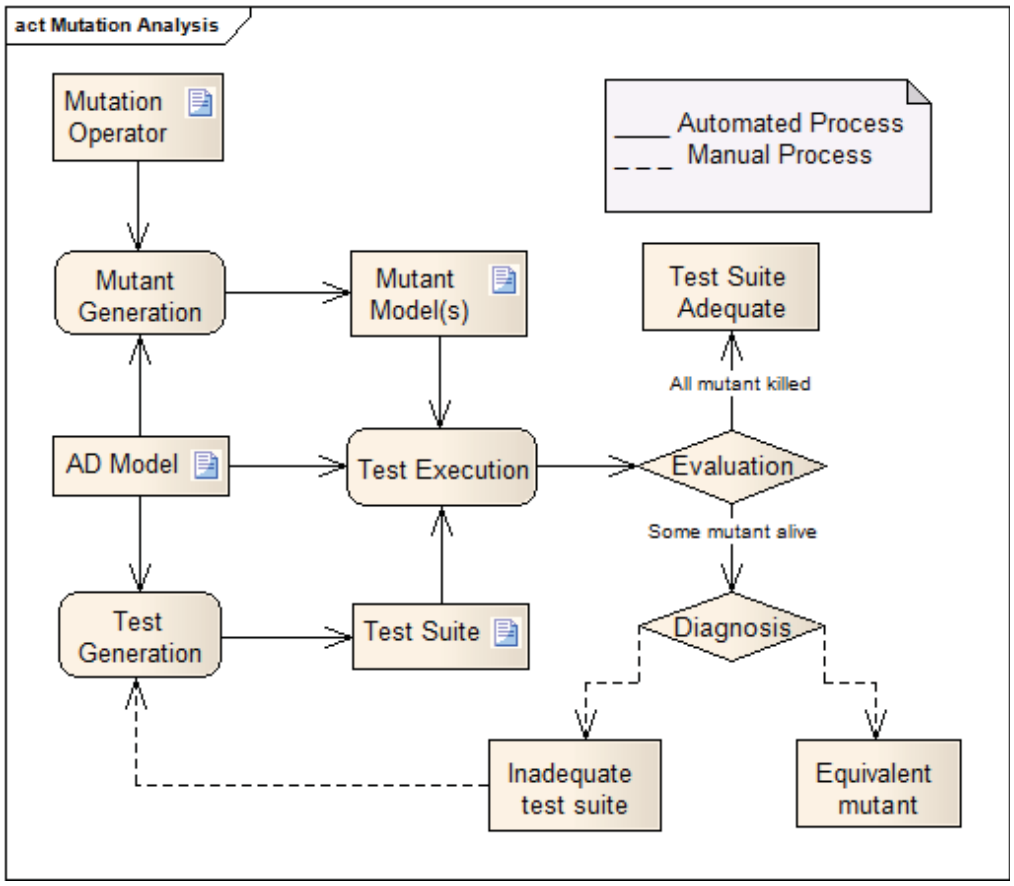


Figure 4-4: Model-based Mutation Analysis

In code-based mutation testing, a fault set is devised based on the simple errors that a competent programmer may commit in practice. In the case of AD mutation analysis, control-flow based fault types associated with semantic bugs that were referred to in a recent study on software error characteristics [Li, Tan, Wang, Lu, Zhou and Zhai 2006] were derived for this study. It is important to mention that AD supports modeling behavior with both control and object flow. However the mutation operators defined here are limited to the control-flow aspects of the AD model and are a minimal set of operators. The consideration for limiting the

scope of this study to control-flow view is as follows: (1) the semantics of control flow view is clear, well established and pragmatic; (2) the semantics of the object-flow view constructs has several ambiguities and inconsistencies [Schattkowsky and Forster 2007]; and (3) there are practical problems with the application of object-flow and high-level constructs [Schattkowsky and Forster 2007].

AD Mutation Operators

Mutation operators are represented as a set of rules that describe syntactic changes to the elements of the AUT. Nevertheless, the focus of the syntactic changes is to alter the behavior depicted in the model which will result in a failure to produce the desired outcome. Similar to the code based mutation analysis where it is assumed that the compiler will catch syntactic errors, it is assumed that model validation will detect the syntactic errors in a model. An example of model validation is available in Enterprise Architect which evaluates the well-formedness of a UML model according to UML specifications and reports errors for detected violations [Systems 2008]. To generate mutant AD models, mutation operators are applied to elements within the AD models and this requires the identification of a set of potential faults. A competent designer may encounter several types of faults in AD based modeling:

- Wrong sequencing of operations (i.e. actions or activities).
- Interface errors (i.e. missing input or output).
- Synchronization errors that may happen due to various situations, such as deadlock, livelock, starvation and race conditions.
- Decision errors.

These four types of faults can be implemented by simple syntactical changes in an AD-model. In order to systematically seed these faults in mutant models, four types of mutation operators are defined. To define mutation operators, the following definition of AD is adapted from [Xu, Li and Lam 2005].

Definition: Let $AD = (Z, E, D, M, F, J, A_I, A_F)$ be a 8-tuple Activity Diagram where

- $Z = \{z_1, z_2, \dots, z_n\}$ is a finite set of action nodes;
- $E = \{e_1, e_2, \dots, e_n\}$ a finite set of edges;
- $D = \{d_1, d_2, \dots, d_n\}$ a finite set of decision nodes such that $\forall d \in D, d =< c > b_k + < c > b_{k+1} + \cdots + < c > b_{k+n}$ where $B = \{b_1, b_2, \dots, b_n\}$ is a finite set of branches such that $\forall$b$\in$B, B $\subset$ E and c$\in$C, $C = \{c_1, c_2, \dots, c_n\}$ is a set of guard conditions;

- $M = \{m_1, m_2, \dots, m_n\}$ a finite set of merge nodes;
- $F = \{f_1, f_2, \dots, f_n\}$ a finite set of fork nodes;
- $J = \{j_1, j_2, \dots, j_n\}$ a finite set of join nodes;
- A_I is an initial node and A_F is an Activity-Final node.

For the application of mutation analysis on ADs, it is assumed that both mutation testing hypotheses are also valid; in that the designer is competent, and simple and composite faults have a coupling effect. It means that the AD model produced by the competent designer is either correct or close to correct, while the coupling effect means the test suite that can detect simple faults is also sensitive enough to catch complex faults as well. Moreover, it is hypothesized that the faults that a designer can commit in modeling system behavior can be detected earlier and fixed. The set of faults that can be injected into an AD model constitute as the operators for AD mutant generation.

Based on the fault types defined here, a set of mutation operators for ADs has been developed (and summarized in Table 4-2) and is specified in the following sections:

Operation Mutation Operator (OMO)

An activity represents a complex behavior which specifies the sequence and condition for execution of operations by directed links (edges). The links between the executable nodes indicate that the node at the target end of the edge cannot start until the source node finishes. Functional errors often constitute a major part of the bugs in software [Beizer 1990; Li, Tan, Wang, Lu, Zhou and Zhai 2006]. In AD, the wrong sequencing of the operations including false activation or a non-execution of an intended operation can be performed by dropping or swapping the links between the action or activity nodes. The *operation mutation* operators are intended for seeding functional faults such as missing, wrong and unwanted functionality.

Definition:

Let IE and OE be the input and output edge set of an action node z respectively, $e(s,t)$ is a directed edge from the source node s to the target node t, $IE = \{e | \ e \in E \ and \ t_e = z\}$ and $OE = \{e | \ e \in E \ and \ s_e = z\}$.

The ***Missing Action Operator*** omits one action node z in the model for each mutant model.

Mutant models X_k, $0 \leq k \leq |Z|$, are generated in such a way that $t_{ie} \leftarrow t_{oe}$ for each z_i such that $z_i \in Z, ie \in IE$ and $oe \in OE$.

The ***Actions Exchanged Operator*** generates the error when the order or position of two action nodes in the models is exchanged. It changes the position or order of the action nodes in the model for each mutant model.

Mutant models $X_k, 0 \leq k \leq |Z|$, are generated in such a way that $t_{ie}^{z_i} \leftarrow t_{ie}^{y}$, $t_{ie}^{y} \leftarrow t_{ie}^{z_i}$, $s_{oe}^{z_i} \leftarrow s_{oe}^{y}$ and $s_{oe}^{y} \leftarrow s_{oe}^{z_i}$ for each z_i such that $z_i \in Z, \text{y} = \{z | z_{random} \in Z, z_{random} \neq z_i\}, ie \in IE$ and $oe \in OE$.

Interface Mutation Operator (IMO)

Most of the conventional interface errors are related with data or object flow when one or more data inputs are required for an operation. In AD, data or object flow can be depicted by object nodes and object flow edges. As object flow modeling is out of scope for this study, only control flow related interface fault patterns are addressed here. In AD, an executable node needs some inputs to start execution and produces some outputs at the end of the execution. It becomes active and ready for execution only when tokens on its all inputs are available. At the start of execution, the tokens are consumed (removed from the inputs) and an operation is performed on them. At the end of execution, tokens are offered on all outputs which in turn are available for consumption to the next node(s) in the sequence. The interface mutation operators inject faults that are related with the interaction between the artifacts of the model and result into non-activation or non-execution of the invoked artifact. This type of faults implies that the required input is missing or output is not being produced.

Definition:

Let IE and OE be the input and output edge set of an action node z respectively, $e(s,t)$ is a directed edge from the source node s to the target node t, $IE = \{e | \ e \in E \ and \ t_e = z\}$ and $OE = \{e | \ e \in E \ and \ s_e = z\}$.

According to AD semantics, more than one inflow into an executable node implies join behavior which means that it needs token available on all inputs to start execution. So the ***Extra Inflow Operator*** creates an extra input for an action node which will result in an unexpected non-execution of that node in the model for each mutant model.

Mutant models $X_k, 0 \leq k \leq |Z|$, are generated in such a way that $t_{oe}^{z_r} \leftarrow z_i$ for each z_i such that $z_i \in Z, z_r = \{a | a_{random} \in Z\}$, and $oe \in OE$.

More than one outflow of an action implies fork behavior and ***Extra Outflow Operator*** induces an unwanted invocation of multiple threads in the model for each mutant model.

Mutant models $X_k, 0 \leq k \leq |Z|$, are generated in such a way that $t_{oe}^{z_i} \leftarrow z_r$ for each z_i such that $z_i \in Z, z_r = \{z | z_{random} \in Z\}$, and $oe \in OE$.

The ***Inflow Exchanged Operator*** generates a wrong method call in the model for each mutant model.

Mutant models $X_k, 0 \leq k \leq |Z|$, are generated in such a way that $t_{ie}^{z_i} \leftarrow t_{ie}^{y}\ and\ t_{ie}^{y} \leftarrow t_{ie}^{z_i}$ for each z_i such that $z_i \in Z, y = \{z | z_{random} \in Z, z_{random} \neq z_i\}, ie \in IE$.

Concurrency Mutation Operator (CMO)

Concurrency is an important factor in the behavior of modern systems. As mentioned earlier, AD supports modeling both parallel and distributed concurrency mechanisms. Fork and join nodes are provided for the creation of processes or threads that can execute concurrently and to specify the synchronization between these concurrent processes or threads respectively. SendSignalAction and AcceptEventAction nodes are provided for specifying process communication through message passing. According to the AD semantics, the incoming tokens at a fork are duplicated and offered to all outputs. At least one input and two outputs are required for a Fork node. A Join node offers a token on the outgoing edge only after receiving tokens on all of its inputs. To facilitate the understanding of common concurrency failures, the following are the necessary definitions along with examples depicted in AD. For more detail, please refer to [Andrews 2000].

Deadlock is defined as a situation where two or more processes are unable to proceed because each is waiting for the other to complete or release some resources in a circular chain. For instance, in Figure 4-5 both Action1 and Action2 are in deadlock as Action1 is waiting for Object2 to be produced by Action2 and Action2 is waiting for Action1 to produce Object3. *Livelock* is similar to deadlock in that the program does not make any progress. However, in deadlocked computation there is no possible execution sequence which succeeds, whereas in a livelocked computation there are successful computations, but there are also one or more execution sequences in which no thread makes progress. For example, as shown in Figure 4-6 if Action1 execute before Action2 then both threads will complete execution, however if Action2 start execution before Action1 then thread1 will keep waiting for Action2 to release Object1 and gets stuck. *Race condition* is defined as a situation when two or more processes attempt to access a shared memory location concurrently and one of the accesses is a write then the output of the execution depends on the order in which the access takes place. For example, as shown in Figure 4-7, action A1 and A2 concurrently try to change the value of 'X' to '1' and '2' respectively

but as the execution order of A1, A2 and A3 is nondeterministic, the value of 'Y' at action A3 will be randomly determined.

Farchi et al. [2003], defined three concurrency related fault patterns and also identified various code level instances of these fault patterns. The suggested fault patterns were based on the common mistakes that a programmer may commit while coding. Accordingly, the first type of fault pattern is based on an assumption that threads interleaving are protected such that no thread executes a concurrent operation during the execution of another thread. The second fault pattern is based on the wrong assumption that certain interleaving will never occur (i.e. certain operations in different threads will not interleave) and no synchronization is required. The third fault pattern is based on the mistaken assumption that interleaving threads are non-blocking (e.g. one of the threads contains a blocking operation that blocks indefinitely or one of the threads terminates due to an exception when it is not expected to terminate). Concurrency mutation operators defined here are based on these concurrency patterns as described in [Farchi, Nir and Ur 2003] to model the concurrency faults in AD models. Given the overlap between the fault patterns [Farchi, Nir and Ur 2003], the "Missing Join Operator" will inject the first two types of faults whilst the "Invalid Synchronization Operator" will inject the third type of faults.

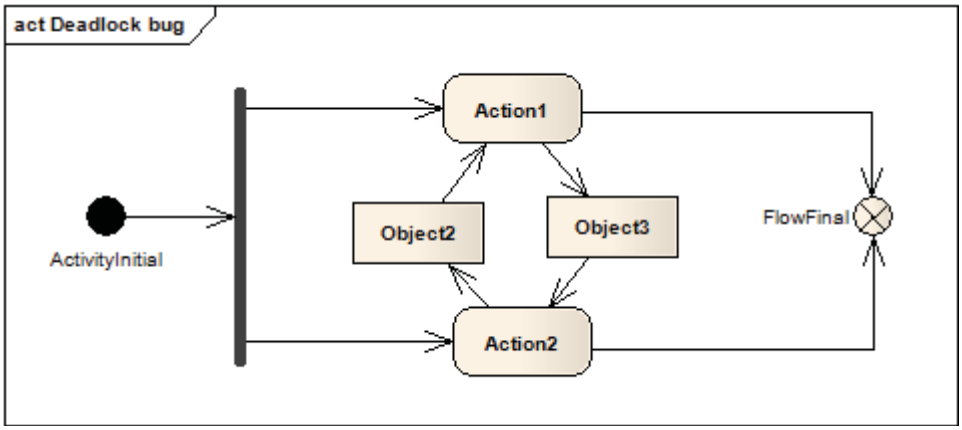


Figure 4-5: Deadlock bug, Action1 in tread1 is waiting for Object2 whereas Action2 in thread2 is waiting for Object3

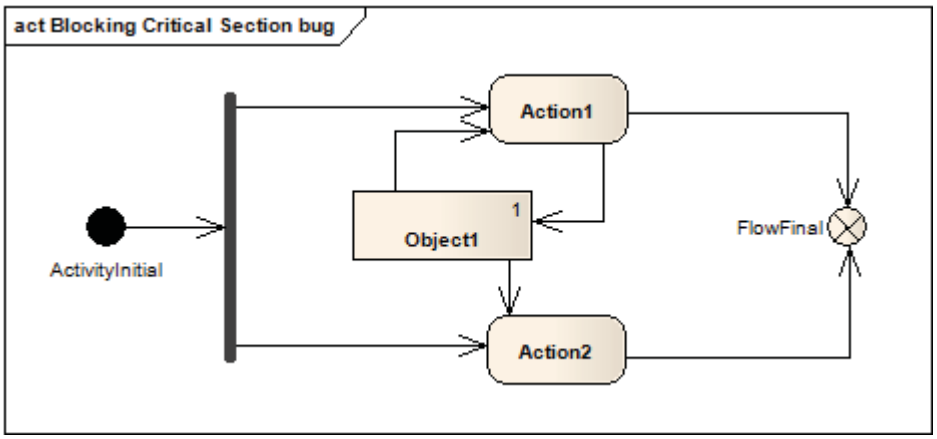


Figure 4-6: Livelock bug, thread-1 which contains Action1 gets blocked if Action2 executes before Action1.

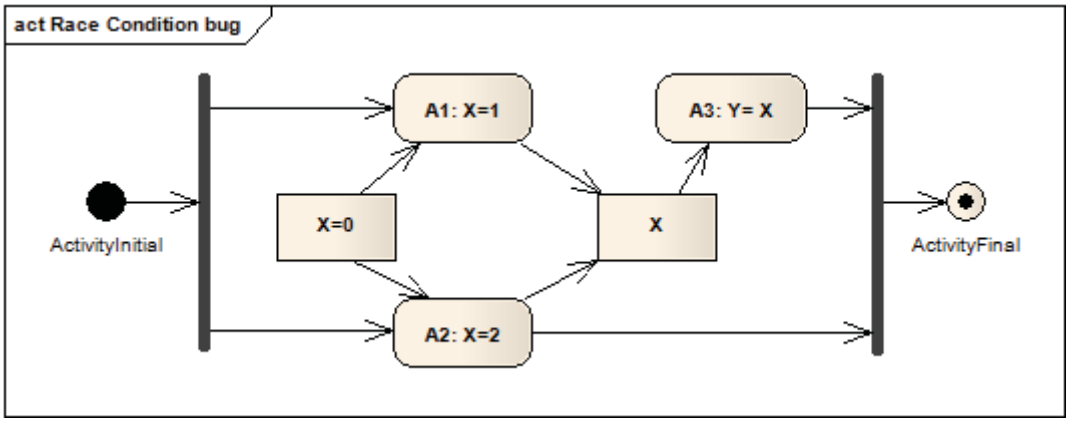


Figure 4-7: Race condition, the final value of Y is dependent on the execution order of the Actions A1, A2 and A3

Definition:

Let IE and OE be the input and output edge set of a join node j respectively, $e(s,t)$ is a directed edge from the source node s to the target node t, and $IE = \{e| \, e \in E \; and \; t_e = j\}$ and $OE = \{e| \, e \in E \; and \; s_e = j\}$.

Synchronization is used to prevent some undesirable interleaving. The ***Missing Join Operator*** models the missing synchronization fault (that may result into race conditions or atomicity violation) in the model for each mutant model.

Mutant models $X_k, 0 \leq k \leq |J|$, are generated in such a way that $t_{ie} \leftarrow t_{oe}$ for each j_i such that $j_i \in J, ie \in IE$ and $oe \in OE$.

The ***Invalid Synchronization Operator*** models the invalid synchronization (dead thread) fault in the model for each mutant model. For instance, Figure 4-8 and Figure 4-9 show two scenarios where deadlock may occur in at least one case. The operator manifest the "Blocking

or dead thread bug", a third type of concurrency fault pattern where some interleaving blocks indefinitely [Farchi, Nir and Ur 2003].

Mutant models $X_k, 0 \leq k \leq |J|$, are generated in such a way that $s_{ie} \leftarrow d$ and $t_{ie} \leftarrow j$ for each j_i such that $j_i \in J, d \in D$ and $ie \in IE$.

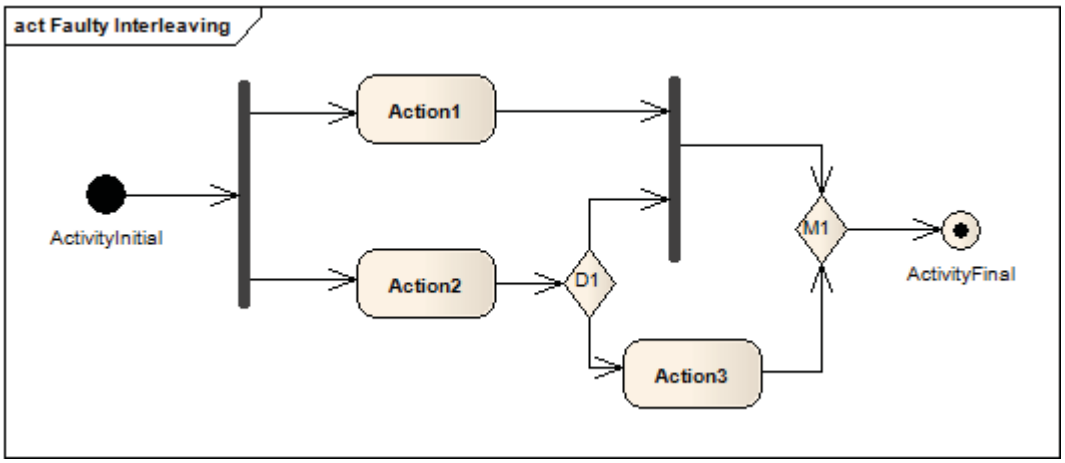


Figure 4-8: Faulty interleaving, execution of thread-1 get stuck in at least one scenario

Decision Mutation Operator (DMO)

Typically, a decision node in an AD has an incoming edge and multiple outgoing edges. Each outgoing edge (branch) from a decision node carries a guard condition that is evaluated at runtime to determine if the token can be offered on the edge or not. The decision mutation operators are intended for seeding branch or decision faults i.e. unreachable paths and missing paths.

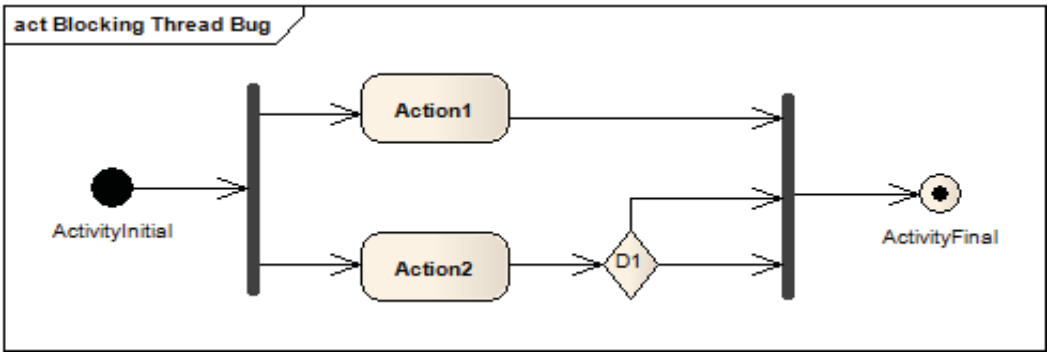


Figure 4-9: Blocking Thread Bug, according to AD semantic Join waits until tokens are available on all inputs

Definition (Extra Branch Operator): The operator injects an extra branch in the model for each mutant model.

Let IE and OE be the input and output edge set of a decision node d with cardinality n and m respectively, $e(s,t)$ is a directed edge from the source node s to the target node t, $IE = \{e|\ e \in E\ and\ t_e = d\}$, $OE = \{e|\ e \in E\ and\ s_e = d\}$, $n \in \mathbb{N}$ and $m \in \mathbb{N}$. Mutant models $X_k, 0 \leq k \leq |D|$, are generated in such a way that $s_{ie}(n+1) \leftarrow d_i$ and $s_{oe}(n+1) \leftarrow A_F$ for each d_i such that $d_i \in D, ie \in IE$ and $oe \in OE$.

Definition (Missing Branch Operator): The operator removes a branch from a decision node in the model for each mutant model.

Let OE be an output edge set of a decision node d, $e(s,t)$ is a directed edge from the source node s to the target node t, and $OE = \{e|\ e \in E\ and\ s_e = d\}$. Mutant models $X_k, 0 \leq k \leq |D|$, are generated in such a way that $s_{oe} \leftarrow t_{oe}$ for each d_i such that $d_i \in D$,and $oe \in OE$.

Definition (Missing Merge Operator): The operator models the missing merge fault in the model for each mutant model.

Let IE and OE be the input and output edge set of a merge node m respectively, $e(s,t)$ is a directed edge from the source node s to the target node t, and $IE = \{e|\ e \in E\ and\ t_e = m\}$ and $OE = \{e|\ e \in E\ and\ s_e = m\}$. Mutant models $X_k, 0 \leq k \leq |M|$, are generated in such a way that $t_{ie} \leftarrow t_{oe}$ for each m_i such that $m_i \in M, ie \in IE$ and $oe \in OE$.

Definition (Negation of Condition Operator): The negation of condition fault refers to a fault where a condition that is required to be true in the specification is changed to false or vice versa. The operator depicts the negation of condition fault in the model for each mutant model. A condition is replaced by its negation in the formula.

Let OE be an output edge set of a decision node d, c be a condition attached to an output edge, $e(s,t)$ is a directed edge from the source node s to the target node t, and $OE = \{e|\ e \in E\ and\ s_e = d\}$. Mutant models $X_k, 0 \leq k \leq |OE|$, are generated in such a way that $c \leftarrow \bar{c}$ for each $oe_i^{d_i}$ such that $d_i \in D$,and $oe \in OE$.

The operators developed and defined in this study for AD models are aimed at injecting faults of both the omission and commission types, where a missing action is an omission fault and an extra action is a commission type of fault. For the application of defined operators, it is assumed that the model has already been refined by replacing each implicit decision, fork and join with an explicit decision, fork and join respectively. The assumption is based on the fact that the replacement of an implicit decision, fork or join with explicit counterparts does not affect the control-flow but does reduce the ambiguity (i.e. multiple inputs or multiple flows) as shown is

Chapter 3. These mutation operators were subsequently implemented as XSL transformation rules and specified in a XML-based Stylesheet for experimental analysis. The code snipped of these mutation operators can be found in Appendix B.

Table 4-2: AD Mutation Operators

Operator Type	Mutation Operator
OMO	▪ Missing Action
	▪ Actions Exchanged
IMO	▪ Extra Inflow
	▪ Extra Outflow
	▪ Inflow (Input) Exchanged
DMO	▪ Missing Branch
	▪ Extra Branch
	▪ Missing Merge
	▪ Negation of Condition
CMO	▪ Missing Fork (Thread)
	▪ Missing Join (Synchronization)
	▪ Invalid Synchronization

4.2 Experimental Analysis

A number of experiments are conducted to examine AD-based technique that was developed for generating test sequences. Two categories of AD-based coverage criteria (sequential and concurrent) have been proposed for test selection and adequacy analysis. Test suites generated are evaluated to determine if a given testing criterion, chosen from the two categories of AD-based coverage criteria, is met. The effectiveness of these generated test suites will also be evaluated using the proposed AD-based mutation analysis technique. The following sections describe the experiments, result and analysis.

4.2.1 Experimental Setup

The four case studies of AD models, as described in Chapter 3 are used in the experiments. Firstly, using the proposed AD-based test sequence generation technique and the AD models in each of the case studies, were generated until each test suite achieved the

complete coverage with respect to the branch coverage criterion. The resulting test suites are analysed and are further evaluated against two concurrent coverage criteria.

Secondly, to evaluate the fault detection effectiveness of the generated test suite, the AD-based mutation analysis technique described in Section 4.1.2.3 was performed. Generally, mutants are generated by introducing *k* simultaneous changes in the original artifact and considered as k-order mutants. Earlier studies [Budd 1981] have indicated only a minor gain in the quality of an artifact with higher-order mutation analysis in comparison to the cost involved in mutant generation and execution. So the mutation analysis applied here in this experiment was limited to the first order mutants. Using the XSLT-based model transformation technique, the mutation rules were applied on the case study models (in XMI format) to produce mutant models. The number of mutants generated for case study models using each of the operators is shown in Table 4-3. The Negation of Condition operator is not used to produce any mutant models as the case study models do not contain the required parameters.

All the generated mutant models were executed with the test suite generated from the original AD model. A mutant model was marked as dead if it failed to execute any one of the test sequences in the given test suite. Due to the large number of mutant models and the repetitive nature of the mutation analysis tasks, manual mutation analysis was deemed infeasible and a tool was developed to generate and detect the mutant models. However, the undetected mutants were analyzed interactively to determine the equivalent mutants and the deficiency of the generated test suite.

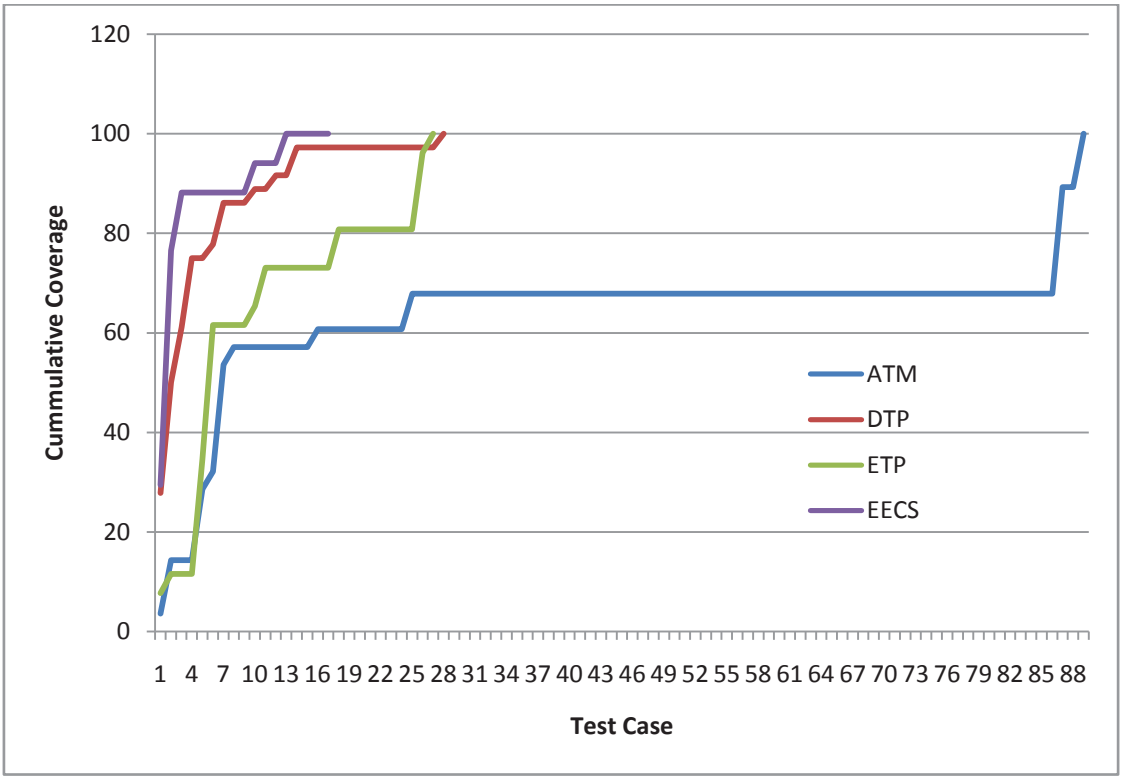


Figure 4-10: Coverage analysis of test suites generated for case study models with respect to branch coverage criterion

Table 4-3: Summary of mutants generated for each of the four case study models

	Number of Mutants Generated			
Mutation Operator	**ATM**	**ETP**	**DTP**	**ECCS**
Missing Action	65	72	60	9
Action Exchange	65	72	60	9
Extra Inflow	65	72	60	9
Extra Outflow	65	72	60	9
Inflow Exchanged	117	72	92	25
Extra Branch	13	12	18	7
Missing Merge	16	11	13	5
Missing Thread	11	0	0	2
Missing Synchronization	8	0	0	1
Invalid Join	8	0	0	1
Total	433	383	363	77

4.2.2 Result and Discussion

Results from the application of the AD-based test sequence generation technique on the case study models are shown in Figure 4-10. This figure illustrates the cumulative coverage of each of the test suites, calculated in the order as each of the test cases are generated using the corresponding case study models. The test adequacy of these test suites, evaluated against the branch coverage criterion and results are shown in Table 4-4. The results demonstrated that (1) the proposed test sequence generation process, involving the transformation of an AD into a CPN model, is a feasible approach and (2) under circumstances of no constraints on the number of test sequences, RW-based algorithm is capable of generating sufficient test sequences to satisfy a given criterion. The graphs in Figure 4-10 show the cumulative coverage of the test suites and reveal two important characteristics common to all test suites. First, the increase in cumulative coverage is discrete and uneven, as signified by the "jumps" in ETP graph. Second, the flat section in each graph (e.g. see ATM graph from test case 25 to 85) indicates the redundancy in the test suite.

In order to get a better insight of the redundancy issue, the test suites generated for ECCS, ETP and DTP models were further analysed and presented in Table 4-4. The column names, Cov and CC in the table are abbreviations for coverage and cumulative coverage respectively. The individual percentage coverage of each test sequence is presented in the column 'Cov'. The 'CC' column is the cumulative coverage gained by each additional test sequence. The evaluation results for ATM test suite can be found in Appendix B where it is presented separately due to the large size of the test suite.

A test case is considered redundant if it does not improve the cumulative coverage. The redundancy issue is further illustrated with the ECCS test suite. The first test case in ECCS test suite, TS-1 attained 29.4% coverage of the ECCS model according to the branch coverage criterion. As it was the first test case in the suite, all tested artifacts were unique and therefore the cumulative coverage was also 29.4%. After that, TS-2 attained 70.6% coverage and the cumulative coverage reached to 76.5% as it covered new artifacts in the model. With the execution of TS-3, the CC reaches to 88.2%. But for next six test cases, from TS-4 to TS-9, the CC does not improve as no new artifact has been executed. The CC further improves when TS-10 executed a new artifact. After that, the CC remains stagnant for TS-11 and TS-12. Finally, CC reaches to 100% when TS-13 executed the remaining artefact of the ECCS model. The next four test cases, from TS-14 to TS-17 are redundant as CC is already 100%, that is all relevant artefacts of the model has been executed at least once. It was found that 70% of test cases in the ECCS test suites were redundant. The ratio of redundant test cases in other test suites was also very

high. In case of ETP and DTP test suites, 67% and 64% test cases were redundant respectively. The proportion of redundant test cases in ATM test suite was highest (88%).

Table 4-4: Summary of the test suite generated for ECCS, ETP and DTP models

	ECCS		ETP		DTP	
Tests	**Cov %**	**CC %**	**Cov %**	**CC %**	**Cov %**	**CC %**
TS-1	29.41	29.40	7.69	7.69	27.78	27.78
TS-2	70.59	76.50	7.69	11.54	38.89	50.00
TS-3	64.71	88.20	7.69	11.54	30.56	61.11
TS-4	47.06	88.20	7.69	11.54	30.56	75.00
TS-5	41.18	88.20	26.92	34.62	22.22	75.00
TS-6	47.06	88.20	34.62	61.54	25.00	77.78
TS-7	35.29	88.20	7.69	61.54	41.67	86.11
TS-8	52.94	88.20	7.69	61.54	27.78	86.11
TS-9	41.18	88.20	26.92	61.54	19.44	86.11
TS-10	52.94	94.10	23.08	65.38	22.22	88.89
TS-11	58.82	94.10	38.46	73.08	36.11	88.89
TS-12	47.06	94.10	7.69	73.08	27.78	91.67
TS-13	58.82	100	23.08	73.08	19.44	91.67
TS-14	41.18	100	7.69	73.08	52.78	97.22
TS-15	47.06	100	7.69	73.08	44.44	97.22
TS-16	70.59	100	7.69	73.08	36.11	97.22
TS-17	52.94	100	7.69	73.08	16.67	97.22
TS-18	-	-	30.77	80.77	30.56	97.22
TS-19	-	-	7.69	80.77	27.78	97.22
TS-20	-	-	7.69	80.77	13.89	97.22
TS-21	-	-	7.69	80.77	36.11	97.22
TS-22	-	-	7.69	80.77	11.11	97.22
TS-23	-	-	7.69	80.77	19.44	97.22
TS-24	-	-	34.62	80.77	33.33	97.22
TS-25	-	-	7.69	80.77	47.22	97.22
TS-26	-	-	50.00	96.15	38.89	97.22
TS-27	-	-	46.15	100	22.22	97.22
TS-28	-	-	-	-	38.89	100

Although some test cases have a higher coverage with respect to a specific criterion than others, the cumulative coverage could still not be improved due to the execution order of test cases in the test suite. For instance, in Table 4-4, although the test case TS-11 of ECCS test suite had almost double the coverage of TS-1, it would be deemed redundant under the current test

execution order of the test cases. However, redundancy is a complex issue and simply changing the execution order of test cases might not be adequate due to the duplication and subsume relationship between the test cases. For example, test cases TS-16, TS-17 are duplicates of TS-2 and TS-8 respectively and changing their order will not resolve the problem. Nevertheless, it seems that finding an optimal combination of test cases rather than just an execution order according to some given criteria may be another promising option employed to optimize the test suite. In next chapter (Chapter 5), the problem of finding an optimal combination of test cases is reformulated as a combinatorial optimization problem and a number of heuristic-based solutions are also considered.

As the execution of the test sequence generation RW-algorithm conforms to CPN semantics, the generated test sequences are feasible for concurrency testing as well. The generated test suites were further evaluated against two of the concurrent criteria (Interleaving node coverage and interleaving edge coverage) defined in Section 4.1.2.2 for following two reasons.

1. To evaluate the adequacy of generated test suite according to the concurrent coverage criteria, and
2. To determine the redundancy in a test suite with respect to the concurrent coverage criteria.

The Table 4-5 presents the evaluation of ECCS test suite with interleaving node and interleaving edge coverage criteria. The column names, Cd, Cov, NC and CC in the table are abbreviations for covered nodes, coverage, newly covered nodes and cumulative coverage respectively. The numbers of paired interleaving artifacts are given in columns 'Cd' for a particular test sequence. The degree of the individual coverage of each test sequence is presented in the column 'Cov'. The 'NC' column contains the number of unique paired interleavings that were covered by a test sequence. The 'CC' column is the cumulative coverage gained by each additional test sequence. As two of the models (ETP and DTP) did not have the multithreading functionality, they were not used in the analysis for concurrency coverage criteria. The evaluation ATM test suite with concurrent criteria can be found in Appendix B.

In term of analysing concurrent coverage criteria for the ECCS model, executing the same ECCS test suite (all 17 test cases), complete coverage could not be achieved. The test suite achieved 50% cumulative coverage for the interleaving node coverage criterion and in the case of interleaving edge coverage criterion, it could only get up to 35.7% cumulative coverage. In addition, as seen in Table 4-4 and Table 4-5, there were a large number of redundant test cases according to both sequential and concurrent criteria (no improvement in terms of the

cumulative coverage). An important revelation for the concurrent coverage criteria is that TS-4 and TS-12 which appeared redundant according to the sequential criteria became effectively important test cases for concurrency testing. For instance, the test case TS-4 is redundant according to both the branch coverage and the interleaving node coverage criteria but as it covered a unique interleaving edge sequence so it was not redundant at least according to the interleaving edge coverage criterion. Similarly, although the test case TS-12 was found redundant with respect to the branch coverage criterion, it was not according to both the interleaving node and interleaving edge coverage criteria. It indicates that the redundancy of test cases in a test suite and effectiveness of test cases is relative to the test criteria in question.

The relative comparison of results for both sets of criteria revealed that the sequential criteria were relatively easier to achieve than the concurrent criteria. As can be seen from the data given in Table 4-4 and Table 4-5, although the generated test suite was adequate according to a sequential (branch) coverage criterion, it was not sufficient for concurrent testing and thus it required generating more test cases.

Table 4-5: Evaluation of the ECCS test suite with interleaving node and interleaving edge coverage criteria

	Interleaving Node Coverage				Interleaving Edge Coverage			
Tests	**Cd**	**Cov %**	**NC**	**CC %**	**Cd**	**Cov %**	**NC**	**CC %**
TS-1	2	14.3	2	14.3	2	7.1	2	7.14
TS-2	1	7.14	1	21.4	1	3.6	1	10.7
TS-3	2	14.3	1	28.6	2	7.1	2	17.9
TS-4	2	14.3	0	28.6	2	7.1	1	21.4
TS-5	1	7.14	0	28.6	1	3.6	0	21.4
TS-6	2	14.3	0	28.6	2	7.1	0	21.4
TS-7	1	7.14	0	28.6	1	3.6	0	21.4
TS-8	2	14.3	0	28.6	2	7.1	0	21.4
TS-9	1	7.14	0	28.6	1	3.6	0	21.4
TS-10	2	14.3	2	42.9	2	7.1	2	28.6
TS-11	1	7.14	0	42.9	1	3.6	0	28.6
TS-12	2	14.3	1	50	3	11	2	35.7
TS-13	2	14.3	0	50	2	7.1	0	35.7
TS-14	1	7.14	0	50	1	3.6	0	35.7
TS-15	1	7.14	0	50	1	3.6	0	35.7
TS-16	1	7.14	0	50	1	3.6	0	35.7
TS-17	1	7.14	0	50	1	3.6	0	35.7

The basic idea behind using a RW-algorithm is to enumerate all the possible and unique control flow paths in a model. With random interactions in the concurrent processes, the number of permutated paths grows exponentially and manual or exhaustive test sequences generation techniques (e.g. depth first algorithm) are therefore infeasible. Consequently, the RW-based TSG algorithm is deemed adequate because of the probabilistic nature of the algorithm. The algorithm incrementally generates more test sequences and stops once a specified coverage criterion is achieved. As the execution of the algorithm conforms to the CPN semantics, every test case generated with the RW-based technique is essentially feasible. However, the proposed algorithm is not ideal as it has a tendency to produce a number of redundant test cases due to the stochastic nature of the algorithm. One option to address this problem is to modify the test sequence generation algorithm to produce an optimual test suite. Another option is to find the optimal subset from the test suite obtained from the RW-algorithm.

Table 4-6: Synthesis of mutation analysis of test suites generated for case study models

Mutation Operator	ATM		ETP		DTP		ECCS	
	Killed	Alive	Killed	Alive	Killed	Alive	Killed	Alive
Missing Action	65	0	72	0	60	0	9	0
Action Exchange	65	0	72	0	60	0	9	0
Extra Inflow	65	0	72	0	60	0	9	0
Extra Outflow	46	19	72	0	60	0	6	3
Inflow Exchanged	117	0	72	0	92	0	25	0
Extra Branch	13	0	12	0	18	0	7	0
Missing Merge	16	0	11	0	13	0	5	0
Missing Thread	11	0	0	0	0	0	2	0
Missing Synchronization	8	0	0	0	0	0	1	0
Invalid Join	8	0	0	0	0	0	1	0
Total	414	19	383	0	363	0	74	3

In terms of the experiment using mutation analysis for determining the fault detection effectiveness of the generated test suites, the first step produced a number of mutants for each of the case studies using the defined mutation operators and this is shown in Table 4-3. The number of both killed and live mutants for each model using the associated test suite is shown in Table 4-6. As can be seen from the table, all the mutants for ETP and DTP models were killed and

the generated test suite for these two models was adequate. However, some mutants could not be killed for the ECCS (3 out of 77) and the ATM (19 out of 433) models. In both cases, the live mutants of both the ECCS and ATM are in the category of "extra outflow". The review of these live mutants revealed that they were not equivalent mutants and the failure to kill them was due to the inadequacy of the associated test suite. For instance, the Figure 4-11 and Figure 4-12 show the selected part of the two "extra outflow" mutant of the ECCS model that could not be killed with the generated test suite. In both cases, an extra outflow from an *action* node (i.e. logon and select) is ending on a *fork* node (i.e. F1 and F2) which means an extra token created on the *action* nodes (due to the intrinsic fork) is available to the *fork* nodes. However, that extra token could not be moved forward due to the intrinsic join behaviour of the *fork* nodes and no unique path was created. Therefore, despite the fact that both mutant models are not equivalent to the ECCS model, they could not be differentiated from the ECCS model by the generated test suite.

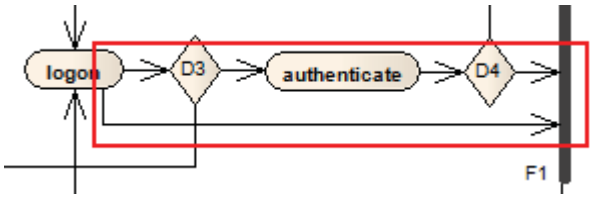


Figure 4-11: ECCS model - Extra Outflow live mutant

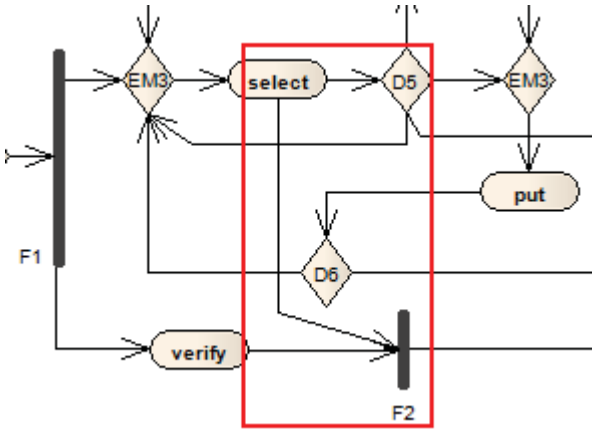


Figure 4-12: ECCS model - Extra Outflow live mutant

4.3 Summary

In this chapter an AD-based test case generation technique and two test suite evaluation techniques were introduced. In order to generate a test suite from an AD model, it was transformed into an executable CPN model and then a stochastic algorithm is used to generate

test sequences. The proposed test case generation techniques free a tester from learning a new language and tool or redesigning his already built models in order to execute them. Two sets of coverage criteria to evaluate the adequacy of a generated test suite were proposed. Mutation analysis technique was also proposed to evaluate the effectiveness of the AD-based test suite. The proposed test sequence generation and evaluation techniques were empirically analysed and results confirmed the effectiveness of proposed techniques.

Chapter 5 Test Suite Optimization

The previous chapter described a stochastic test generation technique for behavioral models depicted in AD. It also addressed the issue of adequacy criteria for a generated test suite and proposed coverage and mutation based analysis for AD based test suites. This chapter described an investigation to examine how metaheuristic techniques may be used to optimize the generated test suites obtained using techniques described in Chapter 4.

One of the desired characteristics associated with effective test cases is that they are unique and non-redundant in order to avoid wastage of resources and time. Therefore, in order to maximize the productivity of the testing effort (such as fault detection capability, coverage or reliability level) several test suite optimization approaches [Jones, Eyres and Sthamer 1996; Jones, Eyres and Sthamer 1998; Michael, McGraw and Schatz 2001; Pargas, Harrold and Peck 1999; Tracey, Clark and Mander 1998; Tracey, Clark, Mander and McDermid 1998] that incorporate an optimization technique in the test suite generation and maintenance process have been proposed. These approaches can be classified into in-test generation and post-test generation optimizing techniques. In the former, the test generation problem is reformulated as a searchable problem and an optimization technique is applied to generate an optimal test suite [Shiba, Tsuchiya and Kikuno 2004; Sthamer 1996; Wegener, Baresel and Sthamer 2001], whereas in the latter, the optimization of a pre-generated test suite is defined as an optimization problem followed by the use of a heuristic technique to derive an optimal version of the original test suite. Both these approaches have their specific limitations. The in-test generation optimization can generate a small and effective test suite according to given test criterion but does not permit retrospective removal of redundant test cases once new and better test cases have been generated. This drawback means that the test suite is virtually locked and more likely to contain some level of unwanted redundancy. On the other hand, with post-generation optimization the efficiency may be enhanced by removing the redundancy, but it cannot improve the coverage level and fault detection capability. However regardless of the differences, several studies [Jones, Eyres and Sthamer 1998; Li, Harman and Hierons 2007; McMinn, Harman, Binkley and Tonella 2006; Wegener, Eyres, Sthamer and Jones 1997; Yoo and Harman 2007] has shown that

incorporating optimization techniques in testing can significantly enhance productivity and cost effectiveness without compromising the fault detection capability.

5.1 Test Suite Minimization

Some studies have shown that more testing can improve the quality of software initially; however, the increase in testing eventually reaches a point of diminishing returns where it no longer translates into better quality. It has been often emphasized that a 100% statement coverage or that of any other test criterion does not necessarily guarantee a 100% defect free software [Beizer 1990; Williams et al. 2001]. Therefore, a great deal of research has focused on developing new test optimization techniques for enhancing software efficiency and effectiveness. Test suite optimization or post-test generation optimization is a class of techniques that aims to minimize the cost of a test suite without compromising its fault detection capability. It is achieved by removing test cases that are considered redundant or ineffective with respect to the test objectives for which they were generated. The process of identification and removal of redundant test cases, finally yielding a minimal test suite, can be defined as test suite minimization.

A test case is considered redundant if it does not add value to the test suite. A test suite produced for a given test criterion (i.e. statement or branch) requires the test requirements (parts of an artifact under test) according to that criterion must be satisfied. Usually, a test case fulfills more than one test requirements (e.g. statements or branches). In the case when some test cases are subset of other test cases, many test requirements are likely to be satisfied by more than one test case.

Test suites may also degrade and become less efficient over the lifetime of software as changes in the software or specification may render some of the test cases redundant or obsolete. Whilst rework or maintenance of software may require additional test cases for new or modified functionality or for feature interaction, it also requires identification and isolations of obsolete or redundant test cases. In practice, software passes through several revisions, each with many build and retesting cycles prior to its release. Regression testing is a very expensive activity as it is extremely time consuming to rerun previous test cases in order to ensure that changes in the software did not introduce any new faults. Studies [Chen and Lau 2001; Wong, Horgan, Mathur and Pasquini 1997] have confirmed that running the minimal subset of a test suite with the same coverage as the original test suite can reduce testing cost with very little or no effect on its fault detection capability.

Table 5-1: Example test suite with coverage illustration

	A	B	C	D	E	F	G
t_1	X		X	X	X		
t_2	X	X		X	X		X
t_3	X		X			X	
t_4	X	X	X	X		X	
t_5	X	X				X	X
t_6	X		X		X	X	
t_7	X	X	X	X		X	

As the redundancy of a test case is relative to and dependent upon the test criteria as well as the other test cases in a test suite, the optimization process may need to evaluate all possible combinations of the test cases in a test suite together with a calculation of their cumulative coverage. The following example provides an explanation of this point. Given a test suite TS with test cases $\{t_1, t_2, \dots, t_7\}$ and a software artifact under test with a set of seven branch elements $\{A, B, \dots, G\}$ as shown in Table 5-1. The execution of an element with a particular test case is indicated by the symbol 'X' . A careful analysis of this test suite reveals that redundancy often can exist in three forms: (1) a test case duplicates one or more test cases; (2) a test case subsumes the coverage of one or more test cases and (3) a combination of test cases subsumes the coverage of one or more test cases. For example in Table 5-1, t_7 is an exact duplication of t_4, a type-1 redundancy, as both test cases cover branch elements A, B, C, D, F; t_7 subsumes both t_3 and t_4 (a type-2 redundancy), and the combination of t_1 and t_5 subsumes the coverage of the rest of the test cases (a type-3 redundancy). The detection and removal of type-1 and type-2 redundancies from a test suite can be done simply by monitoring the cumulative coverage and dropping each additional test case, if it fails to improve the overall coverage level. For example, after selecting the first three test cases (t_1, t_2 and t_3) the remainder of the test cases will become redundant and may be dropped from the test suite. However, using this technique, type-3 redundancy is relatively harder to detect as the execution order of test cases can affect the selection of the subsequent test cases. Therefore, in order to detect a type-3 redundancy, an optimization technique needs to be employed for searching and

evaluating all possible combinations of test executions. Removing type-1 and type-2 redundancies can reduce the test suite size, but cannot ensure the minimal test suite. The removal of a type-3 redundancy also eliminates the need for removing the other two types of redundancy to yield the minimal test suite.

Redundancy in test suites is generally not desirable as it wastes project resources and increases the testing cost. Various researchers have tried to address this problem. Harrold, Gupta and Soffa [1993] proposed a code-based, heuristic technique for removing obsolete and redundant test cases from an original test suite resulting in a reduced test suite. In their approach, the first step involved examining each test requirement that is covered by only one test case and selecting each of these test cases; then iteratively select those test cases that cover the maximum numbers of requirements until all test requirements are covered. In cases of a tie involving multiple test cases with the same coverage level then select the test cases that would cover the higher number of unmarked requirements. Finally, remove the rest of the test cases that become redundant as they do not uniquely cover the test requirement. Their proposed technique is a greedy algorithm and prone to produce suboptimal solutions. Moreover, Jeffrey and Gupta [Jeffrey and Gupta 2005] showed that the test suite reduction technique proposed by Harrold, Gupta and Soffa may compromise its fault detection capability and they proposed a new technique for test suite minimization which retained some level of redundancy in the test suite. They suggested using a secondary criterion (e.g. all def-use criterion associated data flow testing), in the test suite minimization process. They showed that with selective redundancy it is possible to retain the test suite effectiveness with a slightly less test suite reduction. Tallam and Gupta [2005] proposed an adapted greedy algorithm to minimize a test suite by removing redundant test cases. They employed the *Concept Analysis* technique to identify groups of objects and attributes and their implications and then exploit this information in a greedy algorithm for test suite reduction.

Xie, Marinov and Notkin [2004] showed that existing unit testing tools such as JTest and JCrasher, generate a large number of redundant test cases and developed a framework to minimize these generated test suites. They showed that the elimination of redundant test cases can be achieved without compromising their quality. They further proposed a number of redundancy detection approaches for detecting redundancy in test suites associated with an object oriented system.

Chen and Lau [1998; 2001; 2003] investigated Boolean specification-based test suite reduction techniques based on some fault-based test case selection criteria. They proposed a divide-and-conquer algorithm for test suite minimization which actually decomposes the original

problem into k independent sub-problems [Chen and Lau 2003]. Accordingly, the set of test cases is decomposed into $k \geq 2$ mutually disjoint subsets $T_1, T_2, \ldots, T_k$ of test cases such that the sets of all test requirements R_i satisfied by T_i are also mutually disjoint. Although, this divide-and-conquer approach is based on an exact algorithm that guarantees the delivery of an optimal solution from the set of optimal solutions of the sub-problems, it is generally considered not feasible for real world applications due to their sheer size or non-decomposable nature.

In the following section, a model-bases test suite minimization technique developed in this study is described.

5.2 Model-based Test Suite Minimization

Model based testing uses a model as a reference point for test case generation and evaluation purposes, which makes it independent of the implementation language and the need to examine the source code. Typically, a model consists of a set of abstract elements known as nodes and a relation between the nodes is referred to as an edge. The nodes represent the statements in the program and edges express the flow of control or data between the statements. Nodes with two or more outgoing edges are called decision nodes and the edges are referred to as branches. As a model based test suite is directly generated from a model according to a model-specific criterion, the model based test suite minimization problem can be defined as follows:

A model based test suite is given in the form of set TS with elements a_i, size n and coverage m. The set of elements a_i are the test cases where each test case is a sequence consisting of the model elements representing an execution path in the model. The coverage m is calculated as the percentage of model elements, required by a test criterion, that have been covered by a given test suite with n number of test cases. The objective is to find a minimal subset $ts = \{a_1, a_2, \ldots, a_k\}$ in such a way that $m_{ts} = m \; and \; k \leq n$.

5.2.1 Test Suite Evaluation

In order to analyze and compare the different test suite solutions objectively, the attributes of a test suite such as cost, fault detection capability, size and coverage must be measured. As the objective of test suite minimization is enhancing the efficiency of a test suite without compromising its effectiveness, it is important to know the size of the test suite that is produced, the cost to execute it, and the number of faults it can detect. It is postulated that a test suite is inefficient if it has a high degree of redundancy (as it will waste resources) and ineffective if it has “gaps” in the coverage of the given test requirement (as it will leave untested functionality in the system). Generally, the adequacy of a test suite is measured with respect to a

particular test requirement metric e.g. structural coverage, fault coverage and mutation score. Therefore, a test case in a test suite is considered redundant in accordance with a specific criterion:

1) If it fails to improve the consolidated coverage of the test suite [Shiba, Tsuchiya and Kikuno 2004; Sthamer 1996; Wegener, Baresel and Sthamer 2001] or

2) If removing it does not affect the effectiveness of the test suite [Harrold, Gupta and Soffa 1993; Jorgensen 2002; Offutt, Pan and Voas 1995].

The first rationale is the basis of in-test generation optimization whereas the second rationale is considered for post-test generation optimization. The evaluation of a test suite in terms of its ability to fulfill a given test selection or adequacy criterion indicate its degree of effectiveness. A test case is considered essential according to a given criterion if its inclusion or exclusion from the test suite can change the effectiveness of the test suite. The selection of test cases can be guided by various objectives, for instance, structural coverage (e.g. block and decision) to ensure the adequacy of a test suite [Frankl and Iakounenko 1998; Frankl and Weiss 1993; Hutchins et al. 1994], and functional coverage to show conformity to the specification [Beizer 1995].

The efficiency of a test suite can be defined as the cost it needs to achieve a given test criterion per test case used. The cost of the test suite, computed as the sum of the cost of its test cases, can be measured in several ways. One measurement uses the computing time it needs to execute each test case [Wagner 2006]. Another measures the tester time spent on constructing and analyzing test cases [Ellims et al. 2006]. The cost of a test suite to achieve the stated test objective indicates its efficiency in relation to the number of test cases needed to achieve the given test criterion. Such measurements have been used in other studies [Ntafos 1988; Weyuker 1990]. Based on this efficiency measurement, a test suite minimization procedure can find a minimal subset that maintains the coverage of the original test suite with respect to a certain coverage criterion.

As the objective of test suite minimization is enhancing the efficiency and effectiveness of the test suite, these two terms are defined in the context of this study as measurable attributes of a test suite as per the following: consider a model-based test suite TS that contains m test cases, with each test case being a sequence of model elements representing an execution path in the model. These test cases are evaluated with respect to the coverage criterion M that identifies n elements in the artifact under test (AUT). When a subset SS of TS containing x test

case is executed, it traverses p of the n AUT elements and attains p/n percent coverage. In order to quantify the efficiency and efficacy of a test suite, the following definitions are stated.

Definition 1:

The efficacy of a test suite TS with respect to the criterion M is a ratio of p to n.

Definition 2:

The efficiency of a test suite TS with respect to the criterion M is a ratio of x to m. In order to have the efficiency increase as with the decrease in subset, the efficiency metric is formulated as follows: $efficiency = 1 - (x/m)$

The efficacy metric would help to determine the extent to which the test suite satisfies a particular test criterion. The efficacy metric in definition-1 can be used to measure the effectiveness of a test suite in terms of selected criteria i.e. structural coverage and mutation score. When this value is less than 1 (in terms of ratio a.k.a. test effectiveness ratio [Woodward et al. 1980]) or 100%, it indicates the deficiency in the test suite (or inadequacy of the test suite) and the need to generate more test cases. The efficiency metric would help to determine that how economically the test suite satisfies a particular test criterion. The efficiency metric is obvious and a high score indicates a relatively more efficient test suite. For example in Table 5-1, the test suite contains 7 test cases and if executed as it is, it will yield an efficacy = 1 (complete coverage) and efficiency = 0. Executing the subset $TS_{sub} = \{t_1, t_5\}$ of TS and skipping the rest of the test cases can improve the efficiency from 0 to 0.71 without compromising its efficacy.

The efficiency metric allows determining the improvement in the reduced test suite but it does not provide any information about the level of redundancy in the test suite. In Section 5.1, it is shown that the redundancy in a test suite can occur when multiple test cases are covering completely or partially the same set of element(s) in an AUT. The coverage relationship between the test cases and elements according to a given criterion can be defined as the set $\{(t, e) \in TS \times E : t\ covers\ e\}$ where TS and E are the test suite with m test cases and a set of n elements in AUT respectively. Suppose $Elem(t)$ is the set of all elements that are covered by a test case t and $Test(e)$ is the set of all test cases that cover an element e. The redundancy metric can be defined as follows.

Definition 3:

The redundancy in a test suite according to a criterion M is an average of number of test cases per element (ATCE) and formally defined as:

$$ATCE = \frac{\sum_{i=1}^{n} |Test(e_n)|}{n} \qquad (5\text{-}1)$$

The process of manual identification and removal of the redundant test cases for large test suites is both overwhelmingly complex and infeasible. Similarly, exhaustive analysis even with an automated tool would handle only relatively trivial test suites and is prone to a scalability issue. In order to solve the test suite minimization problem using heuristic techniques, it is formulated as a combinatorial optimization problem in the following.

5.2.2 Formulation as an Equality Knapsack Problem

The knapsack problem is a class of combinatorial optimization problems that has been extensively studied. In the basic version, the knapsack has some specific capacity and a set of objects, each with a given weight and profit. The problem is defined as the search for a set of objects, such that the total profit of the set is maximized without exceeding the knapsack capacity. There are many knapsack variants including the Equality Knapsack problem (EKP) where the objective is to find a subset from a given set of items in such a way that the total profit is maximized and the total weight c is exactly equal to the given capacity C [Kellerer et al. 2004].

The test suite minimization problem can be translated into EKP. For instance, the test suite has *n* test cases that correspond to objects in the knapsack problem and the coverage *C* of the test suite corresponds to knapsack capacity. Each test case *i* has coverage c_i that corresponds to the weight of an object. In order to show the inclusion or exclusion of a particular test case, a binary decision variable *x* is used. The requirement to be satisfied is

$$\sum_{i=1}^{n} c_i x_i = C \qquad (5\text{-}2)$$

$$where\ x_i \in \{0,1\}, i = 1, \dots, n.$$

The utility value of a test case in the test suite that corresponds to cost in the knapsack problem is $u = 1$ for each test case. The objective is to find a test suite at a given coverage in such a way that the total cost of the test suite is minimized. Since the capacity of the test suite is *C*, it is required that the total weight of all chosen test cases (the total weight of the minimized test suite) is $f(x) = \sum c_i x_i$ to be *C* exactly. As the EKP can be formulated into a minimization

version by minimizing the cost of the items in the knapsack, the problem can formally be stated as

$$minimize \sum_{i=1}^{n} u_i x_i \qquad (5\text{-}3)$$

5.3 Empirical Study

As it is described in Section 5.2.1, removing the redundant test cases from a test suite can enhance the efficiency of the suite without compromising its effectiveness. However, identifying and removing the redundant test cases is a combinatorial optimization problem. Since it is unlikely that such problems are polynomially solvable, heuristic techniques are commonly used for finding approximate solutions in polynomial time. However, heuristics do not work equally effectively on all problem instances. A commonly held view is that there is a link between problem complexity (both size and the search space) and the performance of the heuristics [Back, Fogel and Michalewicz 1997]. Consequently, following experiments were conducted in order to investigate the comparative performance of the proposed EC-based test suite minimization with three other types of algorithms (Greedy, Hill Climbing Random Ascent and Hill Climbing Next Ascent) for different problem instances.

5.3.1 Research Questions

The experiment is designed with the aim of answering the following research questions based on the assumption that the efficiency of a test suite can be enhanced by removing redundancy in it.

Q1: How do the EC, Greedy, Hill Climbing Random Ascent and Hill Climbing Next Ascent algorithms perform in the context of model-based test suite minimization?

Q2: How does the test suite size affect its minimization through a specific heuristic?

Q3: How does the composition of a test suite (i.e. arrangement of test cases, length of the sequences) affect its minimization through a specific heuristic?

5.3.2 Experimental Setup

In order to investigate the research questions, the following variables were considered. The goal was to empirically assess the heuristic techniques and the effect of test suite size and composition on test suite minimization. The independent variables were test suite size, test suite composition, optimization algorithms and model type. The dependent variables were reduced-

size and coverage. It was expected that the test suites will be homogenous with respect to the source models. Thus, the test suites were blocked into four uniform groups on the basis of the models, namely ECCS, ETP, DTP and ATM (previously described in Chapter 3). Within each of the four blocks, experiments were performed with a 3 x 5 x 4 (three test suites of different sizes, five test suite compositions and four optimization algorithms) Factorial Repeated Measure with Block design [NIST/SEMATECH 2010]. Each factor is elaborated in the following sections.

In order to study the effect of size and composition of a test suite on test suite reduction, multiple versions of a test suite of varying size were generated for each model. Following the random walk based test generation (RW-TSG) technique described in Chapter 4, a test suite was generated from each case study model described in Chapter 3 and then evaluated according to the associated branch coverage criterion proposed in Chapter 4. A test suite generated for branch coverage criterion requires at least one test case to execute each branch. Two more test suites were generated with 25% and 50% more test cases than the original test suite.

Table 5-2 summarizes the composition of every generated test suite. For instance, approximately half of all branches on average are executed by each test case for the ECCS model as shown in the 'Mean' column of the Branch Coverage per Test Case (BCTC) column. The standard deviation 'S.D.' column indicates the variation of test cases in terms of their coverage. The 'Min' and 'Max' columns show the minimum BCTC (indicating smallest test sequence) and maximum BCTC (indicating longest test sequence) respectively. Although, with larger models (e.g. ATM) the average BCTC is smaller, the size of longest test sequences ('Max' column) is same for all three versions of test suites within a group (model) and the average difference within the group is also statistically insignificant at the .05 significance level. The redundancy level of the generated test suites with respect to branch coverage criterion is determined using redundancy metric (equation (5-1)) defined in Section 5.2.1 and presented in the column 'Test Cases per Branch' of Table 5-2. On average in each generated test suite, there are more than four test cases for a branch that can execute it.

In order to see the impact of test cases arrangement[3] on test suite minimization, each generated test suite was randomly shuffled five times and marked as a distinct version (TS-1, TS-2, TS-3, TS-4 and TS-5). Studies have shown that the performance of heuristic algorithms can vary depending on the structure of the fitness landscape [Deb 1997; Mitchell et al. 1992;

[3] The arrangement of test cases in a test suite implies their execution order. As the focus here is to study its impact on test suite minimization, the term 'arrangement' is used in order to avoid confusion with 'execution order' in test case prioritization.

Rothlauf 2006]. Other studies have shown a relationship between the smoothness or ruggedness of the search space and the correlation level between the fitness values of the search points [Jones 1995; Weinberger 1990]. Thus the assumption is that a change in the arrangement of the test cases in a test suite will change the structure of the search space and the complexity of the problem and may thereby affect the optimization of a test suite in general or the performance of a heuristic specifically.

Table 5-2: Composition of generated test suites

Model	Test Suite Size	Branch Coverage per Test Case (%)				Test Cases per Branch			
		Mean	S.D.	Min	Max	Mean	S.D.	Min	Max
ECCS	20	49.12	13.40	29.41	70.59	9.82	5.30	2	20
	25	49.88	12.49	29.41	70.59	12.47	6.69	3	25
	30	50.98	11.39	29.41	70.59	15.29	8.59	3	30
ATM	89	10.88	8.16	3.57	32.14	9.68	11.47	1	45
	111	11.65	8.32	3.57	32.14	12.93	14.35	1	60
	133	11.36	8.21	3.57	32.14	15.11	17.13	1	70
ETP	27	17.24	13.80	7.69	50.00	4.66	4.31	1	17
	34	15.84	12.94	7.69	50.00	5.39	5.61	1	23
	41	16.89	13.18	7.69	50.00	6.92	6.55	1	26
DTP	28	33.14	9.91	16.67	52.78	9.28	6.80	2	28
	35	32.22	9.88	16.67	52.78	11.28	8.54	3	35
	42	32.61	9.57	16.67	52.78	13.69	10.25	3	42

The experiments were performed with four heuristic algorithms namely; Greedy, Hill Climbing Random Ascent, Hill Climbing Next Ascent and Evolutionary Computation, referred to as GD, HCRA, HCNA and EC respectively forthwith. As mentioned earlier, normally different heuristic algorithms comprise of several different combination of operators and their application essentially involves several design considerations such as problem representation, solution selection and production operators, formulation of a fitness function to evaluate the quality of these solutions in guiding the underlying heuristic search and finally the selection of appropriate operating parameters for the optimal performance of the algorithm.

EC algorithms do not usually perform operation directly in the problem space. Therefore, in terms of their implementation the first and foremost step is encoding the problem in a suitable representation (e.g. binary string, real value or tree structure). The encoding scheme defines the search space and links the genotype or genome to a corresponding phenotype. The effect of encoding is very critical as the entire set of algorithmic operations is performed only on the encoded space. Similar to the knapsack-problem, binary encoding was

considered to be a direct and natural representation for the test suite optimization. Test suites are directly encoded in the form of genotype with a constant chromosome length of original test suite size. Each allele at a particular position in a chromosome represents a corresponding test sequence in the original test suite. The inclusion and exclusion of a test sequence within a test suite are represented by 1 and 0 respectively in a chromosome string (binary sequence). So, a chromosome with a given sequence of 0 and 1 represents a test suite with a particular combination of test sequences and the total number of 1's in a binary sequence represents the size of the test suite. A chromosome with a sequence of all 1's represents the original test suite.

The size and coverage of an individual test suite are some of the quality attributes pertinent to this study so these attributes can be expressed as phenotypic properties of an associated genotype and are calculated as the number and collective coverage of all the included test cases respectively. The fitness function calculates the quality of individuals in terms of phenotype in a generation. The phenotype which is the functionality or expression of the genotype maps the search space (encoded in the form of genotype) to the objective space. The notions of superior and inferior solutions and a fitness measuring mechanism have pivotal roles in evolutionary optimization as they guide the underlying search mechanism and greatly affect the convergence of the evolutionary heuristic [Corne, Glover and Dorigo 1999]. As the objective of test suite minimization is finding a minimal combination of test cases from the original test suite, equation-(5-3) is used to evaluate the fitness of each candidate solution. A precondition of the equation-(5-3) requires that a valid candidate solution must satisfy equation-(5-2).

The design of selection and production functions is crucial to the adaptation of the EC. The selection function defines rules for the selection of sub-population (mating pool) for the production of offspring. Various selection techniques i.e. rank, probabilistic, fitness-proportionate and tournament selection are widely used in EC applications. Tournament selection was used owing to its robustness and pressure controllability [Goldberg 1989]. The replacement mechanism defines the placement of offspring into the population and for that it is used as a generational technique. Accordingly, the offspring replaces the parents in next generation. The role of production operators is very critical for solution quality. The crossover process exploits the available fitness information and the mutation process leads to the exploration of the search space. The production function to breed new individuals comprises both recombination and mutation operators. Typically, the recombination operation can be either sexual or asexual. The sexual reproduction a.k.a. crossover produces new offspring from the parents. The individuals selected according to their fitness for mating survive through the generations and propagate their characteristics in the offspring. Therefore, through crossover,

the search converges towards the promising regions of the search space. The mutation operation introduces noise and prevents the premature convergence of the search process to local optima by randomly sampling new points in the search space. In bit string representation, mutation is applied by inverting bits at random in a string with a certain probability called the mutation rate. The mutation rate defines the number of bits that will be flipped at each iteration step. Similarly, the crossover mechanism essentially breeds new solutions by swapping the substrings of existing solutions (test suites) at each iteration step. Double-point operators are used for crossover and single-point operators are used for mutation.

For EC, the binary coding scheme, pairwise tournament selection, objective function (equation-(5-3)) value as fitness, double-point crossover, single-point mutation and maximum generation numbers as stopping criterion, are used in the experiment. Although EC is a generic technique, a relatively large number of operational parameters in addition to design parameters (i.e. selection, crossover and mutation operator types) still need to be configured.

The appropriate values for these parameters except for crossover are empirically determined for each problem instance and described as follows.

For crossover, De Jong [1991] suggested an optimal rate between 0.6 and 0.7 and Grefenstette [1986] recommended a higher rate of 0.95. Back, Fogel and Michalewicz [1997] have reported no significant difference between the low level and high level of the crossover rate on optimization so therefore it was not empirically determined in this study. The double-point crossover with a high constant crossover rate of 0.9 is applied in the experiments.

One important question is the number of individuals (strings) needed in a population. The population size is a critical decision as EC algorithm converges more rapidly with smaller populations whereas with larger populations, it performs better in terms of solution quality. Alander [1992] empirically investigated population sizes for EC algorithms. He suggested $\log_2 N$ and $2\log_2 N$ as an ideal range for the best population size where *N* is the size of the problem's search space [Alander 1992]. For test suite minimization problem, $N = 2^n$, a value between n and $2n$ would be optimal for the population size. In order to determine the population size empirically, an experiment is performed with five profiles (labelled as Q_0, Q_1, Q_2, Q_3, Q_4) to cover both the ideal and imperfect type of values such as upper bound, median, lower bound, out of upper bound and out of lower bound. Given that $N = 2^n$ is the size of the search space of the test suite minimization problem where n is the string length, the lower bound $Q_1 = \log_2 N$, median $Q_2 = [\![1.5 \times Q_1]\!]$, upper bound $Q_3 = 2 \times Q_1$, out of lower bound $Q_0 = [\![0.5 \times Q_1]\!]$ and out of upper bound $Q_4 = [\![2.5 \times Q_3]\!]$ are calculated and result of the experiment is presented in Figure. Although the profile Q_4 on average produced better results than other profiles as shown

in Figure 5-1(a), it was not statistically significant using Tukey HSD Post hoc test at the 0.05 level of significance. It is important to note that a larger population size means a larger search space and more CPU time usage. Therefore, basing the selection decision for population size merely on the superiority or inferiority of the solution without considering the cost is inadequate. Given that the additional computation time and space needed for a larger population size is without any significant gain, it is safe to say that using a smaller population limit (lower bound) is more efficient than the upper limit (upper bound) for an EC-based test suite minimization. Considering this, the initial population is randomly generated for each problem instance according to the associated lower bound mentioned in Table 5-3.

The single-point constant rate mutation is applied with probability P_m. For mutation probability values, very diverse recommendations have been found in the literature. Back [1993] suggested a high mutation rate such as $\frac{1}{\ell}$ where ℓ is the string length as an optimal rate, whereas Schaffes et al. [1989] recommended 0.005 at the other extremity of the mutation rate. In order to determine an optimal value, an experiment is performed where P_m values are selected from 0.005 to 0.5 thereby covering both extremes. Twelve mutation profiles, labeled as (M_0, M_1,..., M_{11}), with different probability rates are given as $P_m = M_x$ and $0 \leq x \leq 10$. Apart from M_0 which is calculated using $\frac{1}{\ell}$, a fixed value mutation rate is used. From the mutation profile graph in Figure 5-1(b), the positive effect of the mutation rate on the EC-based test suite optimization is very obvious. Apart from the two extreme values (M_1 and M_{11} with mutation rate of 0 and 1), the solution quality (reduction %) increases with the mutation rate generally. For mutation rates of less than 0.01 (M_5), the average solution quality was the lowest. Another example of this phenomenon can be seen in Figure 5-1(b) at M_0 for ATM-111 and ATM-133 where the mutation rate is 0.009 and 0.007 respectively and the results were almost similar to that of M_2, M_3 and M_4 which are less than M_5. On average, high mutation rates resulted in better solutions. All the results except for mutation rates of M_6 and M_9 are inconsistent across the different test suites. However, it can be safely inferred that higher mutation rates are better to start the search with for test suite minimization problem. Considering this, a high standard mutation rate of 0.2 is applied in the study for each problem instance.

The bound on the number of generations is usually determined empirically for each problem instance. Therefore an appropriate value as shown in Figure 5-2 was determined via initial experiments and used for rest of the replications. In case of the ECCS problem, the search converged in around 15 generations for all three problem instances. In case of the DTP problem, the search converged less than 25 generations for all problem instances. In case of the ETP and the ATM problems, the search took up to 19 and 29 generations respectively to converge to a

solution. In all cases, the algorithm was further executed up to 100 generations with an assumption that it will give the algorithm ample opportunity to improve the solution.

Table 5-3: Bounds on the population size in EC for test suite minimization based on Alander's empirical study [Alander 1992]

Test Suite Size	Lower bound	Upper bound
20	20	40
25	25	50
30	30	60
89	89	178
111	111	222
133	133	266
27	27	54
34	34	68
41	41	82
28	28	56
35	35	70
42	42	84

The summary of preliminary values of the EC parameters is given in Table 5-4.

Table 5-4: Summary of parametric settings used for EC in the experiment

Parameters	Values
Objective	Minimize Test Suite Size
No. of Generations	100
Replacement Scheme	Generational
Crossover Rate	0.9 (double point)
Mutation Rate	0.2 (single point)
Selection Scheme	Pairwise tournament

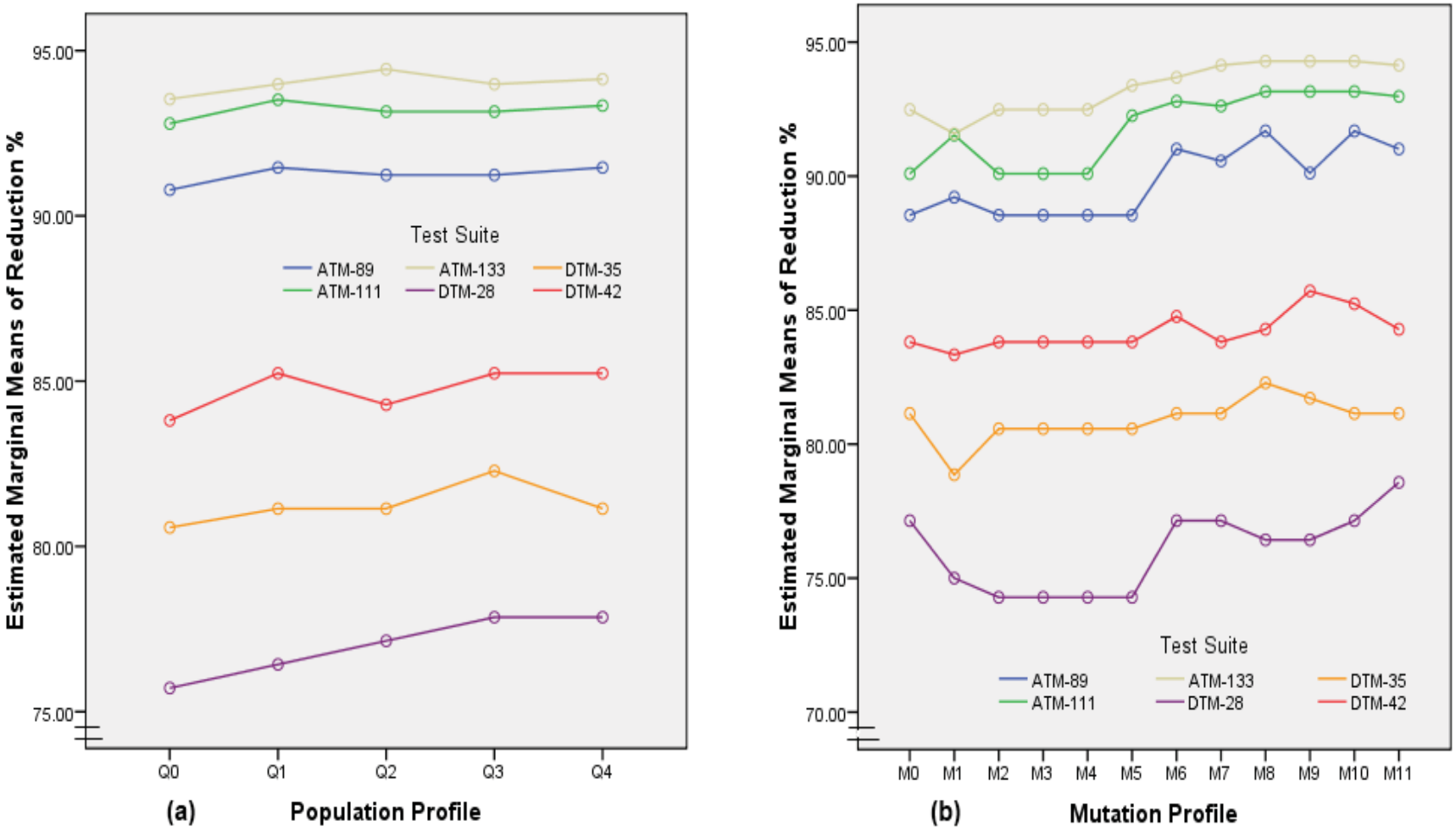
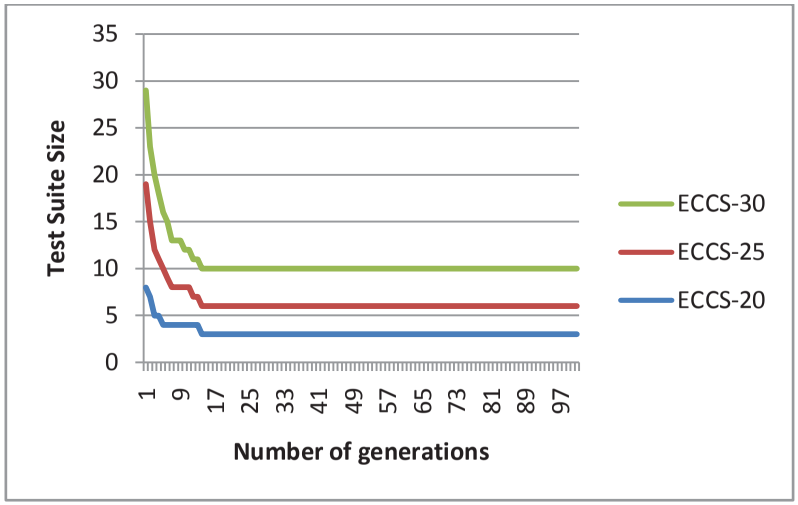
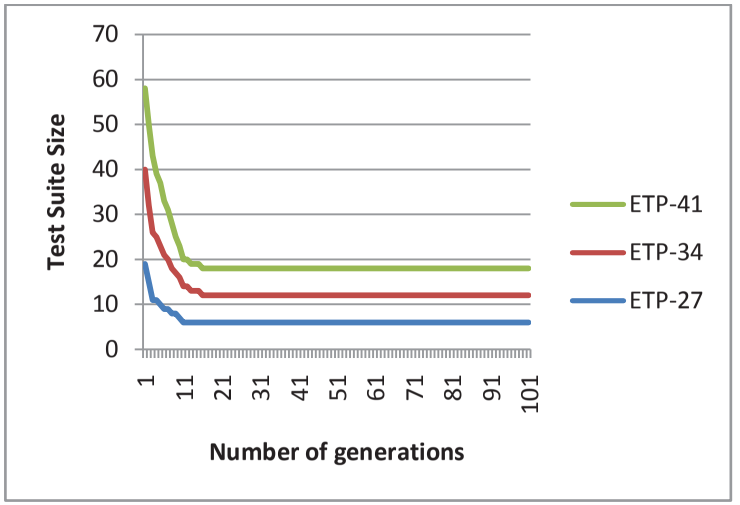
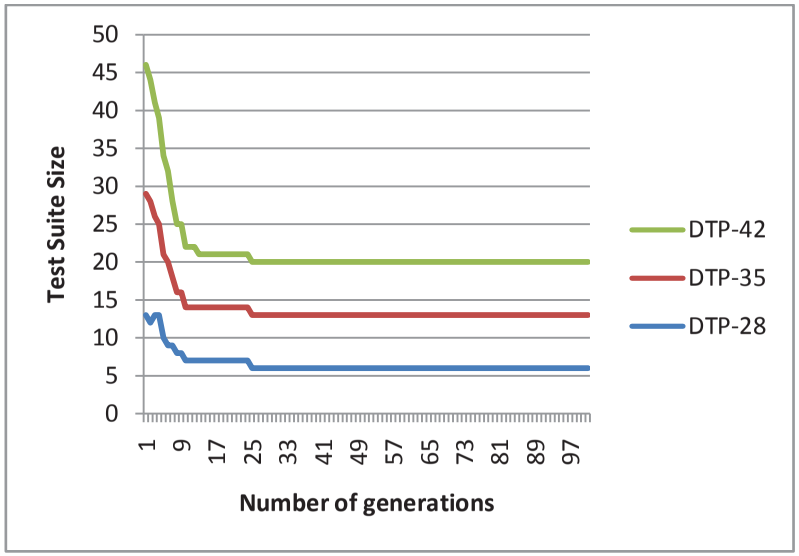


Figure 5-1: Offline performance analysis (reduction percentage) of EC with various population and mutation profiles for six test suites (three DTP and three ATM)







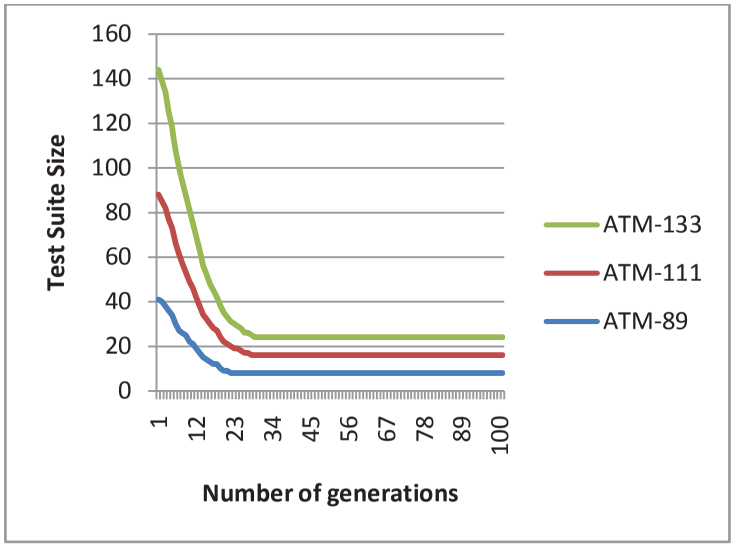


Figure 5-2: Maximum number of generations for different test suite problems was determined empirically such that the algorithm can have ample opportunity to explore the search space before stopping.

For the Greedy algorithm, the test cases were sorted initially in decreasing order of fitness according to their individual coverage level. It then proceeded to include them into the test suite, starting with a randomly selected test case and then include test cases according to their contribution to the fitness of the test suite until it reached the level of 100%.

For Hill Climbing, two versions of the algorithm, HCNA and HCRA, were used. A search point was arbitrarily selected initially. After that, for each step up to the maximum defined iterations, the next item is selected according to the selection schemes of HCRA and HCNA and replaces the current item.

In consideration of the stochastic nature of the EC algorithm and the random starting points for the HCRA and the HCNA algorithms, each experiment was replicated 10 times in order to address the possible effect of randomness on the results.

5.3.3 Metrics

Metrics are crucial to assist the decision making process when integrated with optimization techniques in order to obtain the optimal or better alternate solution for process improvement. The following metrics are used to assess the effects of variables in this experiment:

The efficiency metric defined in Section 5.2.1 is used to measure the test suites in terms of their execution and validation cost. Various cost metrics such as the number of test cases in a test suite, the number of method invocations in the test suite, execution and validation time of each test case and the number of test cases per unit of given criterion have been reported in the literature [Beizer 1990; Briand, Labiche and Wang 2004]. A higher efficiency score indicates relative higher redundancy in a test suite. The cost of test execution and validation is very much implementation and configuration dependent. Therefore, an estimated cost of potential execution and validation of a test case is used in this study. Furthermore, it is assumed that the execution and validation cost is equal on average for all test cases in a test suite [Friedman et al. 2002]. It means though that the test cases with a higher coverage could take longer to execute than smaller test cases but on average still have similar cost [Friedman, Hartman, Nagin and Shiran 2002]. This assumption might cause the optimization to be less sensitive to the composition of a test suite and favour the longer test sequences but can be justified if the setup and initialization overhead of smaller test cases is included. Nevertheless, despite these differences, for the sake of simplicity a constant unit of cost is used for all test cases.

The efficacy metric defined in Section 5.2.1 is used to measure the effectiveness of a test suite. When this value is less than 1 (in terms of ratio) or 100%, it indicates deficiency in the test suite (indicates the inadequacy of the test suite with respect to the stated criterion) and the need to generate more test cases. Many effectiveness measures e.g. mutation score, fault detection rate, fault detection probability have already been reported in the literature. As the original test suites were generated for branch coverage criterion, coverage based criterion called AD-based branch coverage criterion that was previously defined in Chapter 4 was used. It was used as a surrogate measure in efficacy metric for both test suite effectiveness and redundancy and is referred to as the average branch (element) coverage per test case and average test cases per branch (element) respectively.

As the search in heuristic algorithms is guided by the fitness function, the function associated with test suite size and coverage (equation-(5-3)) is used to calculate the fitness of a generated solution. Various measures (i.e. solution quality, time and space) can be used for comparing the performance of optimization algorithms. A number of performance metrics for evolutionary algorithms have been reviewed in Chapter 2. As the optimization techniques used in this study are heuristic-based that improve the solution gradually, evaluating their performance based on the solution they produce at the end of the optimization process seemed to be a reasonable metric. The *best-value* metric (for detail see Chapter 2) was used to compare the performance of algorithms in terms of test suite reduction. For performance evaluation of algorithms with respect to different test suites, the percentage reduction in a test suite size was used and denoted as “Reduction %”.

Additionally, three statistical analyses were conducted using SPSS [IBM 2010] to examine the optimization effect, assess whether difference was statistically significant and to analyze the correlation.

1) The t-test of mean difference between paired Original and Optimized test suites in terms of size is a simple test of significance. The t-test compares the means of two groups and then test whether the difference is zero or significantly larger than zero. The t-test assumes that the population is normally distributed. However, it is robust to the deviations from normality if the sample size is large [StatSoft 2011]. As there are 2400 data points in original and optimized test suites, the assumption of normal distribution is not an issue and it is safe to use t-test in this case. The presence of outliers in data can comprise the reliability of results. The outliers in the data were visually identified through box-plot and Q-Q plot and Winsorizing [NIST 2011] was applied to recode them to the nearest boundary value.

2) The analysis of variance (ANOVA) will allow us to address the question of whether there are meaningful and statistical differences in test suite reduction among the heuristic algorithms under the influence of constant factors known to affect the performance. The Tukey HSD post hoc test is widely accepted for analysis of variance in multiple factor experimental model and hence applied for multiple comparisons in this study. In order to apply the ANOVA, the assumptions of normality and homogeneity of the data need to be satisfied. The F test used in ANOVA is considered quite robust against the violations of these assumptions in case of large sample size [StatSoft 2011]. Owing to very large and balanced group size, the Tukey HSD post hoc test is reasonably safe to be used here.

3) In order to determine the relationship between the test suite size and reduction in redundancy such as decreasing the test suite size by an amount or percentage may or may not equally affect the reduction, the correlation analysis was performed.

5.3.4 Result and Discussion

The summary of results for the experiments is presented in Figure 5-3 and they provide an insight into the performance of optimization algorithms with respect to "Reduction %". The items along the x-axis named as HCRA, HCNA, GD and EC represent Hill Climbing Random Ascent, Hill Climbing Next Ascent, Greedy and Evolutionary Computation algorithms respectively. The results illustrated three significant phenomena: (1) Significant reduction in most of the test suites without affecting their effectiveness, (2) Consistent and scalable evolutionary test suite optimization and (3) better performance by EC than Greedy, HCRA and HCNA algorithms in most cases. Although the reduction in test suites is visually obvious from the data, in order to determine if this difference was statistically significant or just random, the Paired Samples t-test was applied which confirmed that the differences are significant. This was evidenced by the mean difference between the pair of variables (Original test suite size and reduced test suite size) and $t = 55.547$ at 95% confidence interval in Table 5-5, as strong evidence of test suite reduction is significant. In all cases, the final optimal test suite had the same coverage level as in the original test suite.

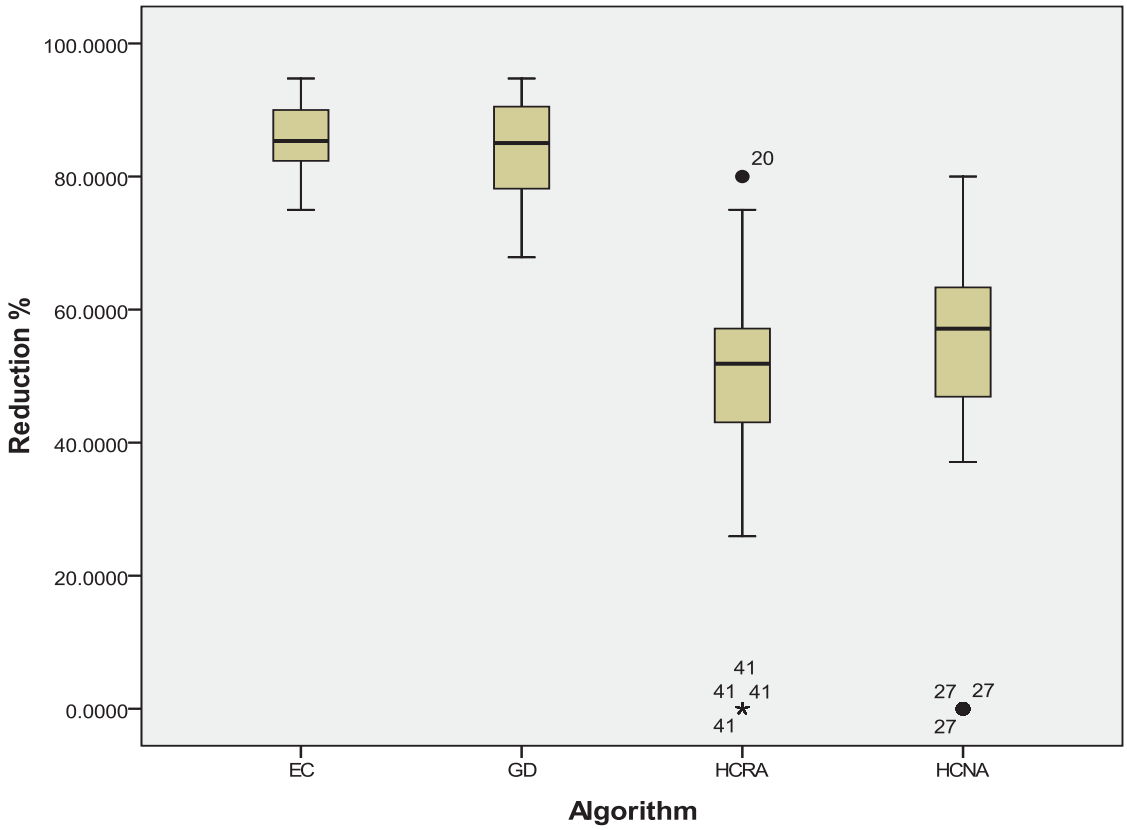


Figure 5-3: Box-plot for test suite reduction with respect to different algorithms (the outliers are shown by the size of the test suite, e.g. 41 is a test suite size associated with the ETP – see Table 5-2)

Table 5-5: Paired Sample t-test between the Original and Reduced Test Suite

	Paired Differences					
			Std. Error	95% Cl of the Difference		
Pair	Mean	Std. Deviation	Mean	Lower Bound	Upper Bound	t
Original Size - Reduced Size	33.660	29.735	.607	32.469	34.850	55.457

Table 5-6: Composition of minimized test suites (minimized using EC algorithm) versus original test suites

		Average Branch Coverage per Test Case		Average Test Cases per Branch	
Model	Test Suite Size	Original TS	Minimized TS	Original TS	Minimized TS
ECCS	20	49.12	62.75	9.82	1.90
	25	49.88	66.67	12.47	2.00
	30	50.98	70.59	15.29	2.10
ATM	89	10.88	20.92	9.68	4.93
	111	11.65	19.05	12.93	5.89
	133	11.36	20.98	15.11	4.61
ETP	27	17.24	33.97	4.66	2.04
	34	15.84	33.97	5.39	2.04
	41	16.89	33.46	6.92	2.00
DTP	28	33.14	41.67	9.28	2.50
	35	32.22	40.28	11.28	2.40
	42	32.61	39.81	13.69	2.40

Test Suite Reduction: Table 5-6 summarizes the composition of the reduced test suites in comparison with the original test suites. It shows an increase in the efficacy of the test suite (average number of test cases per branch) and the reduction in cost as fewer test cases on average per branch in a minimized test suite as compare to the original test suites. For instance, in test suite DTP-42 (column 'model' – 'test suite size' in Table 5-6) there were 13.69 test cases per branch on average in the pre-optimization version which was reduced to 2.4 in the post-optimization version which is a significant improvement in terms of efficiency. Furthermore, on average there were 32.61 branches per test case in the original version of the same test suite (DTP-42) and 39.81 branches per test case in the optimized version thereby indicating the improvement in the effectiveness as well.

Effect of Optimization Technique: The average reduction percentage of all test suites with HCRA, HCNA, GD and EC algorithms were approximately 47, 50, 84 and 86 respectively. Due to the stochastic nature of the algorithms, some variation in the results was expected and evident in the spread of the results as shown in Figure 5-3. The wider spread of results shown between the upper and lower whiskers in Figure 5-3, indicate relatively more inconsistent performance of the algorithms. Although, the average performance of the Greedy algorithm seemed slightly poorer than that of the EC algorithm, over a large data sets (12 test suites with different sizes, five reshuffled versions of each test suite and ten replications) it was found to be statistically significant at the 0.001 significance level (see the difference between the mean reduction % with EC and GD algorithms as shown in Table 5-9). The large spread of results and outliers for HCNA and HCRA confirmed the known issues with these algorithms i.e. inconsistent and un-scalable performance. It suggests that the inconsistency in the Greedy algorithm's performance is due to its non-exploratory searching mechanism which is a known fact that often causes it to converge to a non-optimal solution too early. Both the variants of Hill Climbing often got trapped into local optimum, and in some cases even failed to improve the initial solution (see outliers marked as 41 and 27 in Figure 5-3). EC has a natural advantage over other algorithms owing to its implicit population based, multidimensional search that leverages evaluation of multiple points in parallel.

The test suite reduction using the EC algorithm for each test suite was 75% or higher which was quite phenomenal. The data substantiates the stability and robustness of the proposed evolutionary framework for model-based test suite minimization in comparison to the HCRA, HCNA and GD algorithms. The ANOVA table is presented in Table 5-8 indicating that the test statistic (1241.20) is much larger than the critical value (5.43 which is a tabular value of F distribution). Hence, the variation in the performance of algorithms was statistically significant.

The column (mean reduction % difference) in Table 5-9 reports the difference between each pair of means. In the first row, the mean reduction of GD is subtracted from the mean reduction of EC and yields 1.94 (EC-GD) as the mean difference between these two groups. An asterisk next to the mean difference flags the pair of group means as significantly different at the 0.001 level of significance.

The results of Post-Hoc Tukey HSD (pair-wise) comparison of EC-GD, EC-HCRA and EC-HCNA showed significant differences at the 99.9% confidence interval. It can thereby be inferred that the EC algorithm, on average performs better than the Greedy, Hill Climbing Random Ascent and Hill Climbing Next Ascent algorithms for test suite minimization.

Effect of Test Suite Size and Composition: As mentioned earlier, some variation in the performance of heuristic techniques is generally expected due to their stochastic nature. However, the complexity of an optimization problem, e.g. search space size and structure, can render a particular instance of a problem intractable and may also affect the performance of the applied heuristic technique [Deb 1997; Mitchell, Forrest and Holland 1992; Rothlauf 2006]. Therefore, it is pertinent to investigate the effects of the test suite's composition (problem complexity) if any, upon the performance of the algorithms used.

Table 5-7: Pearson Correlation Matrix among TS Sizes and TS Reductions w.r.t. algorithm

Algorithm		Test Suite Size
EC	Reduction	0.772**
GD	Reduction	0.652**
HCRA	Reduction	-0.364**
HCNA	Reduction	-0.409**

** $p < 0.01$ level (2-tailed) using Pearson Correlation, (N=600).

As mentioned earlier in Section 5.3.2 that multiple versions of a test suite for a model were produced by incorporating additional test cases. The increase in test suite size was expected to increase the degree of redundancy in the test suite. However, the reduction in test suite size was not reflected in the same proportions in HCRA, HCNA and GD algorithms based optimization. In order to see if there is any interaction of test suite size upon optimization with respect to a particular algorithm, correlation analysis was applied. Table 5-7 provides bivariate correlation matrix among the test suite sizes and reductions with respect to the algorithms. The

analysis confirmed the following statistically significant results: 1) there is strong positive association between EC-based reduction and test suite size which means the performance of EC was least influenced by the test suite size and indicates the scalability of the algorithm; 2) there is a strong positive relationship between the test suite size and GD-based reduction which indicates that problem size was not an issue with the GD algorithm; and 3) a moderate negative correlation between test suite size and HCNA or HCRA which highlights the scalability issue with both of the Hill Climbing algorithms.

Earlier in Section 5.3.2, it was hypothesis that the change in the arrangement of the test cases in a test suite will change the structure of the search space and may thereby affect the optimization of a test suite in general or the performance of a heuristic specifically. As mentioned in Section 5.3.2, in order to determine the effect of test cases arrangement on test suite minimization, each test suite was shuffled five times randomly and optimized. In general, there was a very clear variation observed in reduction (%) due to the reshuffling of the test suites resulting in changes in the arrangement of test cases (see Figure 5-4).

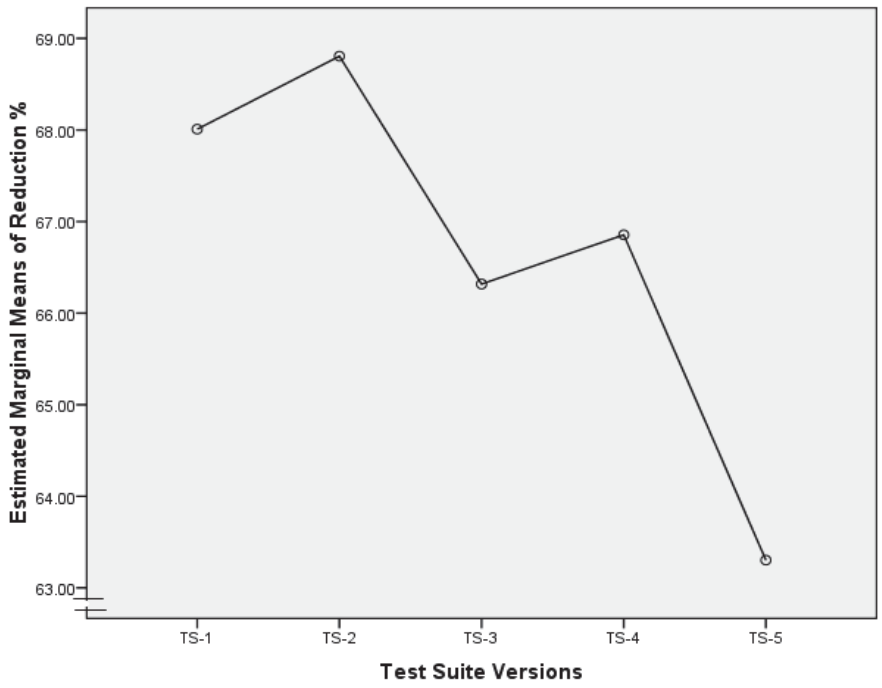


Figure 5-4: Mean reduction for five versions of test suites. See the variation in mean reduction due to reshuffling of test suite causing the rearrangement of test cases.

Table 5-8: Summary of ANOVA

	Sum of Squares	df	Mean Square	F
Between Groups	784610.52	3	261536.84	1241.20
Within Groups	504865.32	2396	210.71	
Total	1289475.84	2399		

Table 5-9: Tukey's HSD comparison between algorithms for test suite reduction

Comparisons	Mean Reduction % Difference	Std. Error	99.9% Confidence Interval	
			Lower Bound	Upper Bound
EC vs. GD	1.94*	0.40	0.45	3.43
EC vs. HCRA	38.47*	0.74	35.70	41.25
EC vs. HCNA	35.63*	0.90	32.25	39.01
GD vs. HCRA	36.53*	0.78	33.61	39.46
GD vs. HCNA	33.69*	0.93	30.18	37.19
HCRA vs. HCNA	-2.85	1.12	-7.05	1.36

* $p < 0.001$

Given the observed differences in the performance of heuristic algorithms, some degree of variation in the shuffling effects on test suite reduction (%) for different algorithms was expected. In order to see that variation and interaction between the shuffling and the test suite size on test suite reduction, line-plots were drawn for each of the four algorithms as shown in Figure 5-5, Figure 5-6, Figure 5-7 and Figure 5-8. The effects of change in the arrangement of test cases on test suite reduction can be seen clearly from the line-plots depicting the estimated marginal means of reduction percentage for five versions of all test suites. From Figure 5-5 and Figure 5-6, it can be seen that the performance of both Hill Climbing algorithms, HCRA and HCNA, was greatly affected by the change in the arrangement of test cases. Similarly, the test suite reduction with EC and GD algorithms also affected as shown in Figure 5-7 and Figure 5-8 by the arrangement of test cases though less severely and variably. It is important to note that in case of GD algorithm, the effect of test suite shuffling on test suite reduction was least likely to occur because of the sorting mechanism. However, it appeared that the sorting algorithm could not shield the later 'greedy solution construction' mechanism from shuffling effect because of the redundancy and produced different sorted lists for different versions of a test suite. Figure 5-7 and Figure 5-8 revealed another interesting aspect that the shuffling in most cases resulted in slightly better overall result than for the initial version of test suite for EC algorithm whereas in the case of the GD algorithm case the effect was mostly negative. The analysis indicated the effect of arrangement of test cases on test suite reduction and on the performance of heuristic algorithms. Given that studies have shown that the performance of heuristic algorithms can vary depending on the structure of the fitness landscape [Deb 1997; Rothlauf 2006], a set of experiments was designed and presented in Section 5.4 to better understand the landscape of the shuffled test suites.

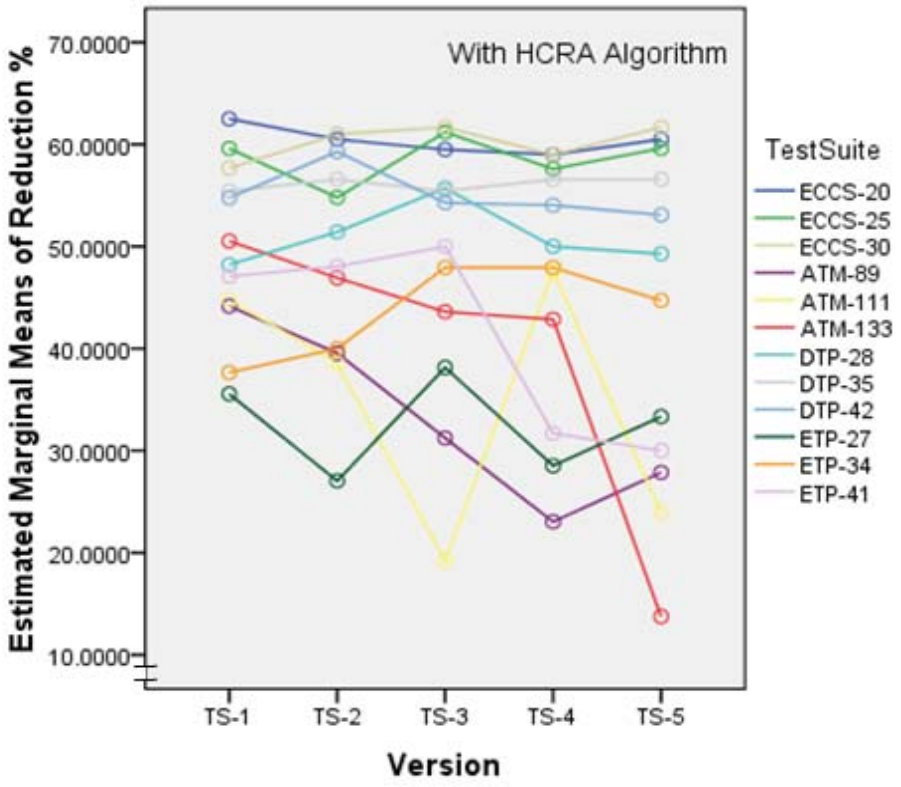


Figure 5-5: Graphs of estimated marginal means of reduction percentage with HCRA algorithm depict the difference in performance due to test suite shuffling.

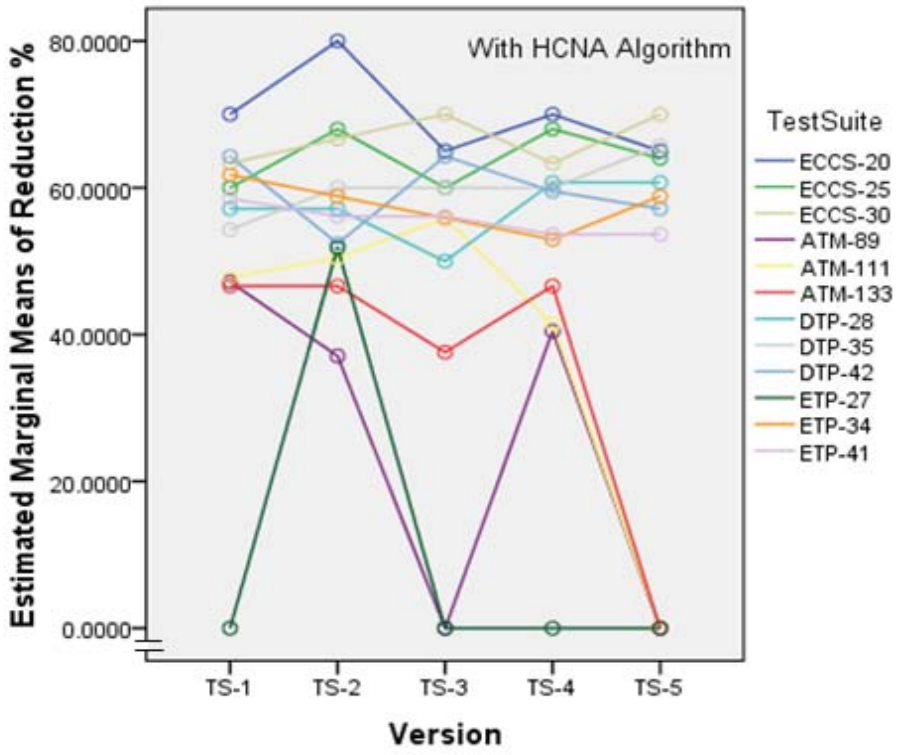


Figure 5-6: Graphs of estimated marginal means of reduction percentage with HCNA algorithm show the difference in performance due to test suite shuffling.

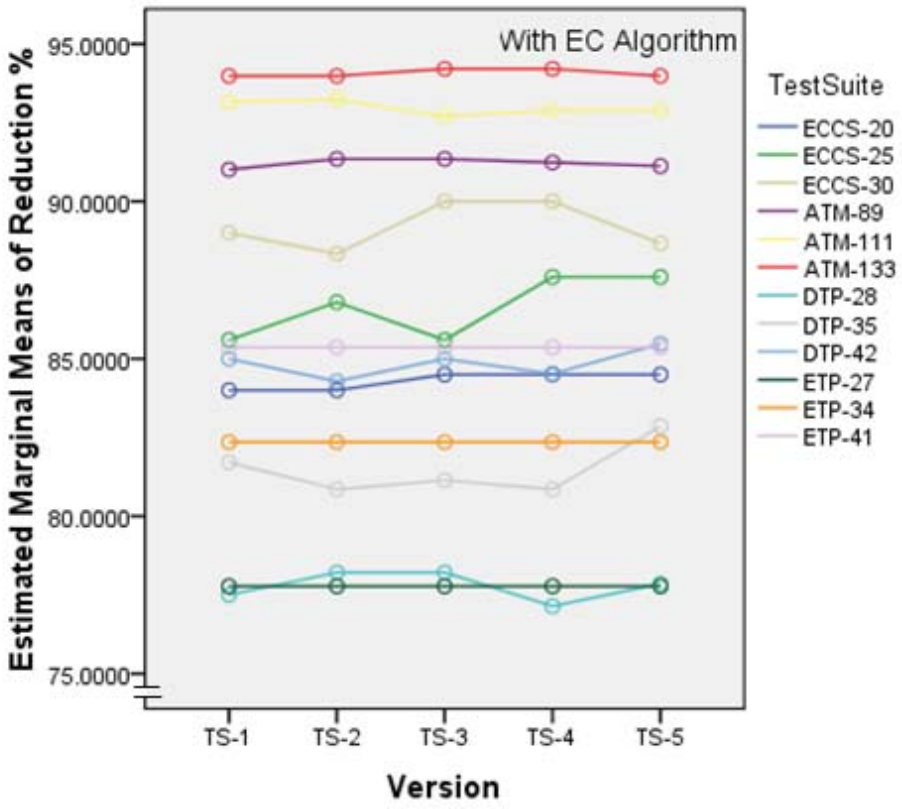


Figure 5-7: Graphs of estimated marginal means of reduction percentage with EC algorithm show the difference in performance due to test suite shuffling.

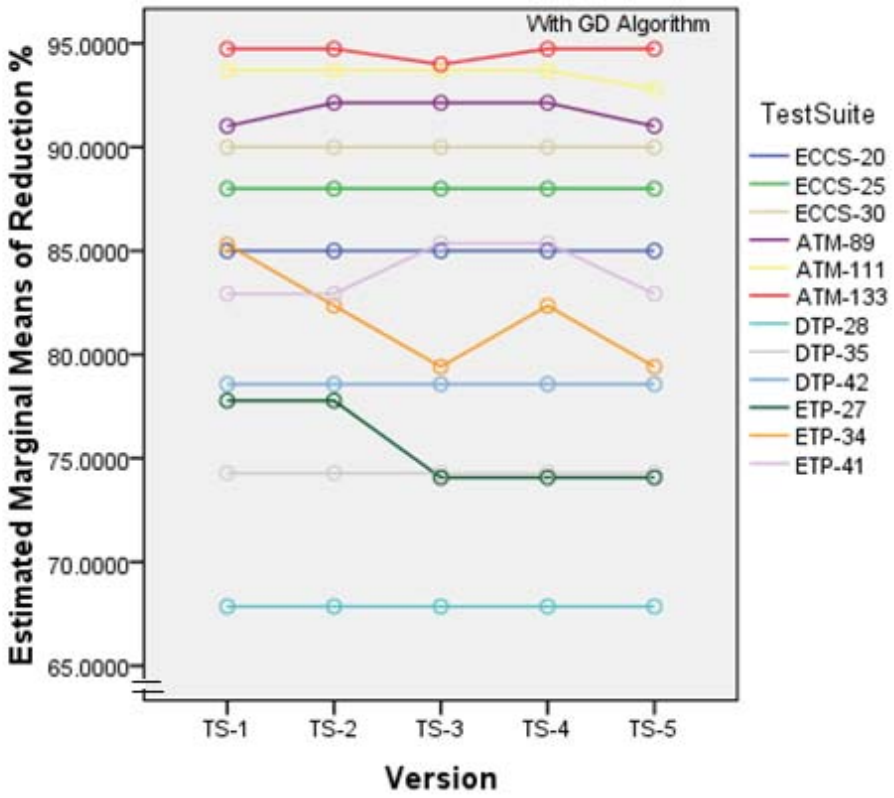


Figure 5-8: Graphs of estimated marginal means of reduction percentage with GD algorithm show the difference in performance due to test suite shuffling.

5.4 Characterizing the Fitness Landscape of Test Suite Minimization

In the previous experiment, it was found that the arrangement of test cases in a test suite can affect the performance of an optimization algorithm and the outcome of test suite minimization. Therefore, it deemed necessary to characterize the fitness landscape of test suite minimization problem which may help us to understand the problem better and develop appropriate search strategies which may result in better results.

Heuristic techniques find optimal solutions by visiting solution points in the search space, evaluating their fitness and converging to the fittest solution. Studies have shown that their performance can vary depending on the structure of the fitness landscape of the given search problem [Deb 1997; Mitchell, Forrest and Holland 1992; Rothlauf 2006]. In terms of landscape, the structure of a landscape is specified by its characteristics namely, smoothness, ruggedness and neutrality. A landscape can range from a uni-modal and very smooth to multi-peaked and very rugged landscapes. A landscape where the average fitness difference between the neighboring points is relatively small is called smooth and finding good global optima within such a space is relatively easier as local information can be used to guide the search. A landscape with a relatively large average fitness difference between neighbors is called rugged and finding good global optima within such a landscape is relatively difficult as the available local information is less useful. The third landscape characteristic of neutrality is associated with building blocks in the landscape.

Correlation functions have been used in various types of search landscapes e.g. NK-model and combinatorial optimization problems, to analyze and classify them based on their structural characteristics. Using the correlation function on NK-model, Kauffman [1993] showed that the properties of landscape vary as a function of rugged and multi-peaked landscape. NK-model is a stochastic fitness function on bit-string to generate fitness landscape with N genes/points and K interactions between these genes/points, where the value of K ranges between 0 and N-1. He explored the link between the epistasis and the ruggedness of landscape and identified the following properties:

1) With high epistatic coupling (interaction between genes/points), the landscapes become progressively less correlated and highly rugged. So with K=0, when there is no epistatic interactions between genes, the fitness landscape is fully correlated and smooth, whereas at K=N-1, each gene is affected by all the remaining genes which indicates the landscape is fully uncorrelated and rugged.

2) In smooth and correlated landscape, the fitness values at one point are more or less similar to the fitness values at the neighboring points. Moreover, a fully smooth correlated landscape is a unimodal (contains a single global optimal solution) and all the one-mutant neighbors have a similar fitness. Whereas, in a fully rugged landscape fitness values are entirely uncorrelated (random) and contains several peaks (a.k.a. multimodal landscape).

3) The selection gradient to the optimal point (peak) is steeper in the rugged landscape than in the smooth landscape, thereby affecting the convergence of the search.

4) For K=0, the landscape is easy to explore, or in other words, the optimization problem is easy to solve. Moreover, increasing K decreases the success rate. The higher K is, the sooner the evolution (a.k.a. premature convergence) ends.

Since the relationship between the fitness space $\mathcal{L}$ and search space X by fitness function which defines the quality of a point (solution) in the search space, the fitness landscape is defined in [Tavares et al. 2008] by the following tuple:

$$\mathcal{L} = (X, f, N_k) \qquad (5\text{-}4)$$

where X is a set of all points in search space, f is a fitness function to compute fitness value for each point , and neighborhood structure N of size k defined over set X by distance metric d as follows:

$$N_k(x) = \{y \in X : d(x, y) \leq k\} \qquad (5\text{-}5)$$

5.4.1 Measures for Landscape Analysis

For a 2-dimensional search space, visualization is through the surface map and for a 3-dimensional search space the additional 'elevation' dimension can easily represent the smooth or rugged nature of the fitness landscape. However, this intuition based analysis is not only hard to extend for the higher dimensional search space, but also tends to be misleading [Deb 2001]. Therefore, in order to evaluate and compare the fitness landscape, various measures have been defined, such as, density of local minima, fitness function distribution and correlation functions. Following are some of the measures that can be used for fitness landscape analysis.

Autocorrelation Function: Two types of techniques, average correlation and stochastic correlation, are used to determine the correlation structure of a fitness landscape. In average correlation, the average correlation level between points in the fitness landscapes is measured [Weinberger 1990]. It is calculated for all pairs of points in the search space with distance '*d*' by the difference between the fitness of the pairs. The high correlation level due to the similarities between the fitness of the points indicates the smoothness of the landscape. Whereas, the low correlation level due to higher differences in fitness values between points implies a rugged landscape. Accordingly, the average correlation between points in a landscape can be estimated by sampling a large data set and is defined as follows:

$$\rho(d) = \frac{E(f_x \times f_d) - E(f_t)E(f_d)}{variance(f)} \qquad (5\text{-}6)$$

where E is the expected or mean value, f_x is the fitness of the point x and f_d is the fitness of each point in the search space with distance *'d'*.

The second technique (stochastic correlation) involves estimating the correlation structure by applying a random walk of a time series, starting from an arbitrary genotype and then randomly moving step by step in the search space using single point mutation. A number of single-point mutation strategies (e.g. adjacent neighborhood and random neighbourhood) have been reported in literature [Kauffman 1993]. Figure 5-9 illustrate the difference in both adjacent and random neighbourhood types of mutation strategies. An assumption about the random walk based analysis is that the landscape is isotropic. An isotropic landscape means that the landscape is statistically similar (on average) from any point and regardless of the starting point the results of the random walk will always be the same. For stochastic correlation method, following equation is used to compute the correlation level:

$$\rho(t,s) = \frac{E(f_t \times f_{t+s}) - E(f_t)E(f_{t+s})}{variance(f)} \qquad (5\text{-}7)$$

where E is the expected or mean value, f_t is the fitness of the point (genotype) at the t^{th} step and f_{t+s} is the fitness of the point (genotype) '*s*' steps apart in the random walk.

The value of autocorrelation function $\rho(t,s)$ *or* $\rho(d)$ ranges from -1 to 1. In case of $\rho(t,s)$ *or* $\rho(d)$ is closer to -1 or 1, means the stronger correlation between the two points apart by '*s*' steps or *'d'* distance away. Whereas, if it is closer to 0 then there is less correlation between the two points.

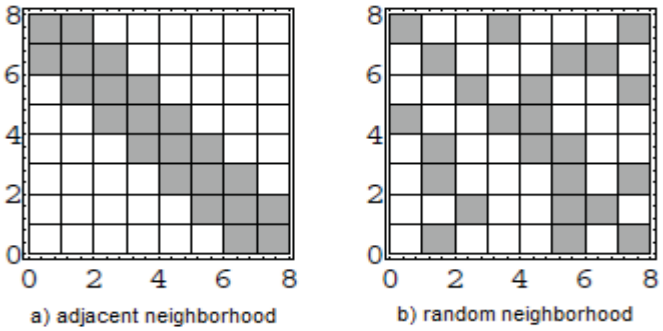


Figure 5-9: Two different neighborhood one-step mutation strategies, (a) Kauffman's adjacent neighborhood (N=8, K=2), (b) Kauffman's random neighborhood (N=8, K=2). [Back, Fogel and Michalewicz 1997]

Correlation Length: The correlation length metric is defined as the rate of decrease in the correlation between the fitness of two points (e.g. correlation between the fitness of parent and child) and often used to assess the ruggedness of a fitness landscape [Jones 1995]. A higher value of the correlation length ℓ indicates a smoother landscape, whereas a lower value of ℓ indicates a more rugged landscape. It is based on the autocorrelation function and computed as follows:

$$\ell = \frac{-1}{\ln\left(|\rho(d)|\right)} \qquad (5\text{-}8)$$

where d is hamming distance and $\rho(1) \neq 0$. In order to estimate a value, it is usual to normalize it with the diameter of the landscape that can be used to determine if it is significantly different to zero. The correlation length closer to 1 is considered to be indicative of a higher correlation, whereas a zero or closer to zero length is indicative of no correlation or a lower degree of correlation.

Number of Local Optima: Another approach to measure the structure of a landscape is the number of local optima. Local optima are considered as obstacles for optimization techniques and fewer local optima mean less chances for a search to trapped into a local optimal. However, it is not an absolute indicator as the size of the basin of attraction of local optima is also an important factor. Moreover, studies [David 1987; Horn and Goldberg 1995] have shown that a high number of local optima does not necessarily render a problem hard for optimization. In binary encoding, there are two alternate alleles per locus so the total number of genotypes is 2^N. The expected total number of local optima with respect to a single-point mutation is:

$$\mathrm{M}_1 = \frac{2^{\mathrm{N}}}{\mathrm{N}+1} \qquad (5\text{-}9)$$

Fitness Distance Correlation: Another approach to measure the structure of a landscape is to determine the extent to which the fitness values are correlated with distance to a global optimum in the search space. The fitness distance correlation coefficient ρ [Jones and Forrest 1995] can be computed as follows:

$$\rho = \frac{C_{FD}}{\sigma_F \sigma_D} \qquad (5\text{-}10)$$

where $C_{FD} = \frac{1}{n}\sum_{i=1}^{n}(f_i - \bar{f})(d_i - \bar{d})$

C_{FD} is the covariance of set $F$and D, $F = \{f_1, f_2, \dots, f_n\}$ is the set of fitness values of n individual points x_i and $D = \{d_1, d_2, \dots, d_n\}$ is a corresponding set of distances to the nearest global maximum. The $\bar{f}$, σ_F, $\bar{d}$ σ_D are the average mean and standard deviation of $F$and D respectively. When the search objective is to maximize then the fitness increases as the distance to the global optimum decreases. Therefore, for an ideal landscape the value of ρ will be -1. Whereas, for minimization the ideal landscape will have a value of $\rho = 1$. In an ideal landscape the search should be easy and indicates the existence of a path via solutions with better fitness values.

5.4.2 Test Suite Minimization Landscape Analysis

Given the known association between the fitness values of neighbors and the problem of the smoothness/ruggedness of the landscape, the following experiment was designed to analyze the fitness landscape of test suite minimization problem and to get a better insight of the problem. It was hypothesized that rearranging the test cases in a test suite may change the correlation level among genes (test cases) which in turn may enable smoothing the fitness landscape and make it more favorable for search. A thorough analysis of the complete fitness landscape for a combinatorial optimization problem is generally considered difficult due to the huge size of the search space. Hence, the decision to use representative parts by sampling through the search space in a random fashion was made. In addition, fitness distance correlation and correlation length measures were used for evaluation and comparison of the landscapes. The objective was to express the correlation structure of the fitness landscape and compare the landscape of the different versions (reshuffling) of a test suite. For each test suite version, Kauffman's neighborhood random walk [Kauffman 1993] of 10,000 steps was performed using the single-point mutation on its landscape. The single-point mutant walk passes through points in the search space regardless of their fitness differential. This way, a time series of fitness values was generated to which the autocorrelation approach was applied for determining the

correlation length. For autocorrelation analysis, it was assumed that the problem is isotrophic, which meant that the contribution of each bit-string (gene) position was normally distributed and independent of each other.

For fitness distance analysis, the global optimum needs to be known. In terms of the analysis described here, the known global optimum which was determined during an earlier experiment was used. Although there could be more than one global optimum in a search space, only one global optimum is considered for a test suite. Moreover, the measure of hamming distance d_H at genotype level, simply counts the number of positions in which two aligned solutions differ. It is used to determine the proximity of the solutions encountered in a random walk with the known optimum solution.

The pseudo code for Neighborhood Random Walk is presented in Figure 5-10.

- Select an item (genotype) arbitrarily from the search space
- For each step up to MAX_ITERATIONS
    - Select an item randomly from one-mutant neighbours
    - Compute and record the fitness value of the selected item
    - Compute and record the hamming distance between the current and the selected item.

Figure 5-10: Neighborhood Random Walk Pseudo Code (adapted from [Kauffman 1993])

In Table 5-10, the results of the measures applied in the experiment are presented. In the first column, names of the test suites that were used in the experiment are given. The second column lists the measures used (Fitness Distance Correlation ρ, Correlation Length ℓ and Normalized Correlation length ξ). The data for fitness distance correlation measure indicates that landscapes of test suites are generally not search friendly. It was observed that the value of ρ was quite far from the ideal values (1 and -1). However, as other studies [Jones 1995] have also suggested, this criteria alone cannot render the given optimization problem hard or difficult for GA or other heuristics.

In Figure 5-12 (adapted from [Jones and Forrest 1995]), the fitness landscape of studied instances (DTP-***, ATM-***) of the test suite minimization problem are positioned on the fitness distance correlation scale in comparison with other optimization problems.

Another important aspect that is observable from the Table 5-10 is the effect of additional test cases in a test suite. The three test suites with different sizes for a model (e.g. referred as DTP-28, 35 and 42) are not significantly different (worse or better) in terms of ρ (fitness distance correlation). A slight increase in the normalized correlation length ξ(for ATM test suites in Table 5-10) indicates that larger test suites are more favorable to search despite the increase in the search space. However, further investigation is required as it is not clearly evident from the DTP test suites.

Jones and Forrest studied a large set of optimization problems and classified them in three groups using the fitness distance correlation (ρ) metric: (1) problems with $(\rho \geq 0.15)$ are considered to be "misleading" as in such problems, fitness tend to increase with distance from the global optimum, (2) problems within $(-0.15 < \rho < 0.15)$ are considered to be difficult as in such problems there is very little correlation between the fitness and distance from the global optimum, and (3) straightforward problems in which fitness tends to increase as the global optimum is approached and where $(\rho \leq -0.15)$ [Jones and Forrest 1995]. Figure 5-12 is adapted from [Jones and Forrest 1995] and shows the studied instances of test suite minimization problem on Fitness Distance Correlation scale. Although, none of the test suite minimization problem instances seemed to fall in the deceptive or misleading problems category, nonetheless most of them clearly appeared to be difficult. In Table 5-10, the Fitness Distance Correlation ρ values are marked with classification keys such as misleading $'m'$, difficult $'d'$ and straightforward $'s'$ according to this classification.

In order to visually see the effect of test suite shuffling on the fitness landscape of a test suite, the line plots of fitness distance correlation of various versions of test suites were drawn and presented here in Figure 5-11. From these plots it can be seen very clearly that the test suite shuffling had affected the fitness landscape in every case. In some cases the change was little but on the other occasions it was very significant. For example in case of TS-2, TS-3 and TS-4 of ATM-89 there was a very small change but in case of TS-5 it was very significant.

Following are the key results of the fitness landscape analysis for the test suite minimization problem:

1) The landscape of the test suite minimization problem was found to be highly rugged invariably. The short correlation length and far from ideal values for fitness distance correlation were the key indicators (see ρ and ξ in Table 5-10). In NK-model analysis, Kauffman [1993] showed that the fitness landscapes highly influenced by the epistatic interactions between genes or points in the search space. In fully uncorrelated and rugged landscape (at K=N-1), each gene is

affected by all the remaining genes in the search space. In an ideal landscape, each gene contributes to overall fitness independently of all other genes. However, in test suite minimization, the fitness contribution of a bit does not only depend on its own value (0 or 1) but also on the values of earlier enabled bits. This high degree of epistatic interaction in the test suite minimization problem renders its fitness landscape very rugged.

2) The addition of more test cases to a test suite may result in a larger but not necessarily more rugged landscape (see ρ and ξ for ATM-89, ATM-111 and ATM-133 in Table 5-10). Therefore, it is important to note that the change in the size of landscape does not necessarily diminish the performance of a heuristic.

3) Shuffling a test suite generally effects its fitness landscape (as shown in Figure 5-11) and in some cases turned a difficult instance into straightforward problem (for instance, see the classification markings on ρ values for different versions of DTP-28 test suite in Table 5-10).

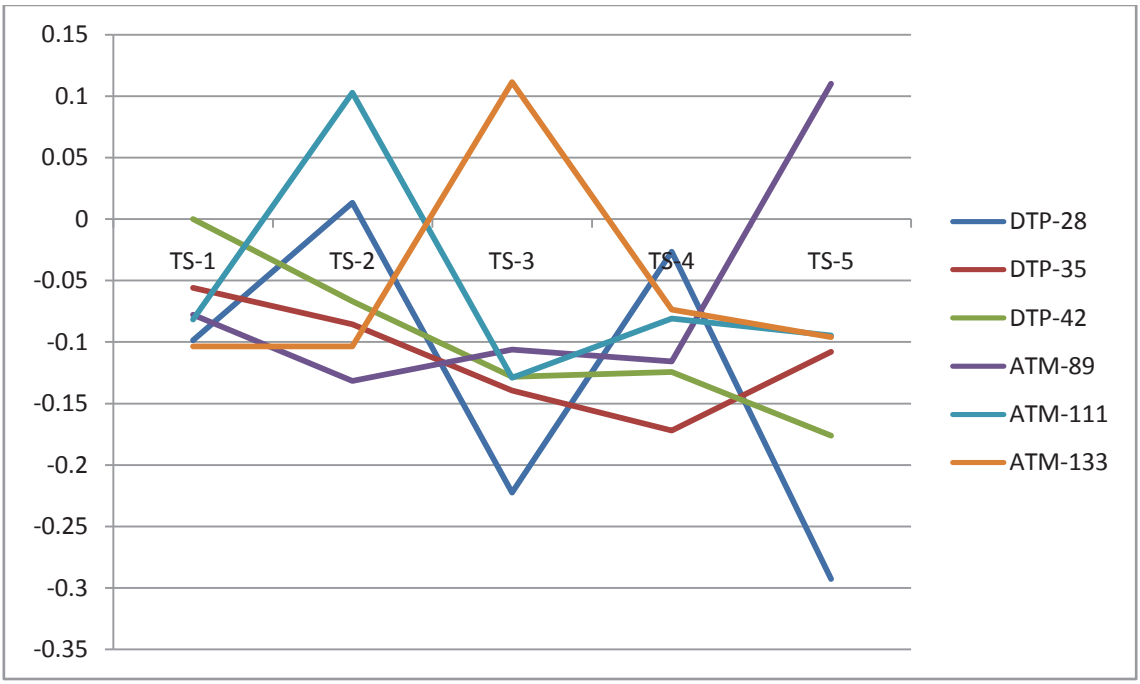


Figure 5-11: Effect of test suite shuffling w.r.t. fitness distance correlation

Table 5-10: Summary of results for fitness landscape of five versions of selected test suites

Test Suite Name	Measure	Test Suite Version				
		TS-1	TS-2	TS-3	TS-4	TS-5
DTP-28	ρ	-0.0985[d]	0.0134[d]	-0.2224[s]	-0.0263[d]	-0.2927[s]
	ℓ	4.2695	4.2497	4.0192	4.2886	4.2418
	ξ	0.1525	0.1518	0.1435	0.1532	0.1515
DTP-35	ρ	-0.0560[d]	-0.0855[d]	-0.1393[d]	-0.1719[s]	-0.1081[d]
	ℓ	5.4712	5.0774	5.0461	5.1030	5.3489
	ξ	0.1563	0.1451	0.1442	0.1458	0.1528
DTP-42	ρ	-0.0564[d]	-0.0669[d]	-0.1282[d]	-0.1243[d]	-0.1762[s]
	ℓ	6.4692	6.6084	6.7645	6.6744	6.5524
	ξ	0.1540	0.1573	0.1611	0.1589	0.1560
ATM-89	ρ	-0.0779[d]	-0.1317[d]	-0.1061[d]	-0.1158[d]	0.1100[d]
	ℓ	16.4441	16.7765	17.2144	17.1629	18.2004
	ξ	0.1848	0.1885	0.1934	0.1928	0.2045
ATM-111	ρ	-0.0816[d]	0.1029[d]	-0.1291[d]	-0.0808[d]	-0.0946[d]
	ℓ	25.8096	24.3688	25.4509	25.4164	24.8516
	ξ	0.2325	0.2195	0.2293	0.2290	0.2239
ATM-133	ρ	-0.1034[d]	-0.1034[d]	0.1113[d]	-0.0736[d]	-0.0959[d]
	ℓ	30.3485	31.1527	30.6404	30.7714	30.7595
	ξ	0.2282	0.2342	0.2304	0.2314	0.2313

$\rho = Fitness\ Distance\ Correlation$, $\ell = Correlation\ Length$ and $\xi = Normalized\ Correlation\ length$ ($\pm 0.02\ standard\ error$).

$According\ to\ the\ classification\ of\ problem\ difficulty\ defined\ in\ [7], a\ problem\ instance\ is\ misleading\ (m)\ if\ (\rho \geq 0.15), difficult\ (d)\ if\ (-0.15 < \rho < 0.15), and\ straightforward(s)\ if\ (\rho \leq -0.15).$

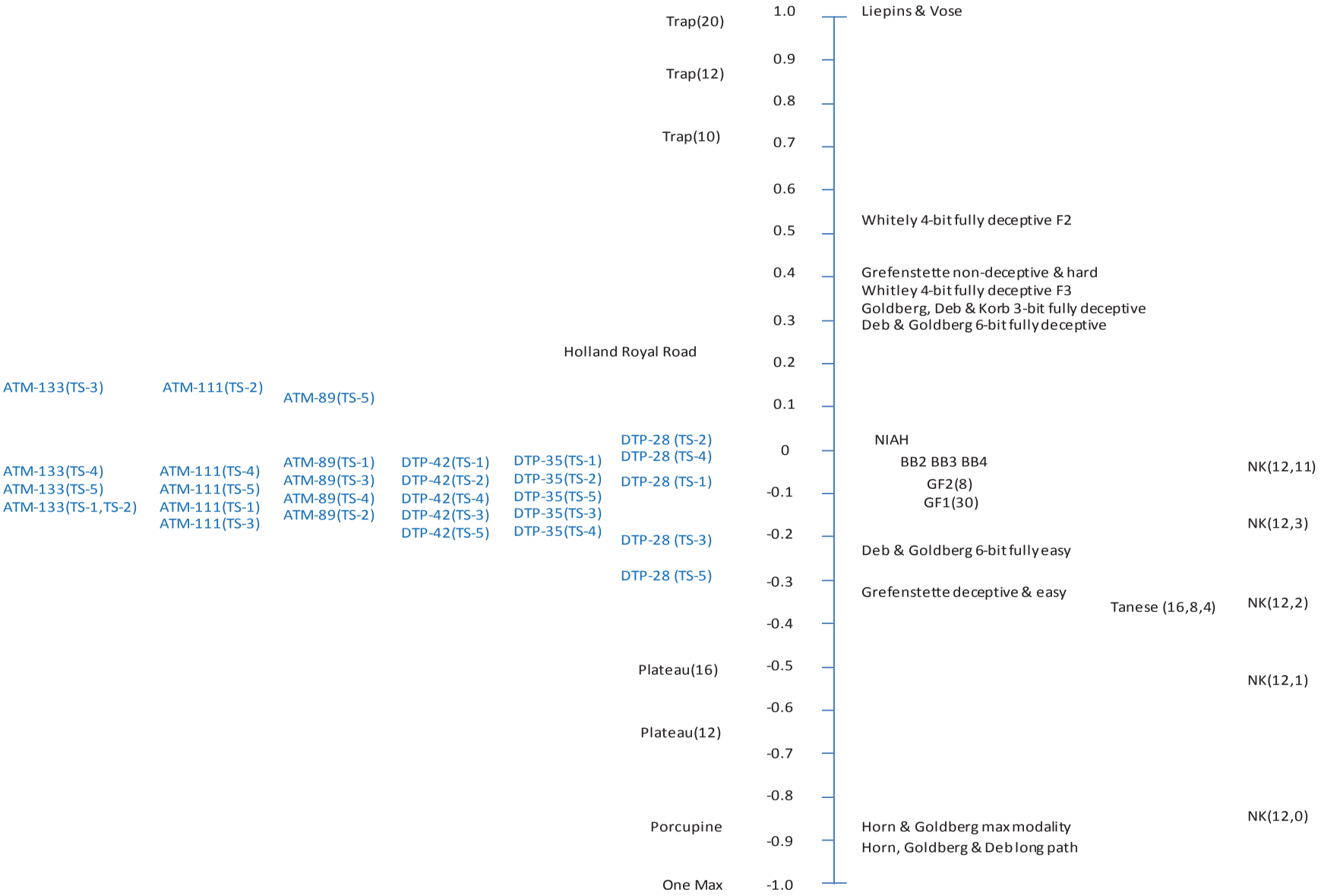


Figure 5-12: Summary view of fitness landscape of TS minimization and other optimization problems on fitness distance correlation scale. Horizontal position is for grouping at that correlation level. Adapted from [Jones and Forrest 1995].

5.5 Threats to Validity

Although, the care was taken in the experimental design, there are some factors that may jeopardize the validity of the experiments and results. These factors are related to internal validity, external validity, construct validity, and conclusion validity [Claes et al. 2000]. Factors that can affect the independent variables without the researcher's knowledge are considered as threat to validity [Claes, Per, Martin, Magnus, Björn and Anders 2000]. Following threats were identified and addressed for the empirical study presented in this chapter.

Internal validity: Internal validity is about the integrity of the experiment and evaluates how well the inference can be made about the causal relationship between the independent and dependent variables. Threats to internal validity include testing, instrumentation, selection of objects, statistical regression, maturation and others. In this study there were two major threats to internal validity. First, there was an instrumentation threat that the difference in the scoring instruments may affect the outcome of different experiments. The instrumentation threat was addressed by using the same fitness or cost function with each of the optimization algorithms to determine the quality of generated solution. Second, statistical regression towards mean is a threat to internal validity that can occur due to extreme values of independent variables. AD models used in the experiments have different size and complexity level. In order to minimize the effect of model size and complexity on the results, as a preventive measure test suites were grouped into homogenous blocks. Given the randomization in test suite generation and shuffling it is assumed that the threat of statistical regression was also very limited.

External validity: Threats to external validity limits the generalization of the experimental results [Claes, Per, Martin, Magnus, Björn and Anders 2000]. The models used in the study represent a mix of software of different sizes and complexities, reduces the threat to external validity. Note that the ECCS model was taken from [Koehler et al. 2005] which was developed by professionals at IBM Zurich Research Laboratory. The ATM model is a typical case study in software engineering research and here it was adapted from [Chandler, Lam and Li 2006]. In this model, it was tried to mimic the real-world scenarios as close as possible. The ETP and DTP models were taken from a large model developed for a commercial product which is being used in transport industry for online and offline reporting of traffic trend. However, the used models were rather small which may restrict the generalizability of the results. Four different heuristic algorithms were used in the experiments. It is important to note that they can be customized which may affect their performance. Therefore, the configurations of these algorithms used in the experiments also limit the generalization of the results.

Construct validity: Construct validity is referred to the degree to which independent and dependent variables accurately measure the constructs they supposed to measure [Claes, Per, Martin, Magnus, Bjöorn and Anders 2000]. Poor construct definition and construct confounding are examples of threat to construct validity. The goal of this study was to empirically assess the performance of heuristic algorithms and the effect of test suite size and composition on test suite reduction. The size and coverage of reduced test suite were considered right indicator of redundancy reduction and therefore valid measures for test suite reduction. Moreover, in order to make meaningful comparison across different groups of test suites percentage of reduction (reduction %) metric was used. Thus, it was considered that the dependent variables had construct validity.

Conclusion validity: Threats to conclusion validity refer to issues that can affect the correct conclusion about relations between independent and dependent variables [Claes, Per, Martin, Magnus, Bjöorn and Anders 2000]. In order to draw valid and accurate conclusions from the study, correct measurements and appropriate statistical tests were used. In order to ensure that measurements were recorded correctly, data was automatically stored in log files. For correct application of statistical tests, it was ensured that none of the associated test assumptions were violated.

5.6 Summary

The chapter introduced the evolutionary computation based minimization technique for model-based test suites. The problem of test suite minimization was formulated as an instance of well known and highly studied knapsack problem. An empirical study was performed to analyze the performance of proposed evolutionary technique with three other optimization techniques. It was found that the proposed evolutionary consistently performed better than other optimization techniques. In order to get a better insight of the test suite minimization problem, its fitness landscape was characterized. Furthermore, the effect of parametric values of the evolutionary technique on test suite minimization was empirically investigated and suitable values were recommended.

Chapter 6 Multi-objective Test Suite Optimization

The previous chapter introduced the evolutionary computation (EC) based minimization technique for model-based test suites. In that chapter, the test suite redundancy removal problem was reformulated as a combinatorial optimization problem and the performance of the EC technique was compared with the Greedy and Hill climbing algorithms. A single criterion (Branch coverage) was used for test suite minimization. Given the differences in test criteria, incorporating other types of possible criteria i.e. usage profile, mutation score and cost into test suite minimization can yield different results. As redundancy is measured relative to the evaluation criteria, it is important to incorporate these criteria during the minimization process in order to avoid losing its aggregated effectiveness. This chapter describes the investigation for multi-criteria minimization of a model-based test suite using multi-objective evolutionary techniques.

6.1 Multi-Objective Model-based Test Suite Optimization

Generally, multiple techniques can be used for software testing depending on the quality, reliability and budgetary considerations of the software project. In safety and mission critical applications, a combination of white-box, black-box and performance testing techniques is mandatory. For instance, according to standard requirement for railway control systems a combination of black-box with performance testing is required for components of the highest integrity level [CENELEC 2001]. Variation in the effectiveness of the testing techniques is one of the factors behind the need for using multiple techniques. In these cases, specification-based testing can determine if the implementation satisfies all the intended requirements. However, paradoxically it is known that the specification based testing is not effective for all types of defects, and conformance can even be demonstrated for implementations that contain faults [Beizer 1990]. Although, code-based testing is considered quite effective in fault detection to the extent that in some industries such as aviation, MCDC coverage is a minimum criterion for software testing [EUROCAE 1992]. However, code-based testing has its own limitations as it fails to detect the omission type of faults [Beizer 1990]. This is due to the fact that code-based techniques derive test cases from code only. Statistical testing is highly recommended for safety-critical systems to provide quantitative measures of quality, reliability and conformance to the

specification [Goel 1985], yet even it cannot ensure that the program does not have any untested partitions. Each technique has its strengths and weaknesses; therefore, it is often recommended that a combination of these techniques be used to achieve better software quality and reliability.

Most of the practical or industrial optimization problems involve more than one decision parameter, and often good tradeoffs are searched for amongst competing constraints. For these types of problems, more than one equally good solution usually exists. Choosing the best one always depends upon the application context. Therefore, in typical multi-objective optimization problems, all of the possible solutions represent some sort of trade-off relationships between the objectives.

The process of finding optimum solution(s) for two or more conflicting objectives simultaneously is known as multi-objective optimization. A multi-objective optimization problem can be resolved as a single objective problem by reformulating it as a constrained problem. In such case one of the objectives are optimized while others are handled as constraints. In the previous chapter, the test suite minimization (TSM) problem is defined as maximizing the effectiveness of a test suite while reducing its cost. It is a dual-objective problem where the one objective is to maximize the effectiveness of a test suite and the other objective is to reduce its cost. However, the problem was reformulated and resolved as a single objective problem by aiming to find a minimal subset of a test suite without compromising its effectiveness adequacy (resolving one objective while restricting the other). In the following section, the same TSM problem is reformulated as the Profitable Tour Problem which is one of the well-known multi-objective optimization problems.

6.1.1 Formulation as a Profitable Tour Problem

The Traveling Salesman Problem (TSP) is one of the most intensely studied problems in combinatorial optimization. In the basic version of the TSP, the objective is to find a most cost-effective round trip path for a given number of cities with the traveling costs between them. Profitable Tour Problem (PTP) is a multi-objective type of TSP with profits (TSPP) and formally defined as follows [Dominique et al. 2005]. Consider $G = (V, E)$ be a complete undirected graph where $V = \{v_1, v_2, \dots, v_n\}$ is a set of n vertices and E is a set of edges. A profit p_i is associated with each vertex $v_i \in V\ (with\ p_1 = 0)$ and a distance c_{ij} with each edge $(v_i, v_j) \in E$. The distance is considered as a travel cost and it is not necessary to visit all vertices. The aim is to find a tour with two conflicting objectives: (1) minimize the travel cost with option to drop vertices and (2) maximize the total profit by visiting more vertices.

In an analogy to the PTP, the dual-objective TSM problem can be defined as finding a subset of a test suite TS with a minimum number of test cases and maximum coverage. For instance, a test suite has n test cases that correspond to cities in PTP. Each test case i has a coverage that corresponds to the profit p_i associated with a city. Like PTP, the selection of a city to visit, a binary variable y_i is used to indicate the inclusion or exclusion of a test case. The traveling cost c_e associated with an edge in PTP implies the utility value of a test case and a binary variable x_e associated with the edge indicates whether the corresponding edge is used in the solution or not. As the objective is to find a subset of test suite TS with maximum coverage at minimum cost, the problem can formally be stated as [Dominique, Pierre and Michel 2005]:

$$minimize \sum_{e \in E} c_e x_e - \sum_{v_i \in V} p_i y_i \qquad (6\text{-}1)$$

The requirements to be satisfied are:

$$\sum_{e \in \delta(\{v_i\})} x_e = 2y_i \quad (v_i \in V)$$

subtour elimination constraints,

$$y_1 = 1,$$

$$x_e \in \{0,1\} \quad (e \in E)$$

$$y_i \in \{0,1\} \quad (v_i \in V)$$

One way of reducing the test suite size is to detect and eliminate test cases that are considered redundant according to a given coverage criterion. However, reducing the same test suite using another different criterion may yield a different solution. For instance, reducing the test suite according to the usage profile of the software will focus mainly on testing the software features that are important to the user. The software features that are rarely used have less impact on customer satisfaction than the features that are in high demand or used regularly. Similarly, in mission critical applications a feature that may execute only once cannot be left untested. Therefore, customer usage data that can leverage the testing effort according to the value of the features from a user's perspective, would add direct value and meaning to the testing. Using this approach, test cases with the least value can be dropped based on the usage patterns of the software. Whilst several other objectives (e.g. setup and execution cost, risk,

failure probability) can be used for test suite minimization, selecting a minimal number of test cases out of all the possible combinations of test cases in a test suite that satisfies the constraints of multiple objectives is harder than the single objective TSM problem.

Given that the multi-objective minimization of a test suite is an optimization problem, the question as to whether test cases can be omitted from execution, in order to improve efficiency and effectiveness of the test suite according to multiple test objectives, is formulated as PTP and defined as follows.

Consider a model based test suite TS of N test cases with cumulative coverage C_{TS}. Each test case x is a sequence of model elements representing an execution path in the model and provides coverage C_x at a utility cost u_x ($where\ x = 1, 2, \dots, N$). The cumulative coverage C_{TS} and test case coverage C_x are the percent values of model elements required by a test criterion that have been executed by the test suite and a test case respectively. The test suite size N is the number of test cases in the test suite. Consider the test suite minimization with an objective of determining a minimal solution subset SS ⊆ TS in such a way that $C_{SS} = C_{TS}$ and cost $u_x = a$. K is a set of all feasible solutions and a feasible solution $k \in K$ is represented by a set $k = (k_1, k_2, \dots, k_N)$ of binary variables k_x such that $k_x = 1$ if test case k_x is included in the solution and $k_x = 0$ otherwise. Using this notion, the objective function can be represented as:

$$f(k) = minimize \sum_{x=1}^{N} u_x k_x$$

Subject to (6-2)

$$\sum_{x=1}^{N} C_x k_x = C_{TS}$$

In general the multi-objective test suite minimization is considered with M objective functions $f_i: K \to \mathbb{R}, 1 \le i \le M$, where the vector function $f := (f_1, f_2, \dots, f_M)$ maps each subset solution $k \in K$ to an objective vector $f(k) \in \mathbb{R}^{\mathrm{M}}$.

For any instance of the problem, the aim is to find the non-dominated solution vector. For M objectives, the dominance relation is defined on feasible solutions K and denoted by Δ. A feasible solution k dominates a feasible solution k', $k \,\Delta\, k'$, if and only if $f_i(k) \ge f_i(k')$, $i = 1, 2, \dots$.

6.1.2 Multi-objective Evolutionary Algorithms

Evolutionary Computation (EC) is a population based meta-heuristic where the search starts from multiple points in the search space and then processes multiple points for each iteration until the specified objective is reached. It means that evolutionary algorithms can explore and produce multiple solutions in single iteration and inherently support parallelism, thus making EC an ideally suitable technique for multi-objective optimization problems. Generally, the multi-objective EC algorithms (MOEAs) are classified into Pareto-based and non-Pareto-based techniques. In non-Pareto-based techniques (e.g. VEGA and Min-Max) prior trade-off relationship or preference among the objectives is defined and used to propagate a globally unique solution. The Pareto-based techniques (e.g. NPGA, NSGA and SPEA) produce multiple distinct solutions known as Pareto optimal (solution) sets without having pre-defined relationship or preference among objectives. A number of Pareto and non-Pareto based MOEAs are described in much greater detail in Chapter 2.

Now in the case of the TSM problem, if the tester has prior knowledge of the relationship among the objectives, then in light of a given preference (i.e. weight, target), the exact trade-off among the objectives can be used to determine the optimal solution. However, if the tester is not sure of the exact relationship among objectives, rather than determining a single approximate solution, a set of Pareto-optimal solutions can be generated initially where each solution represents one possible way of balancing among the objectives. Thereafter, one of these solutions can be chosen, based on some 'posterior' knowledge or consideration of the tester. In situations where time and knowledge is scarce, Pareto-based multi-objective algorithms provide much needed assistance in decision making with various options for optimal solutions. In this study the Pareto-based minimization of model based test suites is investigated and results of empirical study are presented in the following section.

6.2 Empirical Study

In Section 6.1, it is shown that the test suite minimization under multiple objectives is a problem of real practical importance. As there could be more than one equally good solution for such multi-objective problems, it is important to use an optimization technique that can produce a set of solutions rather than a single best solution. The problem of finding various tradeoff solutions while removing the redundant test cases in parallel, is a combinatorial optimization problem. There are several multi-objective evolutionary techniques (e.g. NPGA, NSGA and SPEA) that are commonly used for finding approximate non-dominated solution sets in polynomial time. The performance of these techniques varies depending on the suitability to a problem

instance and parametric values and that need to be determined empirically. Therefore, in this experiment, the objective is to empirically compare two multi-objective evolutionary techniques for the test suite minimization problem in terms of quality of the generated solution.

6.2.1 Research Questions

The following research questions about the effects of the test criteria such as test suite size, test suite coverage and usage profile on test suite minimization were investigated.

Q1: How do the various test criteria in combination affect the test suite minimization?

Q2: How does the test suite size affect the multi-objective test suite minimization?

Q3: How does the usage profile affect the test suite minimization?

Q4: How do the various evolution-based multi-objective optimization techniques perform in terms of the test suite minimization problem?

6.2.2 Experimental Setup

In order to investigate the research questions, the variables that need to be considered are enumerated next. The independent variables are test suite size, optimization algorithms, usage profile and model type. The dependent variables are reduced-size, coverage and weight values. It was expected that the test suites will be homogenous with respect to the source models. So, the test suites were blocked into four uniform groups according to the models, namely ECCS, ETP, DTP and ATM. Within each of our four blocks, experiment was performed with a 4 x 2 x 3 (four test suites of different sizes, two Pareto-based optimization algorithms and three user profiles) Factorial Repeated Measure with Block design. Each factor is elaborated in the following paragraphs.

As the focus of this experiment is TSM by eliminating redundant test cases under different usage patterns, the same set of AD models were used as in the case of single-objective TSM, but they were enhanced with usage profiles. For usage profile based testing, generally empirical data (i.e. usage frequency, the cost and severity of failure) collected from the user is used. A usage profile is specified by a complete set of software functions with their probabilities of occurrence. Generally, four key steps are performed in developing a usage profile viz. 1) identify the user, 2) determine the functions invoked by each user, 3) determine the occurrence frequency of functions identified at step 2, 4) determine the occurrence probabilities by dividing the occurrence frequency with total occurrence frequency. Three different types of users, namely experienced, novice and average (uniform) users were considered. Although other more broad sets of users may be defined, for the purposes of simplicity, only three types of users are

used in this study. As real usage information is not available, similar to another study [Doerner and Gutjahr 2003] the estimated usage pattern based on the potential usage of the processes is used. For annotating usage data on the case study AD models described in Chapter 3, a UML profile as specified in [OMG 2008] is used. The stereotype <<PaStep>> together with its 'prob' attribute is used to annotate the probability of a branch. These annotations on the branches of the decision nodes specify the probability of their execution. For instance, in Figure 6-1, on branch D6-Select, there is a 50% probability specified for adding another product and 50% for proceeding to shopping cart under uniform probability distribution.

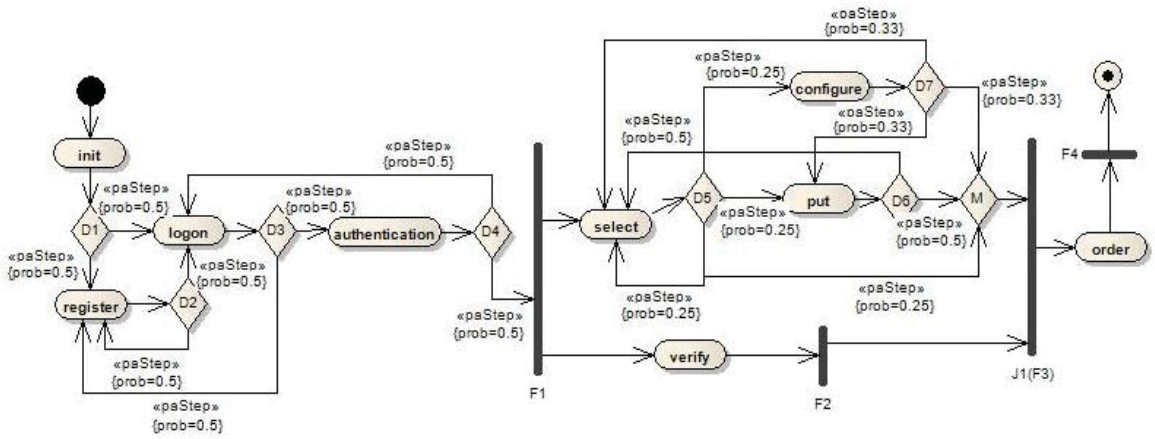


Figure 6-1: ECCS model with usage profile

Test suites of four different sizes are generated for each model using a random walk based test sequence generation (RW-TSG) technique described in Chapter 4. The generated sequences of model constructs, formally referred to as paths are evaluated according to the associated branch coverage criterion proposed in Chapter 4. It is assumed that in the generated test suites there are multiple test cases of varying length covering one or more branches.

Table 6-1 presents the summary of a generated test suite according to Branch Coverage per Test Case (BCTC) and Test Cases per Branch (TCB) metrics. For instance, each test case for the ECCS model covers more than 47% of the branches on average as shown in the 'mean' column of Branch Coverage per Test Case (BCTC). The standard deviation 'S.D.' column indicates the variation in test cases in a test suite. The Test Cases per Branch (TCB) metric indicates the redundancy level of a test suite and as shown in Table 6-1 on average is more than one test case for a branch that can execute it.

Two multi-objectives Pareto-based EC algorithms, namely NSGA-II and NPGA2 were used in the experiment. The detail of these algorithms and evolutionary computation in general is presented in Section 3 of Chapter 2. The same set of evolutionary operators and parameters

were used here as were used in the single objective optimization (Section 5.3.2). For the search space encoding, binary strings were used and the initial population of the candidate solution was produced randomly. For the selection of the parents, tournament selection was used. In NPGA, linear ranking is used with a selection pressure of two individuals per generation.

Single point crossover with the probability of 0.9 was used. Single point mutation of each individual is used with 0.01 rate of occurrence. The maximum generation for both algorithms varies with test suite size e.g. the maximum generations for the ECCS-20 and ATM-300 problems are 100 and 400 respectively. Given the stochastic nature of the optimization algorithm, a small variation is expected in the results. Therefore, the experiment was designed to run for 10 iterations of each algorithm. One of important features of multi-objective optimization is that it does not need scaling or normalization of objectives. However, for comparative analysis of optimizers and generated solutions it is necessary to scale or normalize different objectives. For normalization, the following procedure as defined in [Deb 2001] was used:

$$z_i' = \frac{z_i - z_i^{min}}{z_i^{max} - z_i^{min}} \qquad (6\text{-}3)$$

Where z_i^{max} and z_i^{min} are known or estimated maximum and minimum positive values of $i - th$ objective respectively. Moreover, all objective vectors were transformed in such a way that all objectives were set to be minimized.

Table 6-1: Composition of generated test suites

Model	Test Suite Size	Branch Coverage per Test Case		Test Cases per Branch		Weight per Test Case					
						PF1		PF2		PF3	
		Mean	S.D.	Mean	S.D.	Mean	S.D.	Mean	S.D.	Mean	S.D.
ECCS	20	49.12	13.40	9.80	5.30	39.75	10.48	42.75	9.65	46.10	9.03
	28	50.00	11.15	13.35	7.84	41.96	9.54	44.20	9.03	47.89	8.49
	37	47.85	11.36	17.94	10.62	40.14	9.61	42.74	9.15	46.16	8.46
	50	47.53	11.45	23.76	13.85	39.65	9.57	42.47	9.40	45.70	8.17
ATM	89	10.88	8.16	9.68	11.00	13.79	10.05	13.74	10.79	11.74	12.06
	150	12.05	8.34	15.86	19.43	15.22	10.27	15.20	10.75	13.29	11.78
	200	11.50	8.09	21.50	26.01	14.55	9.95	14.50	10.53	12.54	11.65
	300	10.86	7.87	32.57	38.71	13.78	9.66	13.65	10.35	11.61	11.25
ETP	27	17.24	13.80	4.65	4.31	32.41	16.61	33.63	19.67	33.63	22.51
	45	22.91	14.65	7.92	7.05	28.33	17.93	29.16	20.62	30.20	24.06
	72	19.39	13.91	12.69	11.24	24.06	16.94	24.35	19.17	24.78	22.64
	90	18.21	13.63	16.38	13.64	22.64	16.58	22.83	18.77	23.02	22.16
DTP	28	33.14	9.91	9.30	6.80	59.64	17.84	60.71	21.84	54.50	23.27
	43	31.65	9.83	12.64	7.67	56.98	17.70	57.81	21.46	51.63	22.33
	76	30.34	10.44	22.19	12.62	54.61	18.79	55.22	22.43	49.67	23.55
	104	29.17	9.88	30.33	17.21	52.50	17.79	52.99	21.19	47.21	22.31

- PF1- Profile 1, PF2 – Profile 2 and PF3 – Profile 3

- S.D. – Standard Deviation

6.2.3 Metrics

To quantify the objectives, the following metrics to observe and evaluate the effects of variables (i.e. model type, test suite size and algorithm type) were used in this experiment. For test suite efficiency and efficacy, the metrics defined in the previous chapter (Section 5.2.1) were used. For comparison of multi-objective optimization algorithms, metrics specified in Chapter 2 were used.

As the objective of test suite minimization was to find the trade-off between the effectiveness and efficiency of the test suite, it was pertinent to evaluate the performance of the optimization algorithms by the solutions they produced. The optimization techniques used in this experiment were Pareto-based, so they produced a set of non-dominated solutions also known as Pareto-optimal sets at each generation. An important assumption about these Pareto-based techniques was the absence of prior preference information about the objectives, which meant that each solution in the Pareto-front was equally as good as the others.

In order to statistically analyze the apparent effect of test suite size on multi-objective test suite minimization [IBM 2010], following two measures, ONVGR and ER metrics were used to quantify the solution sets in terms of finding and missing the true Pareto front respectively. Both metrics has been defined in Chapter 2. In order to apply these metrics, it needs to know the true Pareto front. However, given the fact that the true Pareto fronts are generally unknown, Knowles suggested two different approaches for using reference Pareto front [Knowles et al. 2006; Knowles 2002]. According to the first approach all generated non-dominated solution sets produced by algorithms under consideration are combined and used as reference sets. The second approach is by using a median reference set that dominates 50% of the solutions of a large sample (e.g. 1000) produced through a random search. In this study, the first approach was followed and a reference Pareto front Pf_{ref} was created by combining the best unique solution vectors produced for all test suites of a particular model. Using this approach, the reference Pf provided the maximum spread and diversity to the Pareto front and allowed us to measure the quality of a solution Pf, Pf_{sol}, in terms of spread and compliance to reference Pf. The ONVGR was computed as the coverage of reference Pf, Pf_{ref}, by solution Pf Pf_{sol}. Whereas the ER was calculated as the ratio of missing parts in the Pf_{sol} to the total number of solutions in Pf_{ref}. Statistical analysis was conducted for examining the optimization effects, their statistical significant and correlation analysis using SPSS [IBM 2010]. The Tukey HSD analysis of variance (ANOVA) was performed to address the question of whether there are meaningful and statistical differences in multi-objective test suite minimization among the

heuristic algorithms under the influence of constant factors known to affect the performance. Nonparametric Mann-Whitney U test was performed when the assumption of Tukey HSD test were not satisfied. To evaluate and compare the performance of optimization techniques in terms of test suite minimization, the *S*-metric was used and the results were presented using a box plot.

6.2.4 Result and Discussion

Multi-objective Test Suite Minimization: The results of the experiment (presented in Table 6-2) confirmed our premise that a test suite can be reduced with respect to multiple objectives concurrently, by eliminating the redundant test cases and without compromising its effectiveness. In order to see the effects of multiple objectives on test suite minimization, firstly the data was graphically examined. Figure 6-2 and Figure 6-3 illustrate the Pareto fronts with typical cross axes objectives representation on scatter plots for test suite ECCS-20 with 2 and 3-objectives respectively. 3-objective Pareto front is further represented with a matrix scatter plot in Figure 6-4 for pair-wise interaction. In the 2-objectives test suite minimization (TSM-2), the test suite coverage and size were used for test suite minimization, whereas in the 3-objectives test suite minimization (TSM-3), the usage profile was incorporated in addition to test suite coverage and size. In Figure 6-2, the data points with labels {A, B, C, D, E} and {v, w, x, y, z} show the optimal test suite solutions produced by NSGA-II and NPGA2 techniques respectively. Data points {E, D, y, z} are the solutions with minimum size and maximum (complete) coverage level in the Pareto front. As the other solutions on the Pareto fronts (labeled as 'A', 'v', 'B', 'w' and 'C', 'x' in Figure 6-2) with subsets of test cases were less effective than the original test suite (e.g. test suite depicted as 'C' was even smaller than the 'E' and 'D' test suites but coverage was clearly compromised), they could be ignored based on posterior knowledge. Figure 6-3 shows the Pareto front of 3-objective test suite minimization on 3D scatter plot. Similarly for 3-objective TSM, the data points marked as 'X' in Figure 6-4 indicate the different solutions of reduced test suite on Pareto front without any loss of coverage. In order to ensure that these results were statistically significant, a Paired Samples t-test was performed for solutions with complete coverage and results are presented in Table 6-2. There was significant difference between the $Size_O$ and $Size_R$; $t(18452) = 127.585$ and $p = .000$. This confirmed that the generated non-dominated solutions sets (Pareto fronts) for both TSM-2 and TSM-3 included the solutions, which were (1) significantly more efficient than the original test suites, and (2) not inferior in terms of effectiveness.

Effect of Multiple Objectives on Test Suite Minimization: From the tester's perspective, the wider Pareto front is more insightful as it can reveal the differences in the test cases which

could otherwise be overlooked. Another advantage is that it will help the tester to select and execute a test suite among others, which despite having similar quality according to one objective is more effective and valuable from another objective. For instance, finding the minimal subset of test cases with two objectives can yield more than one equally optimal solution, as in the case of solutions 'D' and 'E' in Figure 6-2. However, reducing the same test suite with three objectives revealed a significantly larger and diverse Pareto front. For comparison see the solution sets for ECCS-20 with 2 and 3-objectives in Figure 6-2 and subplot (size-coverage) in Figure 6-4 respectively. Similar phenomenon was observed for even larger test suites as shown in Figure 6-5 and Figure 6-7. The observation provides some insight to our research question Q1, in that incorporating more test criteria (as optimization objectives) in TSM gives the user more options of optimal test suites with varying degrees of efficiency and efficacy.

Table 6-2: Paired Sample t-test between the original test suite size $Size_O$ and the reduced test suite size $Size_R$.

	Paired Differences					
			95% CI of the Difference			
Pair	Mean	Std. Error Mean	Lower Bound	Upper Bound	t	df
$Size_O - Size_R$	51.91	.407	51.115	52.710	127.585	18452

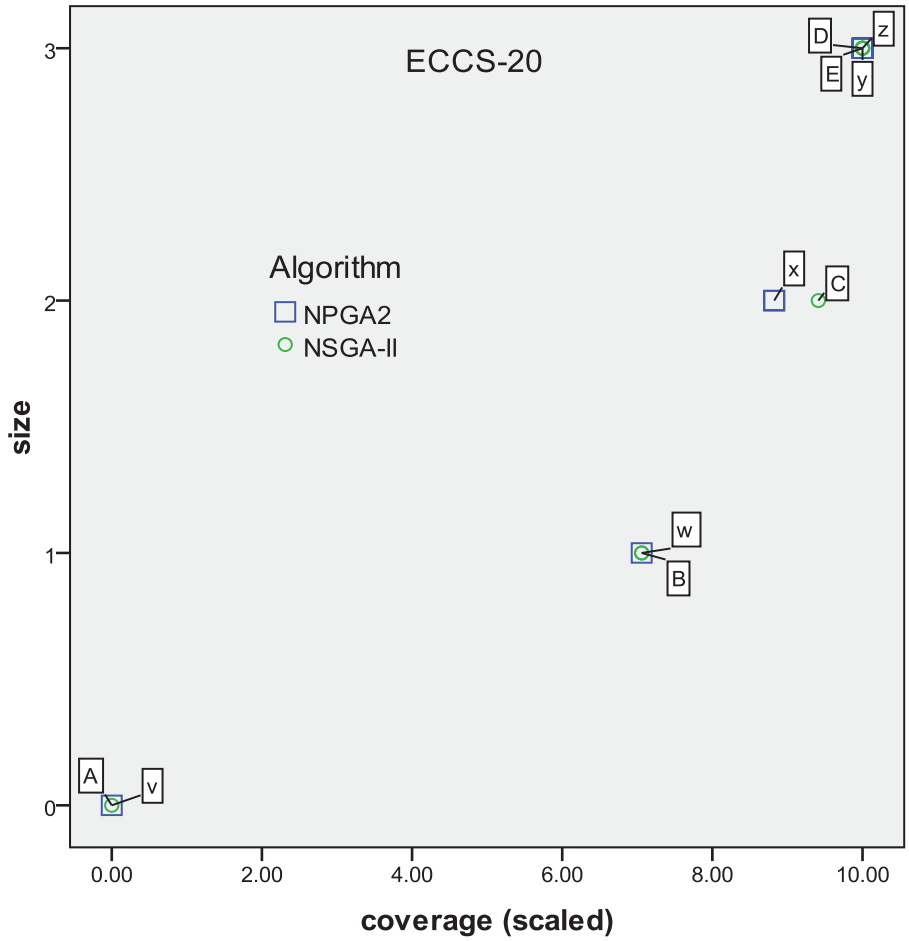


Figure 6-2: Minimization of ECCS-20 test suite with 2-objectives; Data points labeled with capital letters are NSGA-II results and small letter are related to the Pareto-front produced by NPGA2.

In order to determine whether the results were statistically significant, the Independent Samples t-test at the 0.05 level of significance was performed with two groups (2 and 3 objectives) of data sets comprising of sixteen test suites with ten replications each using two optimization algorithms and the result are presented in Table 6-3. There was significant difference between the 2-objectives TSM (NO_2) and 3-objectives TSM (NO_3); t(41657.73)=303.190, p=.000. This confirmed the significantly large increase in the potentially unique trade-off solutions for the reduced test suite when optimized with three objectives as compared to two objectives.

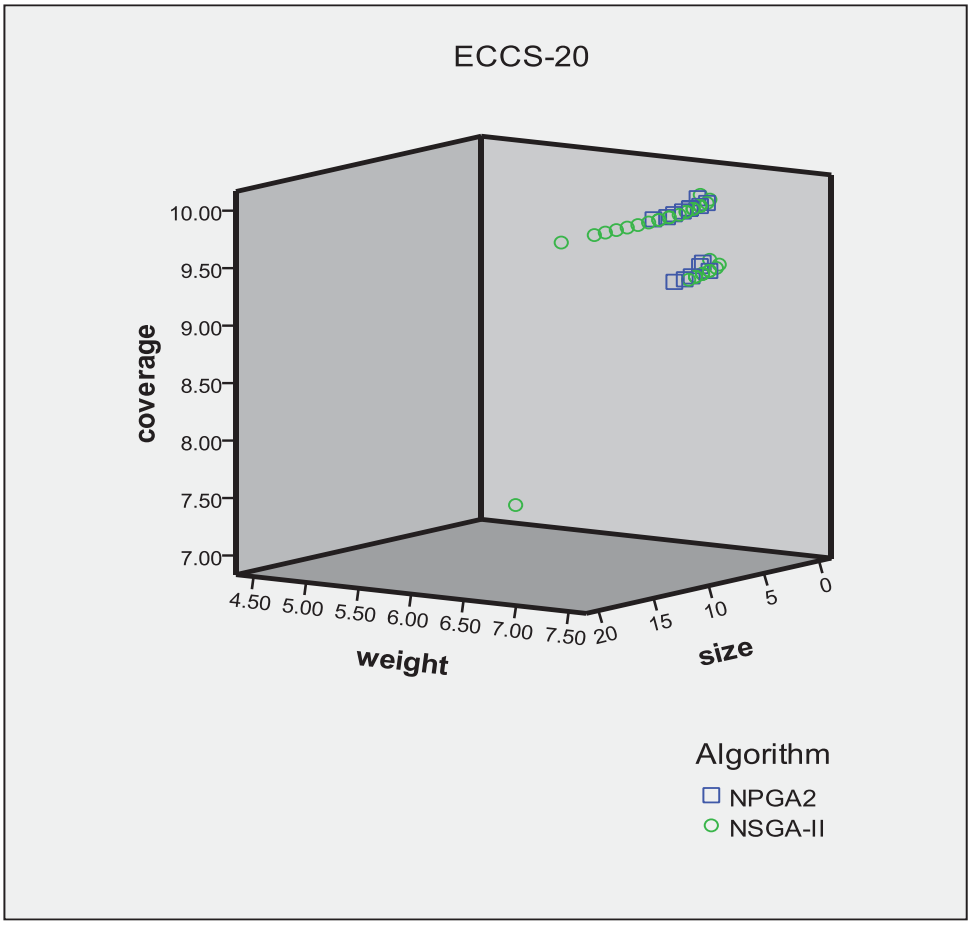


Figure 6-3: 3D scatter plot of ECCS-20 test suite minimization with 3-objectives. A combination of higher coverage, smaller size and more weight indicate a better trade-off solution.

Table 6-3: Independent Sample t-test between the Pareto front size with 2 & 3 objects test suite minimization.

	Differences					
		Std. Error	95% CI of the Difference			
Groups	Mean	Mean	Lower Bound	Upper Bound	T	df
$NO_3 - NO_2$	39.298	.130	39.044	39.522	303.190	41657.731

- NO_2 – Two number of objectives, NO_3 – Three number of objectives

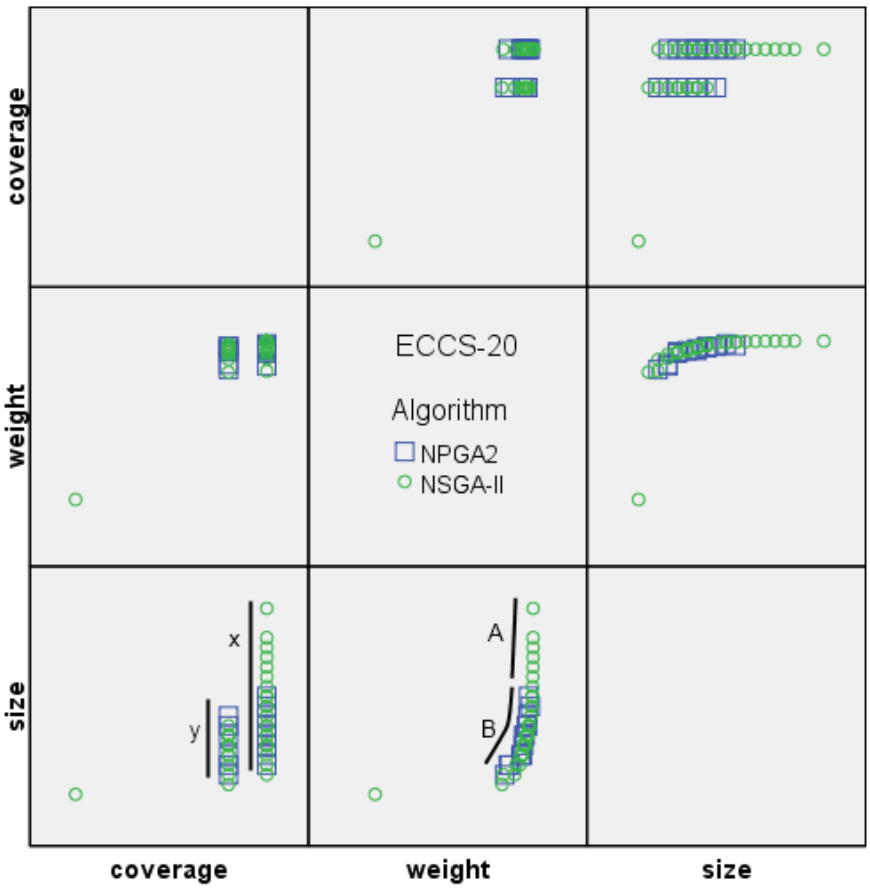


Figure 6-4: ECCS-20 test suite minimization with 3-objectives viz. weight of a test suite w.r.t. a usage profile, coverage as AD-based branch coverage and size as number of test cases in a test suite on Matrix scatter plot.

An important question arose about the cause(s) of wider and diverse Pf for TSM-3 in comparison to TSM-2. Initially, it seemed that the size of a Pf was proportional to the number of objectives. However, further analysis revealed that another reason for a wider Pareto front was the scale and uniformity of the measure associated with a particular objective. For instance, in two objectives TSM, there were only five solutions in the Pareto front (see the Pareto front generated by NSGA-II in Figure 6-2). Whereas in case of three objectives TSM the generated Pareto front was wider and more diverse, as shown in Figure 6-3. By plotting the Pareto front associated with the three objectives TSM in scatter plot matrix as shown in Figure 6-4, the following characteristics was observed. First, the non-uniformity in the coverage level of solution test suites resulted into two distinct clusters in the Pareto front (see 'x' and 'y' in subplot (size, coverage) and subplot (weight, coverage) in Figure 6-4). Second, the weight measure of test suite solutions was more uniform than coverage and third, the size measure was most uniform and finely scaled. For instance, in subplot (size, coverage) it shows that there were three coverage levels (x, y and a single point in the left bottom corner indicating 100%, approximately 92% and 0% coverage respectively), however, there was a test suite of almost every size with

100% coverage which reflected in large PF of test suites and vertically it shows all the sizes of ECCS-20 test suite that can achieve those three coverage levels. Similar effect can be observed in the subplot (size, weight).

The effect of interaction between objectives (criteria) with coarse scale and non-uniform measure was clustered and a narrow Pareto front as shown in subplot (weight, coverage) in Figure 6-4. The interaction between objectives with fine scale and non-uniform measure resulted in a clustered but wider Pareto front (e.g. see subplot (size, coverage) of Figure 6-4). The interaction between objectives with fine scale and uniform measure yielded wider and uniform Pareto front as shown in subplot (weight, size) of Figure 6-4. Thus, it can be inferred that the uniformity and scale of the measure associated with a particular test criterion (optimization objective) played critical role in the width and diversity of a Pareto front.

Effect of Test Suite Size on Multi-objective Test Suite Minimization: Figure 6-5 and Figure 6-6 depict the Pareto front solutions for four test suite versions for the ECCS model with 2 and 3-objectives on a two-dimensional plot matrix respectively. They depict the effects of test suite size on TSM-2 and TSM-3 and for illustration only one solution set for each test suite size produced with NSGA-II is shown. There was some effect of test suite size on multi-objective TSM, for instance, in TSM-2 (Figure 6-5) the $Pf_{ECCS-50}$ (Pareto front solution produced for ECCS-50) was wider than the Pareto front solutions produced for other test suites such as $Pf_{ECCS-20}$, $Pf_{ECCS-28}$ and $Pf_{ECCS-37}$. Similarly in TSM-3 (Figure 6-6) the $Pf_{ECCS-50}$ is clearly the widest (highest cardinality) of all. Although, it appears that the Pareto front solution generated for the ECCS-50 test suite is wider (higher cardinality) than that of the smaller test suites, this effect is not reflected consistently with the increase in test suite size. For instance, in TSM-2 (Figure 6-5) the Pareto fronts generated for ECCS-20, ECCS-28 and ECCS-37 were very similar (same cardinality on phenotypic space), nevertheless in the case of TSM-3 (Figure 6-6) the increase in test suite size also resulted in an extension in the Pareto front to some extent.

As mentioned earlier, the spread, distribution and the proximity to true Pareto front Pf_{true} are the three key criteria for evaluating a Pf solution Pf_{sol}. Although, the larger test suites resulted in wider (higher cardinality) Pareto front, in some cases the solution Pf Pf_{sol} of larger test suites partly or completely missed the true Pf Pf_{true} as observable in Figure 6-5 (see the triangle marker is slightly outside of the values which are part of the true Pf). Nevertheless, with TSM-3 (Figure 6-6), the solution Pf for ECCS-50 $Pf_{ECCS-50}$ has the maximum coverage of the true Pf.

Overall Non-dominated Vector Generation Ratio (ONVGR) and Error Ratio (ER) metrics are used to evaluate the convergence and deficiency properties of a produced Pareto front PF_{prod} with respect to the reference Pareto front PF_{ref}.

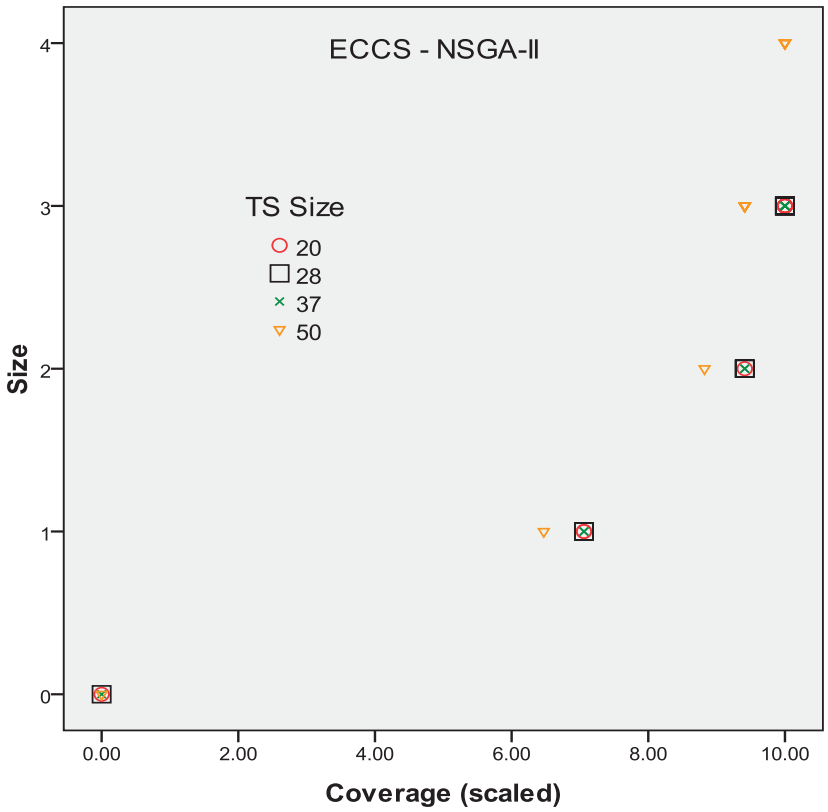


Figure 6-5: Effects of test suite size on test suite minimization with 2-objectives

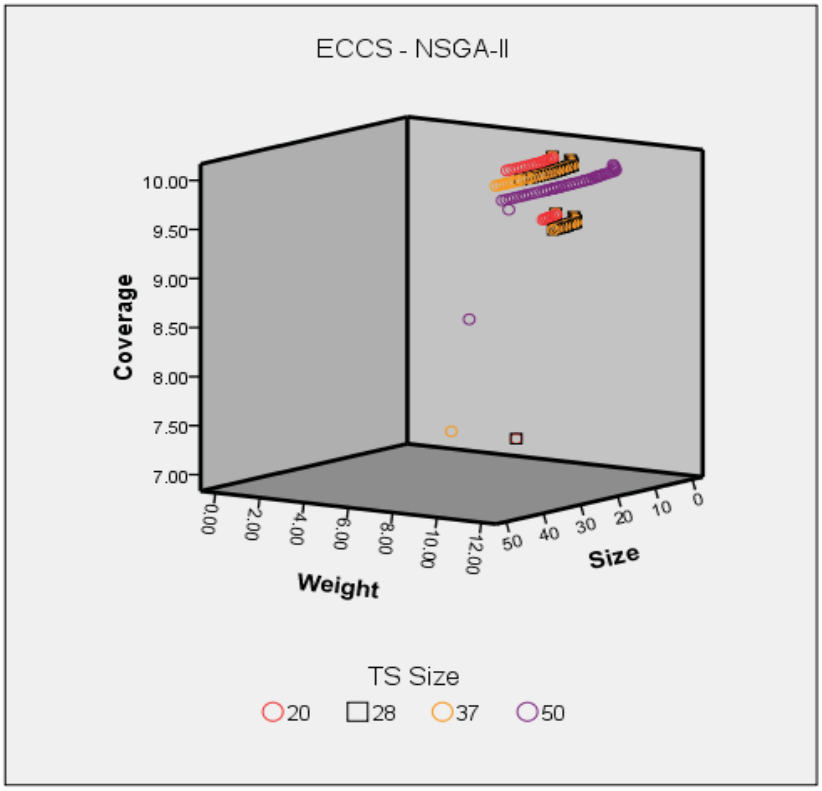


Figure 6-6: Effects of test suite size on test suite minimization with 3-objectives on 3D scatter plot

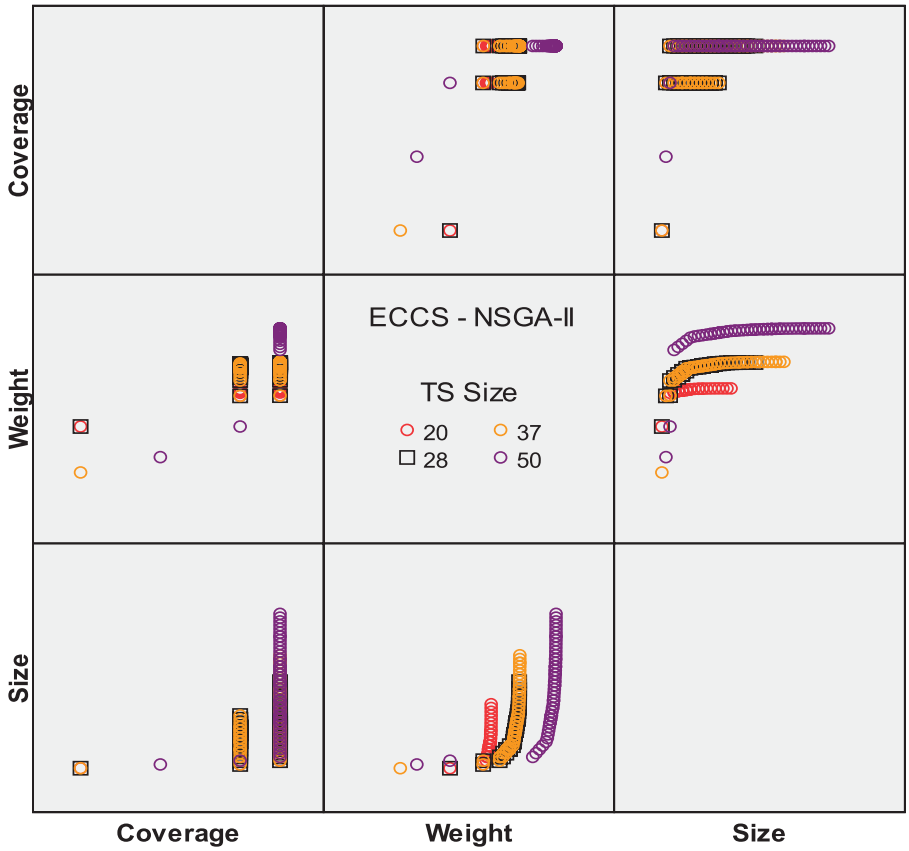


Figure 6-7: Effects of test suite size on test suite minimization with 3-objectives on Matrix scatter plot

Figure 6-8 and Figure 6-9 illustrate the average effect of test suite size on multi-objective TSM with ONVGR and ER metrics. From the mean ONVGR results of Pf solutions generated with NSGA-II algorithm for all test suites as shown in Figure 6-8, some degree of influence of test suite size on multi-objective TSM is seen. For instance, see the mean ONVGR gradient of NSGA-II (green line) from ECCS-20 to ECCS-28 which indicates the increase in coverage of Pf_{true}. Again some degree of rise from ECCS-28 to ECCS-37 is noticeable and then a drop in coverage of Pf_{true} from ECCS-37 to ECCS-50. Although this phenomenon was evident for ECCS, ATM and ETP test suites, it was not apparent in the case of the DTP test suites. In order to determine whether this effect was statistically significant, Tukey-HSD Post Hoc test at 95% confidence interval was performed and results are presented in Table 6-4. As the results with NPGA2 algorithm are clearly insignificant, the Post Hoc test was only applied to NSGA-II related data. The pair-wise comparison of test suites revealed that: (1) the change in test suite size was not reflected significantly across all test suite groups consistently, which means that the increase in test suite size may not yield a higher coverage of the true Pareto front. Moreover, it means that there was no linear relationship between problem size and Pf distribution, (2) there was some statistically significant improvement (higher cardinality) in Pareto front with the increase in test suite size but after a certain level of the change in test suite size, this phenomena did not reflect positively on the coverage of the Pf_{true}. It is found that after a certain threshold of problem size multi-objective EC algorithms may need parametric re-tuning but for meaningful recommendation it will need further experiments.

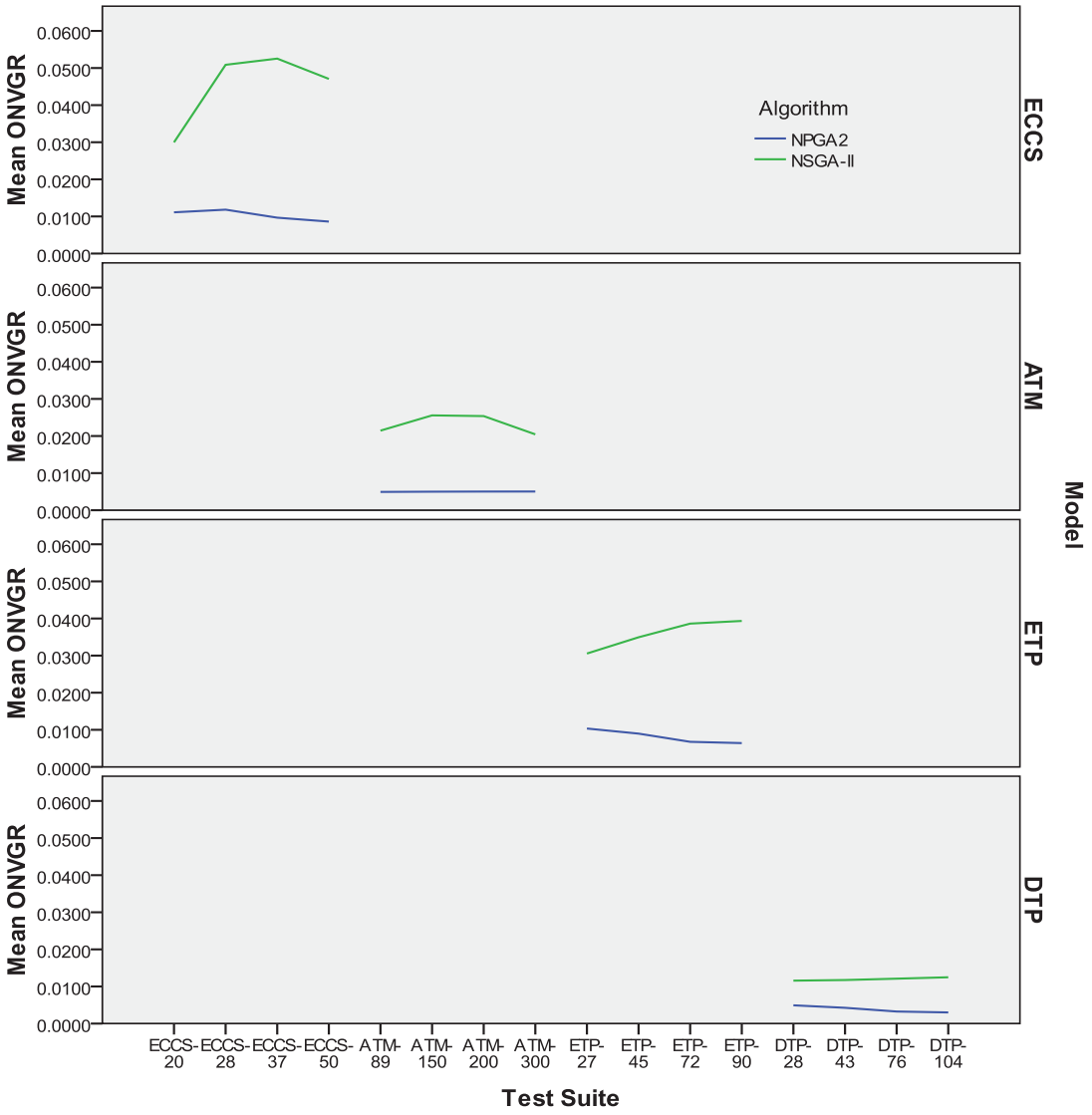


Figure 6-8: Illustration of test suite size effect on Pareto front with ONVGR measure

In case of the DTP test suites, the increase in test suite size did not show an improvement in the Pf but it did not show any loss either, in terms of coverage of the Pf_{true}. The visual analysis of Figure 6-9 reveals that the increase in test suite size did not increase the error rate (diminishing effect). Therefore, even with larger search spaces, the Pfs produced for larger test suites (ATM and DTP) were not worse than the Pfs produced for smaller test suites.

It is important to note that both ONVGR and ER metrics used in this study were computed at phenotypic level. Therefore, there was a limitation with these metrics that two solutions were not different at phenotypic level even if they were different at genotypic level. This is because generally one point in genotypic space can be mapped to multiple points in phenotypic space. It means that no matter how diverse the solution sets (Pareto front) for larger

test suites at genotypic level (different combination of test cases) are, they will not be treated differently in terms of these two metrics.

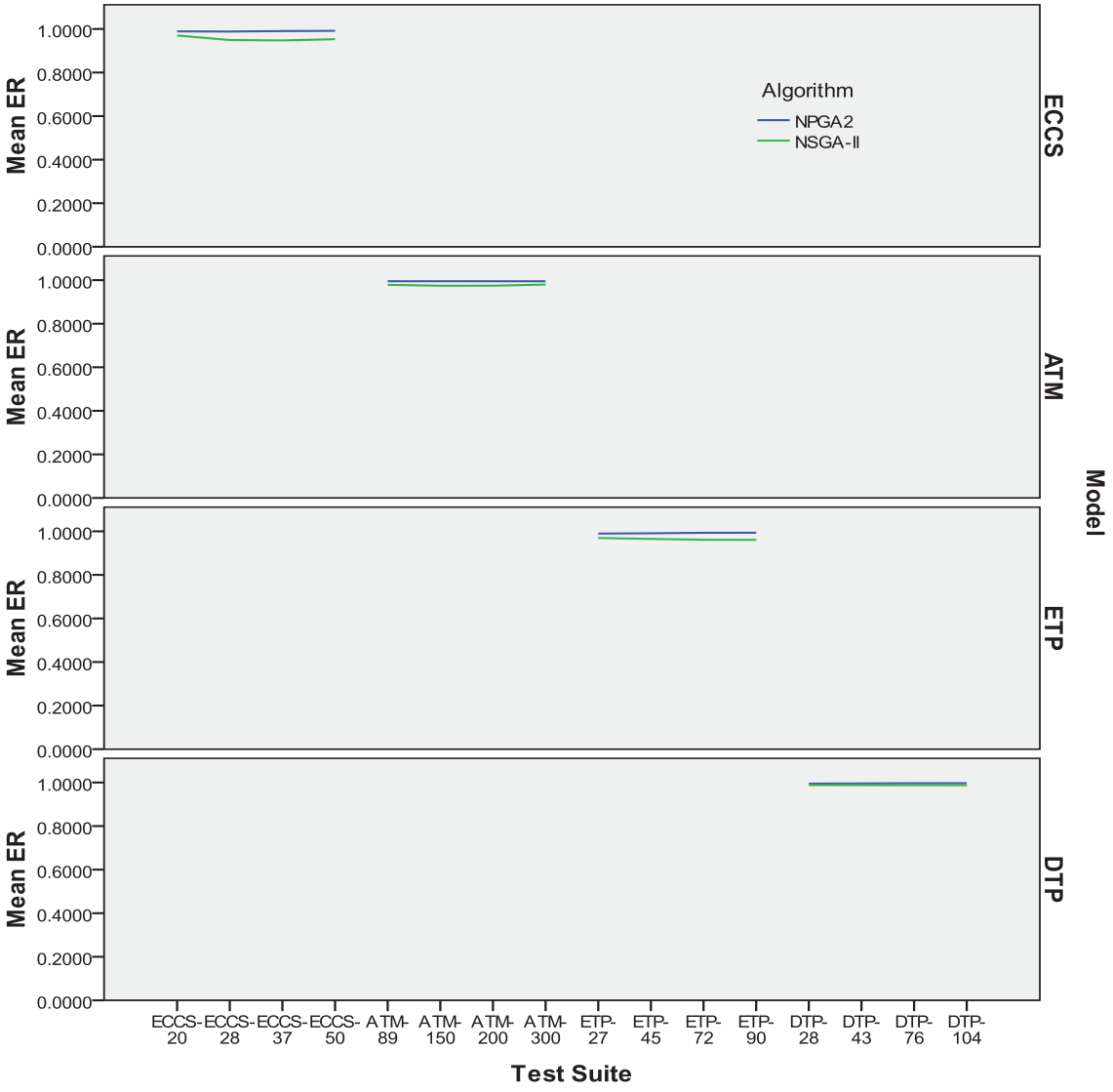


Figure 6-9: Illustration of test suite size effect on Pareto front with ER measure

Effect of Optimization (Multi-objective Evolutionary) Techniques: From Figure 6-8 and Figure 6-9 it appears that the NSGA-II performed consistently superior to NPGA2 for both the two and three objectives TSM in terms of the closeness to the reference Pareto-optimal front, diversity and spread of the generated solution set. Although the NPGA2 produced a good diverse set of solutions, it failed to find a better coverage of the reference Pareto-optimal front. This observation was substantiated by the S-metric analysis as follows.

Table 6-4: Tukey HSD pairwise comparison between Test Suites in terms of ONVGR to reference Pf

Comparisons	Mean ONVGR Difference	Std. Error	95% Confidence Interval	
			Lower Bound	Upper Bound
ECCS-20 vs. ECCS-28	-.020897*	.0006020	-.022972	-.018822
ECCS-20 vs. ECCS-37	-.022563*	.0006020	-.024638	-.020488
ECCS-20 vs. ECCS-50	-.017097*	.0006020	-.019172	-.015022
ECCS-28 vs. ECCS-37	-.001667	.0006020	-.003742	.000408
ECCS-28 vs. ECCS-50	.003800*	.0006020	.001725	.005875
ECCS-37 vs. ECCS-50	.005467*	.0006020	.003392	.007542
ATM-89 vs. ATM-150	-.004113*	.0006020	-.006188	-.002038
ATM-89 vs. ATM-200	-.003923*	.0006020	-.005998	-.001848
ATM-89 vs. ATM-300	.001003	.0006020	-.001072	.003078
ATM-150 vs. ATM-200	.000190	.0006020	-.001885	.002265
ATM-150 vs. ATM-300	.005117*	.0006020	.003042	.007192
ATM-200 vs. ATM-300	.004927*	.0006020	.002852	.007002
ETP-27 vs. ETP-45	-.004383*	.0006020	-.006458	-.002308
ETP-27 vs. ETP-72	-.008063*	.0006020	-.010138	-.005988
ETP-27 vs. ETP-90	-.008783*	.0006020	-.010858	-.006708
ETP-45 vs. ETP-72	-.003680*	.0006020	-.005755	-.001605
ETP-45 vs. ETP-90	-.004400*	.0006020	-.006475	-.002325
ETP-72 vs. ETP-90	-.000720	.0006020	-.002795	.001355
DTP-28 vs. DTP-43	-.000170	.0006020	-.002245	.001905
DTP-28 vs. DTP-76	-.000533	.0006020	-.002608	.001542
DTP-28 vs. DTP-104	-.000910	.0006020	-.002985	.001165
DTP-43 vs. DTP-76	-.000363	.0006020	-.002438	.001712
DTP-43 vs. DTP-104	-.000740	.0006020	-.002815	.001335
DTP-76 vs. DTP-104	-.000377	.0006020	-.002452	.001698

* $p < 0.05$

The size of the dominated space metric (*S*-metric), which is generally used to compute both the convergence and spread of the generated non-dominated solution sets with respect to a reference point. An S-metric box plot for each Pareto-based optimization algorithm with respect to 2 and 3 objectives is presented in Figure 6-10 and Figure 6-11. The column of box-plots on the left summarized the results for NPGA2 algorithm. The column on the right depicts the NSGA-II algorithm and the eight box-plots in each figure depict the results for each of the test suite based on the model types (ECCS, ATM, ETP, DTP) respectively. The x-axes represent the test suites with respect to the model types. The y-axes represent the S-values (the size of the space covered) measured for each algorithm. Here, box plots show the distribution of the

results. A central box shows the middle 50% of the data. The black bar in the middle of a box represents the median value, the whiskers show the range of the data, upper whisker indicates the maximum value and lower whisker shows the minimum value (except for the outliers).

The box plots in Figure 6-10 and Figure 6-11 compare the two algorithms in terms of TSM problem and display two important characteristics that are of interest: 1) in general the NSGA-II perform relatively better than the NPGA2 for the TSM problem, and 2) the TSM with 3-objectives in all test suite types and sizes, have an S-value with about 50% of its readings being within 1 unit. These findings provide an answer to our research question Q4. The S-value, indicating the overall quality of the produced Pf with respect to the reference Pf is used to compare the performance of NSGA-II and NPGA2 for TSM. From the visual comparison of the results for the 2-objectives TSM, it can be observed that the NSGA-II in most cases has a smaller centre box and the results are skewed to the bottom indicating minimal variation in performance as compared to NPGA2. Similarly, the box plot for the 3-objectives TSM shows negligible variation and a more stable performance. In order to determine that the results were statistically significant, the Mann-Whitney U test was applied owing to the presence of outliers in the data. Test was performed at 95% confidence interval with two groups (NPGA2 and NSGA-II) of data sets comprising of sixteen test suites with ten replications each. The median S-values for NSGA-II and NPGA-2 for 2-objectives TSM were 8.3077 and 11.4444 respectively. There was significant difference in the performance of NSGA-II and NPGA2 in relation to 2-objectives TSM; $U = 531898, p < .05\ two\text{-}tailed$. The median S-values for NSGA-II and NPGA-2 for 3-objectives TSM were 36.336 and 47.1739 respectively and similar performance difference between the two algorithms was found for 3-objectives TSM with $U = 70709340.5, p < .05\ two\text{-}tailed$. This confirmed the statistically significant differences between the two algorithms for both the 2 and 3-objectives TSM.

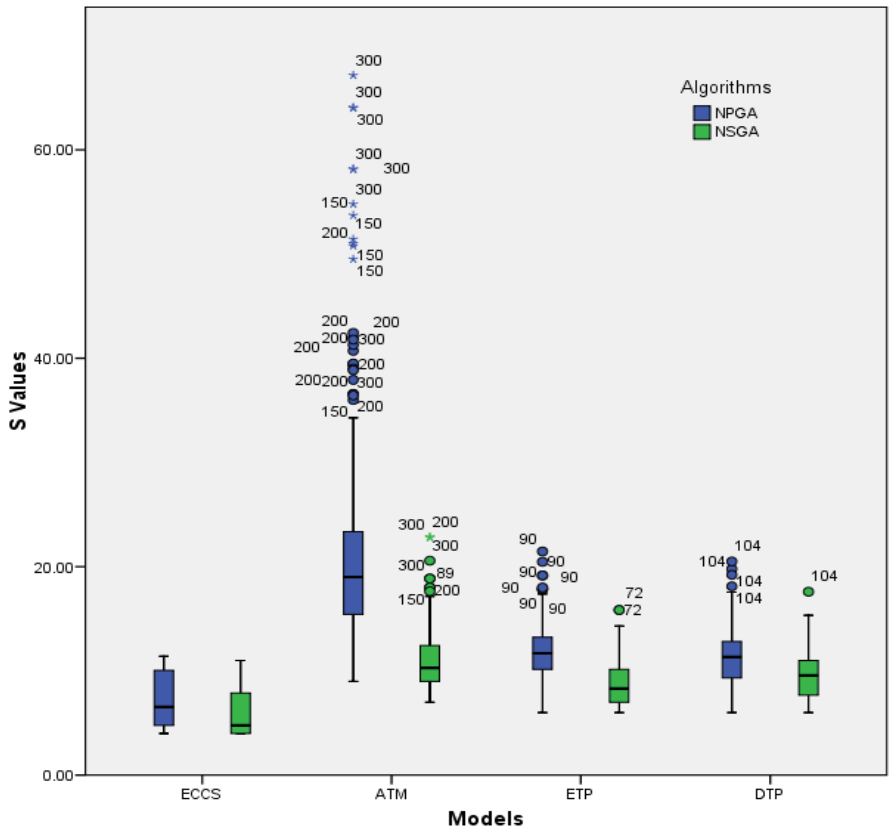


Figure 6-10: Performances of algorithms with 2-objectives

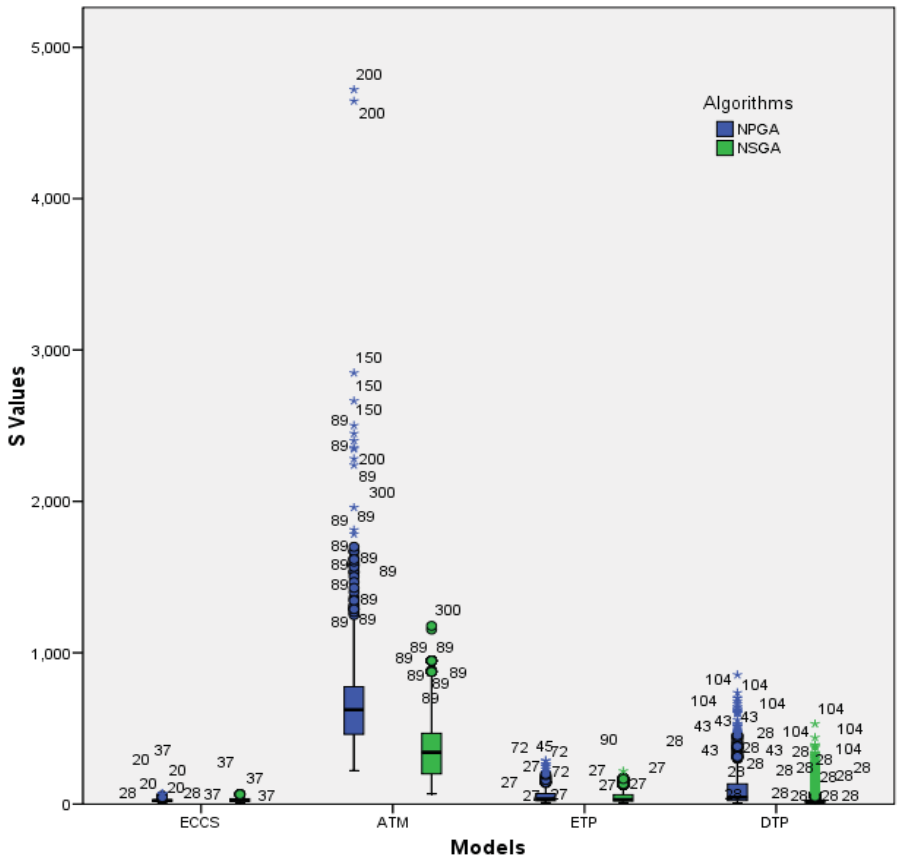


Figure 6-11: Performance of algorithms with 3-objectives

6.3 Summary

In this chapter the various aspects of multi-objective test suite minimization were investigated, evolutionary computation based optimization framework was developed and empirically validated. The multi-objective minimization of model-based test suite minimization with respect to test suite size, coverage and value problem was reformulated as Profitable Travelling Salesman Problem which is multi-objective version of Traveling Salesman Problem. The empirical result shows that Pareto-based multi-objective test suite minimization not only reduces test suite size but also provide a tester several options of a minimal test suite for execution. It is found that a combination of factors i.e. algorithm type, scale and uniformity in the measure of an objective plays important role in the quality of Pareto-based solutions. The elitism-based NSGA algorithm performed consistently better than niche-based NPGA algorithm in terms of proximity to the Pareto-optimal front, diversity and spread of the generated solution set. Although, the size of Pareto front increases with the number of objectives but scale and uniformity in the measure of an objective are critical to spread and diversity in the Pareto front.

Chapter 7 Conclusion and Future Work

In this chapter the results of the study are summed up by reviewing and assessing them. Furthermore, the limitations and restrictions of the study and conclusions are also explained. The chapter concludes with discussion about the future research possibilities.

7.1 Conclusion

In this thesis, a novel model based test suite optimization framework was presented. A number of techniques have been developed within the framework to address the following problems that UML-based models pose for the standard model based test generation and optimization.

7.1.1 The execution of models

Software models have long been used for requirement and design verification and validation purposes. They are cheaper to execute than the whole implemented system and therefore the smaller overhead makes them ideal to use in test suite generation and optimization. UML is de facto standard for software analysis and design modeling. However, models developed in UML are not outright executable as it lacks precise semantic. The thesis presented a CPN-based execution approach for AD models.

7.1.2 The ambiguous and semantically incorrect models

An automated and seamless transformation methodology from AD to CPN models is implemented and evaluated with case study models. UML has evolved into a huge language over several revisions of the standard specification and contains several compromises. Studies have shown that models developed in UML are often incorrect with respect to the standard specification [Henderson-Sellers 2005]. The thesis identified a number of sources that contributed to the ambiguity and inconsistency in an AD model and proposed an endogenous transformation and a set of control flow refinement patterns to address this issue by automatic refinement of the model. The thesis reviewed the AD, Colored Petri Nets and the syntactic and semantic differences between them. A set of transformation rules was defined and a XSLT-based transformation mechanism was used to demonstrate the automatic transformation.

7.1.3 The need for test sequences

Test generation is one of the most expensive and critical activity in the testing process. In model based testing, test sequences which represent the logical paths in the software are generated from the software model. This thesis presented a novel semantic-aware stochastic test generation algorithm for AD models. The proposed guided random walk algorithm was first of its kind to automatically derive valid test sequences from the AD models while simulating the intuitive human behavior.

7.1.4 The adequacy of a test suite

A test suite was usually generated for the coverage of certain artifacts, which could be used to calculate its quality or adequacy level. The thesis presented two types of coverage criteria for evaluating the quality or adequacy of test suites generated from AD model. Sequential coverage criteria were defined for AD-based test suite to ensure that it was adequate for sequential execution. Similarly, the concurrency coverage criteria were defined to ensure that the test suite generated from AD was adequate for concurrency testing. The thesis also proposed mutation analysis based technique to evaluate the adequacy of AD based test suite. A set of mutation operators was defined to verify the design correctness of an AD-based model according to the concurrency and control-flow logic. The proposed RW-based test generation algorithm was empirically analyzed by generating test suites for four case study models and evaluated with both coverage and mutation based techniques. Empirical results showed that the generated test suites achieved complete coverage according to a sequential criterion (branch coverage criterion); however same test suites could achieve only 50% coverage at maximum according to a concurrency coverage criterion. Furthermore, the mutation analysis of same test suites revealed that they were able to detect at least 88% of all mutants generated.

7.1.5 The optimization of a test suite

Another important contribution of this thesis is the evolutionary computation based optimization framework for model based test suite. The purpose of test suite optimization is to minimize the cost of a test suite without compromising its fault detection capability. It is achieved by either removing those test cases that are redundant or ineffective with respect to the test objectives for which they are generated or by prioritizing them to increase the rate of test objective fulfillment. In this thesis, only the former case also known as test suite minimization was used to demonstrate the proposed model based test suite optimization framework. It is shown that the redundancy that can exist in a test suite can be classified into three types. Removing the type-1 redundancy in a model based test suite is a simple task;

however removing the type-2 and the type-3 redundancy is a combinatorial optimization problem. The test suite minimization problem was reformulated as equality knapsack problem which is a well known combinatorial optimization problem. The performance of the proposed evolutionary computation based test suite minimization approach was empirically evaluated and compared with that of a Greedy and two Hill Climbing algorithms. The results showed that the evolutionary computation performed better than the other three algorithms and the difference in their performance was statistically significant. It was also found that the evolutionary computation based solution was scalable and the quality of solution was not affected much by the size of search space (test suite size). However, an important revelation of the study was that the arrangement of test cases in a test suite can affect the complexity of search space and hence the performance of algorithms. A further study to characterize the fitness landscape of test suite minimization problem revealed that it is highly rugged in general and finding the optimal solution for such problems is difficult due to high epistatic coupling between genes (test cases). Another interesting finding was that the increase in the size of a test suite may enhance the search space of that instance of the problem but not necessarily the ruggedness of its fitness landscape. On the other hand, changing the arrangement of test cases in a test suite can change the fitness landscape of that instance of the problem and might be its difficulty level as well.

As a natural extension to the evolutionary computation based optimization of model based test suite, the model based test suite optimization was extended to include multiple objectives. The multi-objective formulation of test suite optimization problem will allow incorporating multiple objectives in the minimization process. For demonstration, two instances (i.e. bi-objectives and tri-objectives) of multi-objective minimization problem of case study test suites were defined. Generally, multi-objective problems can be optimized with single objective algorithms but in these cases the problem is resolved with each of the objectives separately and finally a global trade-off solution is manually identified from a set of constrained solutions [Deb 2001]. On the other hand, multi-objective algorithms can optimize all objectives simultaneously and a trade-off solution or a set of solutions is devised automatically. For this study, two Pareto-based multiobjective evolutionary algorithms were implemented and empirically evaluated with bi- and tri-objectives minimization of case study test suites. Statistical tests were performed in order to determine whether results were significant or not. The results showed that Pareto-based multiobjective test suite minimization not only reduced test suite size but also provide a tester with several options of a minimal test suite for evaluation. It was found that a combination of factors i.e. algorithm type, scale and uniformity in the measure of an objective played important role in the quality of Pareto-based solutions. The elitism-based NSGA

algorithm performed consistently better than niche-based NPGA algorithm in terms of proximity to the Pareto-optimal front, diversity and spread of the generated solution set. Although, the size of Pareto front increased with the number of objectives but scale and uniformity in the measure of an objective were critical to the spread and diversity in the Pareto front.

7.2 Limitations and Future Work

The thesis leaves many questions unanswered and motivates several new directions of research. This section outlines some of the limitations closely associated with challenges related to the AD to CPN model transformation, AD-based test generation and test suite optimization. Furthermore, there are few issues and potential extension that may be addressed in future work.

7.2.1 Transformation of object-flow

Activity Diagram provides elements to depict both control-flow and object-flow in a system. The AD to CPN transformation presented in Chapter 3 was defined for control-flow elements only. As there is no transformation rules defined for object nodes, the XSLT-based translation can automatically skip the object node transformation. However, as such transformation can leave dangling nodes and edges in the target CPN model; it is recommended to remove object nodes from the AD model before transformation.

7.2.2 Limited transformation

The modeling concepts or elements of AD are defined in multiple packages according to their modeling capabilities. The transformation presented in Chapter 3 handled the modeling elements of AD up to *Intermediate* package as it natively supports the modeling similar to Petri Nets. Other high-level modeling elements such as Loop node, exception handling even though are related with control-flow modeling but their transformation is not supported currently in this project.

7.2.3 Mutation analysis

The set of mutation operators in Chapter 4 was defined for syntactic errors according to the control-flow and concurrency features of Activities. In order to conduct mutation analysis, a mutation tool that generates mutants for AD-model was implemented. In order to limit the cost of mutation analysis, the coupling effect was assumed to be true in this study but this hypothesis needs validation study in the context of AD. Detecting equivalent mutants is an undecidable problem and manual identification of equivalent mutants is often required. Mutation analysis for

semantic errors, data-flow and high-level constructs of AD can be very useful and need further investigation.

7.2.4 Test generation from AD model

The AD based test generation technique presented in Chapter 4 is incorporated with CPN semantic in order to produce valid execution paths for both sequential and concurrent testing. Although, it is a stochastic exploratory technique which allows it mimicking the intuitive human behavior, it has two limitations. First, it is prone to produce redundant test sequences and second, it cannot guarantee complete coverage according to a given criterion. An exhaustive technique may provide complete coverage of a trivial model but for large and complex models, it needs a guided exploration based technique to cover the unexplored parts of the model.

7.2.5 No-Free-Lunch algorithm

In Chapter 5, an EC based solution was proposed for test suite minimization problem and an empirical study showed that it performed better than the three other heuristic algorithms. According to the No-Free-Lunch theorem presented by Wolpert and Macready [1997], all heuristic algorithms perform equally on average over all possible optimization problems. The theorem basically has following implications: (a) there is no universally best general-purpose algorithm; (b) one heuristic algorithm can suite a problem better than another algorithm and outperform it on that problem; (c) a heuristic algorithm designed or adapted to a particular problem by incorporating problem specific knowledge into the algorithm can perform better than the general purpose algorithm [Knowles 2002]. So in light of these implications of No-Free-Lunch algorithm, it is important to state that all four algorithms presented in Chapter 5 were general purpose and the performance differences between these algorithms indicate that EC suits test suite minimization problem more than other three algorithms.

Bibliography

2010. Petri Nets Tool Database Department of Informatics, University of Hamburg, Petri Nets Tool Database.

AGRAWAL, A., KARSAI, G., NEEMA, S., SHI, F. and VIZHANYO, A. 2004. The design of a language for model transformations *International Journal on Software and Systems Modeling 5*, 261-288.

AGUILAR-RUIZ, J.S., RAMOS, I., RIQUELME, J.C. and TORO, M. 2001. An evolutionary approach to estimating software development projects. *Information and Software Technology 43*, 875-882.

AGUILAR, J. 2001. A General Ant Colony Model to solve Combinatorial Optimization Problems. *Revista Colombiana de Computacion 2*, 7-18.

ALANDER, J.T. 1991. On finding the optimal genetic algorithms for robot control problems. In *Proceedings of the International Workshop on Intelligent Robot and Systems*, Osaka, 3-5 November 1991 IEEE, 1313-1318.

ALANDER, J.T. 1992. On optimal population size of genetic algorithms. In *Proceedings of the 6th Annual European Computer Conference on Computer Systems and SoftwareEngineering*, Hague, 4-8 May 1992 IEEE Computer Society, 65-70.

ANDREWS, A.A., FRANCE, R.B., GHOSH, S. and CRAIG, G. 2003. Test adequacy criteria for UML design models. *Journal of Software Testing, Verification and Reliability 13*, 95-127.

ANDREWS, G.R. 2000. *Foundations of Multithreaded, Parallel and Distributed Programming*. Addison-Wesley.

ANDREWS, J.H., BRIAND, L.C. and LABICHE, Y. 2005. Is mutation an appropriate tool for testing experiments? [software testing]. In *Proceedings of the International Conference on Software Engineering*, Sept 2005, 402-411.

ANDREWS, J.H., BRIAND, L.C., LABICHE, Y. and NAMIN, A.S. 2006. Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria. *IEEE Transactions on Software Engineering 32*, 608-624.

ANG JUAY, C., HO WEE, K. and ANDREW, L. 1999. A New GA Approach for the Vehicle Routing Problem. In *Proceedings of the 11th IEEE International Conference on Tools with Artificial Intelligence*, Nov 1999 IEEE Computer Society.

APFELBAUM, L. and DOYLE, J. 1997. Model Based Testing. In *Proceedings of the Software Quality Week Conference*, May 1997.

BACK, T. 1993. Optimal mutation rates in genetic search. In *Proceedings of the 5th International Conference on Genetic Algorithms*, Urbana-Champaign, June 1993 Morgan Kaufmann, 2-8.

BACK, T., FOGEL, D.B. and MICHALEWICZ, Z. 1997. Handbook of Evolutionary Computation IOP Publishing and Oxford Univeristy Press.

BACK, T. and SCHWEFEL, H.-P. 1993. An overview of evolutionary algorithms for parameter optimization. *Evol. Comput. 1*, 1-23.

BAGNALL, A.J., RAYWARD-SMITH, V.J. and WHITTLEY, I.M. 2001. The next release problem. *Information and Software Technology 43*, 883-890.

BAKER, B.M. and AYECHEW, M.A. 2003. A genetic algorithm for the vehicle routing problem. *Computers & Operations Research 30*, 787-800.

BALA, J.W., DEJONG, K. and PACHOWICZ, P. 1991. Using Genetic Algorithms to Improve the Performance of Classification Rules Produced by Symolic Inductive Methods. In *Proceedings of the 6th International Symposium on Methodologies for Intelligent Systems, ISMIS '91*, Charlotte, N.C., USA, Oct. 16-19 1991 Springer, 121-138.

BARESI, L. and PEZZÈ, M. 2001. On Formalizing UML with High-Level Petri Nets. *Lecture Notes in Computer Science 2001*, 276-304.

BASANIERI , F. and BERTOLINO, A. 2000. A Practical Approach to UML-based Derivation of Integration Tests. In *Proceedings of the Proceedings of 4th International Quality Week Europe*, Brussels, Belgium, 20-24 November 2000.
BEIZER, B. 1990. *Software Testing Techniques*. Van Nostrand Reinhold, New York.
BEIZER, B. 1995. *Black-box Testing: Techniques for Functional Testing of Software and Systems*. Wiley, New York.
BERTOLINO, A. 2003. Software Testing Research and Practice. In *10th International Workshop on Abstract State Machines (ASM'2003)* LNCS, Taormina, Italy, 1-21.
BINDER, R. 1999. *Testing Object-Oriented Systems: Models, Patterns and Tools*. Addison-Wesley Longman, Reading, Massachusetts.
BINKLEY, D. 1995. Reducing the cost of regression testing by semantics guided test case selection. In *Proceedings of the International Conference on Software Maintenance*, Oct 1995, 251-260.
BOCK, C. 2003. Activity and Action Models Part 2: Actions. *Journal of Object Technology 2*, 41-56.
BOCK, C. 2003. UML 2 Activity and Action Models. *Journal of Object Technology 2*, 43-53.
BOCK, C. 2003. UML 2 Activity and Action Models Part 3: Control Nodes. *Journal of Object Technology 2*, 7-23.
BOCK, C. and GRUNINGER, M. 2005. PSL: A semantic domain for flow models. *Software and Systems Modeling 4*, 209-231.
BOGDAN, K. and ALI, M.A.-Y. 1998. Automated regression test generation. In *Proceedings of the International symposium on Software testing and analysis*, Clearwater Beach, Florida, United States, March 1998 ACM.
BOGDAN, K., GEORGE, K. and LUAY, H.T. 2007. Model-based test prioritization heuristic methods and their evaluation. In *Proceedings of the 3rd international workshop on Advances in model-based testing*, London, United Kingdom, July 2007 ACM.
BOGDAN, K. and KOUTSOGIANNAKIS, G. 2009. Experimental Comparison of Code-Based and Model-Based Test Prioritization. In *Proceedings of the Software Testing, Verification and Validation Workshop (ICSTW'09)*, April 2009, 77-84.
BOGDAN, K., LUAY, H.T. and MARK, H. 2005. Test Prioritization Using System Models. In *21st IEEE International Conference on Software Maintenance* IEEE Computer Society.
BOGDAN, K., LUAY, H.T. and MARK, H. 2005. Test Prioritization Using System Models. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, Nov 2005 IEEE Computer Society.
BOOCH, G. 2010. Handbook of Software Architecture. In *Preface* online.
BORGER, E., CAVARRA, A. and RICCOBENE, E. 2000. An ASM Semantics for UML Activity Diagrams. In *Proceedings of the 8th International Conference on Algebraic Methodology and Software Technology*, May 2000 Springer-Verlag.
BORGER, E., CAVARRA, A. and RICCOBENE, E. 2000. Modeling the Dynamics of UML State Machines. In *Proceedings of the Proceedings of the International Workshop on Abstract State Machines, Theory and Applications*2000 Springer-Verlag.
BOUQUET, F., JAFFUEL, E., LEGEARD, B., PEUREUX, F. and UTTING, M. 2005. Requirements traceability in automated test generation: application to smart card software validation. *SIGSOFT Softw. Eng. Notes 30*, 1-7.
BOUQUET, F. and LEGEARD, B. 2003. Reification of Executable Test Scripts in Formal Specification-Based Test Generation: The Java Card Transaction Mechanism Case Study. In *FME 2003: Formal Methods* Springer Berlin / Heidelberg, 778-795.
BRIAND, L. and LABICHE, Y. 2002. A UML-based Approach to System Testing. *Software and Systems Testing 1*, 10-42.
BRIAND, L., LABICHE, Y. and WANG, Y. 2004. Using Simulation to Empirically Investigate state Coverage Criteria based on Statecharts. In *Proceedings of the International Conference on*

Software Engineering Edinburgh, Scotland, May 2004, http://www.sce.carleton.ca/Squall/pubs/tech_report/TR_SCE-02-09.pdf, 62.
BROY, M., CRANE, M.L., DINGEL, J., HARTMAN, A., RUMPE, B. and SELIC, B. 2007. UML 2 Sematics Symposium: Formal Semantics for UML. *Lecture Notes in Computer Science 4364*, 318-323.
BROY, M., JONSSON, B., KATOEN, J.-P., LEUCKER, M. and PRETSCHNER, A. 2005. *Model-Based Testing of Reactive Systems*. Springer-Verlag, New York, Inc.
BUDD, T.A. 1981. Mutation Analysis: Ideas, Exampies, Problems and Prospects. In *Summer School on Computer Program Testing*, B. CHANDRASEKARAN and S. RADICCHI Eds. Elsevier Science Inc., Sogesta, Urbino, Italy, 129-148.
BUDD, T.A. and GOPAL, A.S. 1981. Program testing by specification mutation. In *Proceedings of the Computer Program Testing : proceedings of the Summer School on Computer Program Testing*, Urbino, Italy, June 29-July 3 1981, 129-148.
BUTLER, B.R.R. 2004. The Challenges of Complex IT Projects The Royal Academy of Engineering and The British Computer Society, Westminster, London,, 45.
CENELEC 2001. Railway Applications: Software for railway, control, and protections systems. In *EN Std 50128 - 2001 BS EN* CENELEC, 0_1-110.
CHANDLER, R., LAM, C.P. and LI, H. 2006. UML Models with Activity Diagrams: for Case Studies SCIS, Edith Cowan University, Perth, 30.
CHEN, A. and BHASKAR, K. 2008. The power of choice in random walks: An empirical study. *The International Journal of Computer and Telecommunications Networking 52*, 44-60.
CHEN, T.Y. and LAU, M.F. 1998. A new heuristic for test suite reduction. *Information and Software Technology 40*, 347-354.
CHEN, T.Y. and LAU, M.F. 2001. Test Suite Reduction and Fault Detecting Effectiveness: An Empirical Evaluation. In *Reliable Software Technologies: Ada Europe 2001* A. STROHMEIER Ed. Springer-Verlag, Leuven, Belgium, 253-265.
CHEN, T.Y. and LAU, M.F. 2003. On the divide-and-conquer approach towards test suite reduction. *Information sciences 152*, 89-119.
CHEVALLEY, P. and THEVENOD-FOSSE, P. 2001. Automated generation of statistical test cases from UML state diagrams. In *Computer Software and Applications Conference, 2001. COMPSAC 2001. 25th Annual International*, 205-214.
CHRISTOS, G., MILENA, M. and AMIN, S. 2006. Random walks in peer-to-peer networks: algorithms and evaluation. *Performance Evaluation 63*, 241-263.
CHU, P.C. and BEASLEY, J.E. 1998. A Genetic Algorithm for the Multidimensional Knapsack Problem. *Journal of Heuristics 4*, 63-86.
CLAES, W., PER, R., MARTIN, H., MAGNUS, C.O., BJÖORN, R. and ANDERS, W. 2000. *Experimentation in software engineering: an introduction*. Kluwer Academic Publishers.
COELLO, C.A.C. 1998. Using the Min-Max Method to Solve Multiobjective Optimization Problems with Genetic Algorithms. In *Proceedings of the 6th Ibero-American Conference on AI (IBERAMIA 98)*, Lisbon, Portugal, October 5-9 1998 Springer, 303-314.
COELLO, C.A.C. 2000. An updated survey of GA-based multiobjective optimization techniques. *ACM Comput. Surv. 32*, 109-143.
CORNE, D., GLOVER, F. and DORIGO, M. 1999. *New ideas in optimization*. McGraw-Hill, London.
CPN-GROUP 2010. CPN Tools CPN Group, University of Aarhus, Denmark.
CPN-GROUP 2010. File formats for CPN Tools CPN Group, University of Aarhus, Denmark.
CPN-GROUP 2010. Introduction to Hierarchical Nets CPN Group, University of Aarhus, Denmark.
CPN-SC 2010. Syntax checking. In *CPN Tools*.
CSERTÁN, G., HUSZERL, G., MAJZIK, I., PAP, Z., PATARICZA, A. and VARRÓ, D. 2002. VIATRA - Visual Automated Transformations for Formal Verification and Validation of UML Models. In *Proceedings of the Automated Software Engineering*, Edinburgh, 23-27 September 2002 IEEE Computer Society, 267-270.

DALAL, S., JAIN, A., KARUNANITHI, N., LEATON, J., LOTT, C., PATTON, G. and HOROWITZ, B. 1999. Model-Based Testing in Practice. In *Proceedings of the 21st International Conference on Software Engineering*, California, USA, May 16-22 1999 IEEE / ACM, http://www.argreenhouse.com/papers/lott/1999-icse.pdf, 285-294.

DALAL, S.R., JAIN, A., KARUNANITHI, N., LEATON, J.M., LOTT, C.M., PATTON, G.C. and HOROWITZ, B.M. 1999. Model-Based Testing in Practice. In *Proceedings of the International Conference on Software Engineering*, May 1999 ACM Press.

DAN, H., LU, Z., HAO, Z., HONG, M. and JIASU, S. 2005. Eliminating harmful redundancy for testing-based fault localization using test suite reduction: an experimental study. In *Proceedings of the 21st IEEE International Conference on Software Maintenance* Sept 2005, 683-686.

DAVID, H.A. 1987. *A connectionist machine for genetic hillclimbing*. Kluwer Academic Publishers.

DE JONG, K.A. 1975. An analysis of the behavior of a class of genetic adaptive systems University of Michigan, 266.

DE JONG, K.A. 1988. Learning with Genetic Algorithms: An Overview. *Machine Learning 3*, 121-138.

DE JONG, K.A. 1991. An analysis of the interacting roles of population size and crossover in genetic algorithms. *Lecture Notes in Computer Science 496*, 38-47.

DE JONG, K.A. 1993. Genetic Algorithms are not function optimizers. In *Proceedings of the Foundations of Genetic Algorithms*, Whitley L, California, Feb 1993, 5-17.

DEB, K. 1997. Introduction to representations. In *Handbook of Evolutionary Computation*, T. BACK, D.B. FOGEL and Z. MICHALEWICZ Eds. IOP Publishing and Oxford Univeristy Press, 127-131.

DEB, K. 2001. *Multi-Objective Optimization Using Evolutionary Algorithms*. John Wiley Sons, Inc.

DEB, K. 2007. Evolutionary Practical Optimization. In *Proceedings of the GECCO*, July 2007 ACM.

DEB, K., SAMIR, A., AMRIT, P. and MEYARIVAN, T. 2000. A Fast Elitist Non-dominated Sorting Genetic Algorithm for Multi-objective Optimisation: NSGA-II. In *Proceedings of the 6th International Conference on Parallel Problem Solving from Nature*, Sept 2000 Springer-Verlag.

DELAMARO, M.E., MALDONADO, J.C. and MATHUR, A.P. 2001. Interface Mutation: An Approach for Integration Testing. *IEEE Trans. Softw. Eng. 27*, 228-247.

DEMILLO, R.A., LIPTON, R.J. and SAYWARD, F.G. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer 11*, 34-41.

DIAS NETO, A.C. and TRAVASSOS, G.H. 2009. Porantim: An approach to support the combination and selection of Model-based Testing techniques. In *Automation of Software Test, 2009. AST '09. ICSE Workshop on*, 1-9.

DICKINSON, W., LEON, D. and PODGURSKI, A. 2001. Finding Failures by Cluster Analysis of Execution Profiles. In *Proceedings of the International Conference on Software Engineering*, Toronto, May 2001 IEEE Computer Society, 339-348.

DINH-TRONG, T.T., GHOSH, S. and FRANCE, R.B. 2006. A Systematic Approach to Generate Inputs to Test UML Design Models. In *Proceedings of the 17th International Symposium on Software Reliability Engineering*, Raleigh, North Carolina, USA, 7-10 November 2006, 95-104.

DINH-TRONG, T.T., KAWANE, N., GHOSH, S., FRANCE, R.B. and ANDREWS, A.A. 2005. A Tool-Supported Approach to Testing UML Design Models. In *Proceedings of the International Conference on Engineering of Complex Computer Systems (ICECCS 2005)*, Shanghai, China, Jun 2005 IEEE Computer Society, 519-528.

DO, H. and ROTHERMEL, G. 2005. A controlled experiment assessing test case prioritization techniques via mutation faults. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, May 2005, 411-420.

DOERNER, K. and GUTJAHR, W.J. 2003. Extracting Test Sequences from a Markov Software Usage Model by ACO. In *Proceedings of the Genetic and Evolutionary Computation Conference*, Chicago, IL, USA, July 12-16 2003 Springer, 2465-2476.

DOMINIQUE, F., PIERRE, D. and MICHEL, G. 2005. Traveling Salesman Problems with Profits. *Transportation Science 39*, 188-205.
DORIGO, M. and GAMBARDELLA, L.M. 1996. Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem Universite Libre de Bruxelles, IRIDIA, Brussels, 24.
DYE, D. 2002. Abstraction. In *Encyclopedia of Software Engineering*, J.J. MARCINIAK Ed. John Wiley & Sons Inc.
EA 2008. Enterprise Architect Sparx Systems, Creswick, Australia, CASE Tool.
EDVARDSSON, J. 1999. A Survey on Automatic Test Data Generation. In *Proceedings of the 2nd Conference on Computer Science and Engineering*, Linkoping, Sweden, October 1999, http://www.ida.liu.se/~joned/papers/class_atdg.pdf, 21-28.
EL-FAR, I.K. 2001. Enjoying the Perks of Model-Based Testing. In *Software Testing, Analysis, and Review Conference (STARWEST 2001)*, California, USA.
EL-FAR, I.K. and WHITTAKER, J.A. 2002. Model-based Software Testing. In *Encyclopedia on Software Engineering*, J.J. MARCINIAK Ed. John Wiley & Sons, Inc., New York, 22.
ELBAUM, S., MALISHEVSKY, A. and ROTHERMEL, G. 2001. Incorporating varying test costs and fault severities into test case prioritization. In *Proceedings of the International Conference on Software Engineering*, May 2001, 329-338.
ELLIMS, M., BRIDGES, J. and INCE, D.C. 2006. The Economics of Unit Testing. *Empirical Software Engineering. 11*, 5-31.
ERICKSON, M., MAYER, A. and HORN, J. 2001. The Niched Pareto Genetic Algorithm 2 Applied to the Design of Groundwater Remediation Systems. In *Evolutionary Multi-Criterion Optimization* Springer Berlin / Heidelberg, 681-695.
ERMEL, C., RUDOLF, M. and TAENTZER, G. 1997. The AGG approach: Language and tool environment. In *Handbook of graph grammars and computing by graph transformation: Applications, Languages and Tools*, H. EHRIG, H.-J.K. G. ENGELS and G. ROZENBERG Eds. World Scientific Publishing, 551-601.
ESHUIS, R. and WIERINGA, R. 2001. An Execution Algorithm for UML Activity Graphs. In *The Unified Modeling Language, Modeling Languages, Concepts, and Tools* Springer, Toronto, Canada, 47-61.
ESHUIS, R. and WIERINGA, R. 2001. A Formal Semantics for UML Activity Diagrams - Formalising Workflow Models University of Twente, Department of Computer Science, Enschede, Netherlands, 44.
EUROCAE 1992. Software Considerations in Airborne Systems and Equipment Certification. In *DO-178B / ED-12B* EUROCAE.
EVANS, A. and KENT, S. 1999. Core Meta-Modelling Semantics of UML: The pUML Approach. In *Proceedings of the 2nd International Conference on the Unified Modeling Language: Beyond the Standard*, Colorado, USA, October 28-30 1999, R.B. FRANCE and B. RUMPE Eds. LNCS, 140-155.
FABBRI, S.C.P.F., MALDONADO, J.C., MASIERO, P.C. and DELAMARO, M.E. 1999. Proteum/FSM: A Tool to Support Finite State Machine Validation Based on Mutation Testing. In *Proceedings of the International Conference of the Chilean Computer Science Society*, Nov 1999 IEEE Computer Society.
FABBRI, S.C.P.F., MALDONADO, J.C., MASIERO, P.C., DELAMARO, M.E. and WONG, E. 1996. Mutation Testing Applied to Validate Specifications Based on Petri Nets. In *Proceedings of the Proceedings of the IFIP TC6 Eighth International Conference on Formal Description Techniques VIII*1996 Chapman; Hall, Ltd.
FABBRI, S.C.P.F., MALDONADO, J.C., TATIANA, S. and MASIERO, P.C. 1999. Mutation Testing Applied to Validate Specifications Based on Statecharts. In *Proceedings of the International Symposium on Software Reliability Engineering*, Nov 1999 IEEE Computer Society, 210-219.
FACTOR, M., FARCHI, E., LICHTENSTEIN, Y. and MALKA, Y. 1996. Testing concurrent programs: a formal evaluation of coverage criteria. In *Israeli Conference on Computer-Based Systems and Software Engineering (ICCSSE '96)* IEEE Computer Society, Washington, DC, USA, 119.

FACTOR, M., FARCHI, E., LICHTENSTEIN, Y. and MALKA, Y. 1996. Testing concurrent programs: a formal evaluation of coverage criteria. In *Proceedings of the Israeli Conference on Computer-Based Systems and Software Engineering (ICCSSE '96)* Washington, DC, USA, June 1996 IEEE Computer Society, 119-126.
FARCHI, E., HARTMAN, A. and PINTER, S.S. 2002. Using a Model-Based Test Generator to Test for Standard Conformance. *IBM Systems Journal 41*, 89-110.
FARCHI, E., NIR, Y. and UR, S. 2003. Concurrent Bug Patterns and How to Test Them. In *Proceedings of the Parallel and Distributed Systems: Testing and Debugging (PADTAD)*, Nice, France, April 22-26 2003, 7.
FAROOQ, U. and LAM, C.P. 2008. Mutation Analysis for the Evaluation of AD Models. In *Proceedings of the International Conference on Computational Intelligence for Modelling Control & Automation*, Dec 2008, 296-301.
FAROOQ, U. and LAM, C.P. 2009. Evolving the Quality of a Model Based Test Suite. In *International Conference on Software Testing, Verification and Validation Workshops*, 141-149.
FAROOQ, U., LAM, C.P. and LI, H. 2006. Transformation Methodology for UML 2.0 Activity Diagram into Colored Petri Nets. In *4th IASTED International Conference on Advances in Computer Science and Technology* ACTA Press, Phuket, Thailand, 6.
FAROOQ, U., LAM, C.P. and LI, H. 2008 Towards Automated Test Sequence Generation. In *Proceedings of the 19th Australian Software Engineering Conference (ASWEC'2008)*, Perth, Australia, March 25-28 2008
FENTON, N.E. and PFLEEGER, S.L. 1997. *Software Metrics: A Rigorous & Practical Approach*. ITP.
FLOCKHART, I.W. and RADCLIFFE, N.J. 1996. A Genetic Algorithm-Based Approach to Data Mining. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, Portland, Oregon, USA, August 1996, 299-302.
FLORIN, J. 2008. An Evaluation on Model Based Testing at Ericsson. In *School of Electrical Engineering* Royal Institute of Technology, Stockholm, Sweden, 80.
FOGEL, L.J., OWENS, A. and WALSH, M. 1966. *Artificial Intelligence Through Simulated Evolution*. Hohn Wiley, New York.
FONSECA, C.M. and FLEMING, P.J. 1993. Genetic Algorithms for Multiobjective Optimization: Formulation, Discussion and Generalization. In *Proceedings of the Fifth International Conference in Genetic Algorithm*, San Mateo, CA, June 1993, 416-423.
FORREST, S. and MITCHELL, M. 1993. What Makes a Problem Hard for a Genetic Algorithm? Some Anomalous Results and Their Explanation. *Machine Learning 13*, 285-319.
FORTUNATO, S. and FLAMMINI, A. 2007. RANDOM WALKS ON DIRECTED NETWORKS: THE CASE OF PAGERANK. *International Journal of Bifurcation and Chaos (IJBC) in Applied Sciences and Engineering 17*, 2343-2353.
FRANCE, R., EVANS, A., LANO, K. and RUMPE, B. 1998. The UML as a formal modeling notation. *Comput. Stand. Interfaces 19*, 325-334.
FRANKL, P.G. and IAKOUNENKO, O. 1998. Further Empirical Studies of Test Effectiveness. In *Proceedings of the International Symposium on Foundations of Software Engineering*, Lake Buena Vista, FL, USA, Nov 1998 ACM SIGSOFT, 153-162.
FRANKL, P.G. and WEISS, S.N. 1993. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Transactions on Software Engineering 19*, 774-787.
FRIEDMAN, G., HARTMAN, A., NAGIN, K. and SHIRAN, T. 2002. Projected State Machine Coverage for Software Testing. In *Proceedings of the ISSTA*, Rome, Italy, July 22-24 2002, 10.
FRISCH, A.M., HNICH, B., MIGUEL, I., SMITH, B.M. and WALSH, T. 2002. Towards CSP Model Reformulation at Multiple level of Abstraction. In *Proceedings of the International Workshop on Reformulating Constraint Satisfaction Problems*, Ithaca, NY, USA, Sep 2002, 42-56.
FUJIKI, C. and DICKINSON, J. 1987. Using the Genetic Algorithm to generate lisp source code to solve the prisoner's dilemma. In *Proceedings of the Second International Conference on Genetic Algorithms and their Applications*, July 1987, 236-240.

GARFINKEL, S. 2005. History's Worst Software Bugs Wired Magazine.
GEHRKE, T., GOLTZ, U. and WEHRHEIM, H. 1998. The Dynamic Models of UML: Towards a Semantics and its Application in the Development Process Universitt Hildesheim.
GERARDO, C., MASSIMILIANO DI, P., RAFFAELE, E. and MARIA LUISA, V. 2005. An approach for QoS-aware service composition based on genetic algorithms. In *Proceedings of the Genetic and evolutionary computation conference (GECCO)*, Washington DC, USA, June 2005 ACM.
GIULIANO, A., MASSIMILIANO DI, P. and MARK, H. 2004. A Robust Search-Based Approach to Project Management in the Presence of Abandonment, Rework, Error and Uncertainty. In *Proceedings of the International Symposium on Software Metrics*, Sep 2004 IEEE Computer Society.
GOEL, A.L. 1985. Software Reliability Models: Assumptions, Limitations, and Applicability. *IEEE Transaction on Software Engineering 11*, 1411-1423.
GOLDBERG, D.E. 1989. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley.
GOLDBERG, D.E. and SAMTANI, M.P. 1986. Enginnering optimization via Genetic Algorithm. In *Proceedings of the 9th Conference on Electronic Computation*, Feb 1986, 471-482.
GÓRRIZ, J.M. and PUNTONET, C.G. 2005. GA-ICA algorithms applied to image processing In *Soft Computing as Transdisciplinary Science and Technology* Springer, 1269-1277.
GRADY, L. 2006. Random Walks for Image Segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence 28*, 1768–1783.
GREFENSTETTE, J. 1986. Optimization of control parameters for genetic algorithms. *IEEE Trans. Syst. Man Cybern. 16*, 122-128.
GREFENSTETTE, J.J. and FITZPATRICK, J.M. 1985. Genetic Search with approximate function evaluation. In *Proceedings of the first international Conference on Genetic Algorithm and their Applications*, July 1985, 112-120.
GREFENSTETTE, J.J., GOPAL, R., ROSMAITA, B. and VAN GUCHT, V. 1985. Genetic Algorithms for the Traveling Salesman Problem. In *Proceedings of the first international Conference on Genetic Algorithm and their Applications*, July 1985, 160-168.
HAHNERT, W.H. and RALSTON, P.A.S. 1995. Analysis of population size in the accuracy and performance of genetic training for rule-based control systems. *Computers & Operations Research 22*, 55-72.
HAMLET, D. 2006. When only random testing will do. In *Proceedings of the 1st international workshop on Random testing*, Portland, Maine, July 2006 ACM.
HAMLET, R.G. 1977. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering 3*, 279-290.
HANSEN, M.P. and JASZKIEWICZ, A. 1998. Evaluating the quality of approximations to the non-dominated set Institute of Mathematical Modelling, 31.
HAREL, D. and GERY, E. 1997. Executable object modeling with statecharts *IEEE Computer 30*, 31-42.
HARMAN, M. and JONES, B.F. 2001. Search based software engineering. *Information and Software Technology 43*, 833-839.
HARMAN, M., MANSOURI, S.A. and ZHANG, Y. 2009. Search Based Software Engineering.: A Comprehensive Analysis and Review of Trends Techniques and Applications King's College London, London, 78.
HARROLD, M.J., GUPTA, R. and SOFFA, M.L. 1993. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering Methodology 2*, 270--285.
HECKEL, R. 2003. Foundations of Visual Modelling Techniques. In *Advance Course on Petri Nets* Universitat Paderborn, 6.
HEIMDAHL, M.P.E. and GEORGE, D. 2004. Test-suite reduction for model-based tests: Effects on test quality and implications for testing. In *Proceedings of the Automated Software Engineering* Linz, Austria, September 20-24 2004, 176-185.

HEIMDAHL, M.P.E., RAYADURGAM, S., VISSER, W., DEVARAJ, G. and GAO, J. 2003. Auto-generating Test Sequences Using Model Checkers: A Case Study. In *Formal Approaches to Software Testing* Springer, Montreal, Quebec, Canada, 42-59.
HENDERSON-SELLERS, B. 2005. UML– the Good, the Bad or the Ugly? Perspectives from a panel of experts. *Software System Model 4*, 4-13.
HEYLIGHEN, F. 2010. Web Dictionary of Cybernetics and Systems, Online Dictionary.
HILLAH, L., KORDON, F., PETRUCCI, L. and TRÈVES, N. 2006. PN Standardisation: A Survey. In *Formal Techniques for Networked and Distributed Systems - FORTE 2006*, 307-322.
HOLLAND, J.H. 1973. Genetic Algorithms and the Optimal Allocation of Trials. *SIAM J. Comput. 2*, 88-105.
HOLLAND, J.H. 1992. *Adaptation in natural and artificial systems*. MIT Press.
HORN, J. and GOLDBERG, D.E. 1995. Genetic Algorithm Difficulty and the Modality of Fitness Landscapes. In *Foundations of Genetic Algorithms* Morgan Kaufmann, 243--269.
HORN, J., NAFPLIOTIS, N. and GOLDBERG, D.E. 1994. A niched Pareto genetic algorithm for multiobjective optimization. In *Proceedings of the IEEE World Congress on Computational Intelligence Evolutionary Computation*, Jun 1994, 82-87 vol.81.
HUCKLE, T. 2005. Collection of Software Bugs, München.
HUGHES, C. and HUGHES, T. 2003. *Parallel and Distributed Programming Using C++*. Addison-Wesley, Bostan, MA.
HUTCHINS, M., FOSTER, H., GORADIA, T. and OSTRAND, T. 1994. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the International Conference on Software Engineering*, Sorrento, Italy, May 1994, 191-200.
IBM 2010. Rational XDE IBM, Rational eXtended Development Environment.
IBM 2010. SPSS SPSS Inc, Chicago.
INRIA 2010. Guimoo - User's Manual INRIA, www.lifl.fr/OPAC/guimoo.
ISO/IEC 2005. Information technology -- XML Metadata Interchange (XMI) ISO/IEC.
ISO/IEC 2005. Unified Modeling Language Specification International Organization for Standardization, 454.
ISO/IEC 2009. Software and system engineering -- High-level Petri nets -- Part 2: Transfer Format ISO/IEC, 102.
JEFFREY, D. and GUPTA, N. 2005. Test Suite Reduction with Selective Redundancy. In *Proceedings of the International Conference on Software Maintenance (ICSM 2005)*, Budapest, Hungary, September 25-30 2005, 549-558.
JEFFREY, D. and GUPTA, N. 2007. Improving Fault Detection Capability by Selectively Retaining Test Cases during Test Suite Reduction. *Software Engineering, IEEE Transactions on 33*, 108-123.
JENNIFER, B., EMANUEL, M. and DAVID, K. 2004. Bi-Criteria Models for All-Uses Test Suite Reduction. In *Proceedings of the International Conference on Software Engineering*, May 2004 IEEE Computer Society.
JENSEN, K. 1992. Coloured Petri Nets: Basic Concept, Analysis Methods and Practical Use. *EATCS Monographs on Theoretical Computer Science 1*.
JENSEN, K. 1993. An introduction to the theoretical aspects of coloured petri nets. *Lecture Notes in Computer Science 803*, 230-272.
JENSEN, K. 1996. An introduction to the practical use of coloured petri nets. *Lecture Notes in Computer Science 1492*, 237-292.
JENSEN, K. 1997. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*. Springer-Verlag.
JIA, Y. and HARMAN, M. 2010. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering*.
JONAS, B. 2008. Early fault detection with model-based testing. In *Proceedings of the 7th ACM SIGPLAN workshop on ERLANG*, Victoria, BC, Canada, Sept 2008 ACM.

JONES, B., SHAMER, H. and EYRES, D. 1995. Automatic structural testing using genetic algorithms. *Software Eng Journal 12*, 57-74.
JONES, B.F., EYRES, D.E. and STHAMER, H.H. 1996. Automatic structural testing using genetic algorithms. *The Software Engineering Journal 11*, 299-306.
JONES, B.F., EYRES, D.E. and STHAMER, H.H. 1998. A strategy for using genetic algorithms to automate branch and fault-based testing. *The Computer Journal 41*, 98–107.
JONES, J. and HARROLD, M. 2003. Test-Suite Reduction and Prioritization for Modified Condition/Decision Coverage. *IEEE Transactions on Software Engineering 29*, 195-209.
JONES, T. 1995. Evolutionary Algorithms, Fitness landscapes and Search The University of New Mexico, 249.
JONES, T. and FORREST, S. 1995. Fitness Distance Correlation as a Measure of Problem Difficulty for Genetic Algorithms. In *Proceedings of the 6th International Conference on Genetic Algorithms*, July 1995 Morgan Kaufmann Publishers Inc., 184-192.
JORGENSEN, P.C. 2002. *Software Testing: A Craftman's Approach*. CRC Press, Boca Raton, Florida.
KANER, C. 1997. The Impossibility of Complete Testing. In *Software QA*, 28.
KANER, C., FALK, J. and NGUYEN, H.Q. 1993. *Testing Computer Software* International Thomson Computer Press.
KANSOMKEAT, S., OFFUTT, J., ABDURAZIK, A. and BALDINI, A. 2008. A Comparative Evaluation of Tests Generated from Different UML Diagrams. In *Proceedings of the 9th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing*, Aug 2008, 867-872.
KAUFFMAN, S.A. 1993. *The Origins of Order : Self-Organization and Selection in Evolution* Oxford University Press.
KAY, M.H. 2010. SAXON: The XSLT and XQuery Processor Sourceforge.net.
KEITH, D.C., PHILIP, J.S. and DEVIKA, S. 1999. Optimizing for reduced code space using genetic algorithms. In *Proceedings of the ACM SIGPLAN 1999 workshop on Languages, compilers, and tools for embedded systems*, Atlanta, Georgia, United States, May 1999 ACM.
KELLERER, H., PFERSCHY, U. and PISINGER, D. 2004. *Knapsack Problems*. Springer.
KERSTEN, M. and NEBE, W. 2004. On detecting deadlocks in large UML models. In *Proceedings of the IFIP Working Conference on Distributed and Parallel Embedded Systems*, Toulouse, France, Aug 2004, 11-20.
KHADKA, B. 2007. Transformation of Live Sequence Charts to Colored Petri Nets (LSCTOCPN). In *Computer and Information Science Department* UNIVERSITY OF MASSACHUSETTS, DARTMOUTH, 51.
KHURSHID, S., PASAREANU, C.S. and VISSER, W. 2003. Generalized Symbolic Execution for Model Checking and Testing. In *Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems*, Warsaw, Poland, April 7-11 2003 Springer, 553-568.
KNOWLES, D.J. and CORNE, W.D. 2000. Approximating the Nondominated Front Using the Pareto Archived Evolution Strategy. *Evolutionary Computation 8*, 149-172.
KNOWLES, D.J., THIELE, L. and ZITZLER, E. 2006. A Tutorial on the Performance Assessment of Stochastic Multiobjective Optimizers Computer Engineering and Networks Laboratory, ETH Zurich, Switzerland, Zurich.
KNOWLES, J.D. 2002. Local-Search and Hybrid Evolutionary Algorithms for Pareto Optimization. In *Department of Computer Science* University of Reading, Reading.
KOEHLER, J., HAUSER, R., SENDALL, S. and WAHLER, M. 2005. Declarative techniques for model-driven business process integration. *IBM Syst. J. 44*, 47-65.
KOEHLER, J., HAUSER, R., SENDALL, S. and WAHLER, M. 2005. Declarative techniques for model-driven business process integration. *IBM Systems Journal 44*, 47-65.

KOEHLER, J., KUSTER, J.M., NOVATNACK, J. and RYNDINA, K. 2006. A Classfification of UML2 Activity Diagrams IBM Research GmbH, Zurich Research Laboratory, Ruschlikon, Switzerland, 1-287.
KOREL, B. and KOUTSOGIANNAKIS, G. 2009. Experimental Comparison of Code-Based and Model-Based Test Prioritization. In *Software Testing, Verification and Validation Workshops, 2009. ICSTW '09. International Conference on*, 77-84.
KUHN, D.R. 1999. Fault classes and error detection capability of specification-based testing. *ACM Transactions on Software Engineering and Methodology 8*, 411-424.
KÜSTER, J. 2006. Definition and validation of model transformations. *Software and Systems Modeling 5*, 233-259.
LAKHOTIA, K., HARMAN, M. and MCMINN, P. 2007. A multi-objective approach to search-based test data generation. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, London, England, July 2007 ACM.
LANGE, C.F.J. and CHAUDRON, M.R.V. 2005. Managing Model Quality in UML-Based Software Development. In *Proceedings of the 13th International Workshop on Software Technology and Engineering Practice (STEP 2005)*, Budapest, Hungary, 24-25 September 2005 IEEE Computer Society, 7-16.
LEE, D., SABNANI, K.K., KRISTOL, D.M. and PAUL, S. 1996. Conformance Testing of Protocols Specified as Communicating Finite State Machines - a Guided Random Walk Based Approach. *IEEE Trans. on Communications 44*, 631-640.
LEVESON, N.G. 1995. *SAFEWARE: SYSTEM SAFETY AND COMPUTERS*. Addison-Wesley.
LI, H. and LAM, C.P. 2005. An Ant Colony Optimization Approach to Test Sequence Generation for Statebased Software Testing. In *Proceedings of the 5th International Conference on Quality Software*, Sept 2005 IEEE Computer Society.
LI, Z., HARMAN, M. and HEIRONS, R. 2007. Search Algorithms for Regression Test Case Prioritisation *IEEE Transactions on Software Engineering. 33*, 225-237.
LI, Z., HARMAN, M. and HIERONS, R.M. 2007. Search Algorithms for Regression Test Case Prioritization. *IEEE Transactions on Software Engineering 33*, 225-237.
LI, Z., TAN, L., WANG, X., LU, S., ZHOU, Y. and ZHAI, C. 2006. Have things changed now?: an empirical study of bug characteristics in modern open source software. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, San Jose, California, Oct 2006 ACM.
LINZHANG, W., JIESONG, Y., XIAOFENG, Y., JUN, H., XUANDONG, L. and GUOLIANG, Z. 2004. Generating test cases from UML activity diagrams based on gray-box method. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference*, Busan, Korea, 30 Nov - 3 Dec 2004, 284-291.
LOPEZ-GRAO, J.P., MERSEGUER, J. and CAMPOS, J. 2004. From UML activity diagrams to Stochastic Petri nets: application to software performance engineering. In *Proceedings of the 4th international workshop on Software and performance*, Redwood Shores, California, Jan 2004 ACM.
MADRAS, N. and SLADE, G. 1996. *The Self-Avoiding Walk*. Springer Verlag.
MASRI, W., PODGURSKI, A. and LEON, D. 2007. An Empirical Study of Test Case Filtering Techniques Based on Exercising Information Flows. *IEEE Transactions on Software Engineering 33*, 454-477.
MCGRAW-HILL 2003. Dictionary of Scientific & Technical Terms. In *Dictionary of Scientific & Technical Terms* The McGraw-Hill Companies, Inc.
MCMINN, P. 2004. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability 14*, 105-156.
MCMINN, P., HARMAN, M., BINKLEY, D. and TONELLA, P. 2006. The species per path approach to search-based test data generation. In *Proceedings of the International symposium on Software testing and analysis*, Portland, Maine, USA, July 2006 ACM.

MCQUILLAN, J.A. and POWER, J.F. 2005. A Survey of UML-Based Coverage Criteria for Software Testing National University of Ireland, Maynooth, 18.
MENS, T. and GROP, P.V. 2006. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science 152*, 125-142.
MENS, T., VAN GORP, P., VARRO, D. and KARSAI, G. 2006. Applying a Model Transformation Taxonomy to Graph Transformation Technology. In *Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005)*, Sept 2006, 143--159.
MERSEGUER, J. 2003. Software Performance Engineering based on UML and Petri Nets University of Zaragoza, Zaragoza.
MERZ, S. 2000. Model Checking: a tutorial overview. In *Modeling and verification of parallel processes* Springer, 3-38.
MICHAEL, C., MCGRAW, G., SCHATZ, M. and WALTON, C. 1997. Genetic algorithms for dynamic test data generation. In *Proceedings of the IEEE International Conference on Automated Software Engineering*, Nov 1997, 307-308.
MICHAEL, C.C., MCGRAW, G. and SCHATZ, M.A. 2001. Generating software test data by evolution. In *IEEE Transactions on Software Engineering*, 1085–1110.
MILICEV, D. 2002. Automatic Model Transformations Using Extended UML Object Diagrams in Modeling Environments. *IEEE Transactions on Software Engineering 28*, 413-431.
MINGSONG, C., XIAOKANG, H. and XUANDONG, L. 2006. Automatic Test Case Generation for UML Activity Diagrams. In *Proceedings of the Automated Software Testing*, Shanghai, China2006 ACM.
MITCHELL, M. 1998. *An Introduction to Genetic Algorithms*. The MIT Press, Massachusettss.
MITCHELL, M., FORREST, S. and HOLLAND, J.H. 1992. The Royal Road for Genetic Algorithms: Fitness Landscapes and GA Performance. In *First European Conference on Artifical Life* MIT Press.
MÜHLENBEIN, H. 1991. Darwin's continent cycle theory and its simulation by the Prisoner's Dilemma. *Complex Systems 5*, 459-478.
MYERS, G.J. 1979. *The Art of Software Testing*. John Wiley & Sons, New York.
NAMIN, A.S. and ANDREWS, J.H. 2009. The influence of size and coverage on test suite effectiveness. In *Proceedings of the Proceedings of the eighteenth international symposium on Software testing and analysis*, Chicago, IL, USA2009 ACM.
NETO, A.C.D., SUBRAMANYAN, R., VIEIRA, M. and TRAVASSOS, G.H. 2007. A survey on model-based testing approaches: a systematic review. In *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*, Atlanta, Georgia, Nov 2007 ACM.
NIST 2011. Dataplot Satistical Engineering Division, NIST.
NIST/SEMATECH 2010. e-Handbook of Statistical Methods NIST/SEMATECH.
NTAFOS, S.C. 1988. A comparison of some structural testign strategies. *IEEE Transactions on Software Engineering 14*, 868-874.
O'DARE, M.J. and ARSLAN, T. 1994. Generating test patterns for VLSI circuits using a genetic algorithm. *Electronics Letters 30*, 778-779.
OFFUTT, A.J. 1992. Investigations of the Software Testing Coupling Effect. *ACM Transactions on Software Engineering Methodology 1*, 3-18.
OFFUTT, A.J. and ABDURAZIK, A. 1999. Generating Tests from UML Specifications. In *2nd International Conference on the Unified Modeling Language (UML'99)*, Colorado, USA, 14.
OFFUTT, A.J., PAN, J. and VOAS, J. 1995. Procedures for Reducing the Size of Coverage-based Test Sets. In *International Conference on Testing Computer Software*, Washington D.C., USA, 111-123.
OFFUTT, A.J., PAN, J. and VOAS, J.M. 1995. Procedures for Reducing the Size of Coverage-based Test Sets. In *Proceedings of the International Conference on Testing Computer Software*1995, 111-123.

OFFUTT, J. 1995. A Practical System for Mutation Testing: Help for the Common Programmer. In *12th International Conference on Testing Computer Software*, Washington D.C., USA, 99-109.
OLIVER, I.M., SMITH, D.J. and HOLLANDT, J.R.C. 1987. A study of permutation crossover operatord on the Traveling Salesman Problem. In *Proceedings of the Second International Conference on Genetic Algorithms and their Applications*, July 1987, 224-230.
OMG 2005. UML 2.0 Superstructure Specification, 709.
OMG 2005. UML 2.0 Superstructure Specification Object Management Group, 709.
OMG 2007. MOF 2.0 / XMI Mapping 2.1.1 Object Management Group, 120.
OMG 2007. Unified Modeling Language: Superstructure Object Management Group, 732.
OMG 2008. UML Profile for Modeling and Analysis of Real-time and Embedded Systems Object Management Group, 1-664.
OSTER, N. and SAGLIETTI, F. 2006. Automatic Test Data Generation by Multi-Objective Optimisation. In *25th International Conference on Computer Safety, Reliability and Security (SAFECOMP 2006)* Springer-Verlag, 426-438.
OULD, M. and PRAXIS, B. 1991. Testing-a challenge to method and tool developers. *Software Engineering Journal 6*, 59-64.
PARADKAR, A. 2005. Case studies on fault detection effectiveness of model based test generation techniques. In *Proceedings of the 1st international workshop on Advances in model-based testing*, St. Louis, Missouri, July 2005 ACM.
PARGAS, R.P., HARROLD, M.J. and PECK, R.R. 1999. Test-data generation using genetic algorithms. *The Journal of Software Testing, Verification and Reliability 9*, 263–282.
PARMEE, I.C. and DENHAM, M.J. 1994. The integration of adaptive search techniques with current engineering design practice. In *Proceedings of the ACEDC'94*, Plymouth, UK, Aug 1994, 1-13.
PATTON, R.M., WU, A.S. and WALTON, G.H. 2003. A genetic algorithm approach to focused software usage testing. In *Software Engineering with Computational Intelligence*, T.M. KHOSHGOFTAAR Ed. Kluwer Academic Publishers, 259-286.
PEDRO, S.-N., RODOLFO, F.R., CLARINDO, P. and DUA 2008. An evaluation of a model-based testing method for information systems. In *Proceedings of the ACM symposium on Applied computing*, Fortaleza, Ceara, Brazil, Mar 2008 ACM.
PELÁNEK, R., HANŽL, T., ČERNÁ, I. and BRIM, L. 2005. Enhancing random walk state space exploration. In *Proceedings of the 10th international workshop on Formal methods for industrial critical systems*, Lisbon, Portugal, Sept 2005 ACM.
PELTIER, M., ZISERMAN, F. and BÉZIVIN, J. 2000. On Levels of Model Transformation. In *Proceedings of the XML Europe 2000*, Paris, France, 12-16 June 2000.
PERRY, W. 2006. *Effective Methods for Software Testing*. Wiley Publishing, Inc., Indianapolis, Indiana.
PHYLLIS, G.F. and OLEG, I. 1998. Further empirical studies of test effectiveness. *SIGSOFT Softw. Eng. Notes 23*, 153-162.
PRENNINGER, W. and PRETCHNER, A. 2005. Abstractions for Model-Based Testing. *Electronic Notes in Theoratical Computer Science 116*, 59-71.
PRESSMAN, R. 2001. *Software Engineering: A Practitioners Approach*. McGraw - Hill, New York.
PRETSCHNER, A. and PHILIPPS, J. 2005. Methodological Issues in Model-Based Testing. In *Model-Based Testing of Reactive Systems*, 281-291.
PRETSCHNER, A., PRENNINGER, W., WAGNER, S., KUHNEL, C., BAUMGARTNER, M., SOSTAWA, B., ZOLCH, R. and STAUNER, T. 2005. One evaluation of model-based testing and its automation. In *Proceedings of the International conference on Software engineering*, St. Louis, MO, USA, May 2005 ACM.
PRINS, C. 2004. A simple and effective evolutionary algorithm for the vehicle routing problem. *Computers & Operations Research 31*, 1985-2002.

RAMASWAMY, S. and NEELAKANTAN, R. 2002. Software Design and Testing Using Petri Nets: A Case Study Using a Distributed Simulation Software System. In *Proceedings of the Performance Metrics for Intelligent Systems*, Gaithersburg, MD, Gaithersburg, MD 2002.
RECHENBERG, I. 1973. Evolutionsstrategie, optimierung technischer systeme nach prinzipien der biologischen evolution. *Stuttgart: Frammann-Holzboog Verlag.*
RENLUND, H. 2005. Reinforced Random Walk Department of Mathematics, Uppsala University, 53.
RESEARCH, M. 2010. Model-based Testing with SpecExplorer Microsoft Research.
REYNOLDS, R., ZANNONI, E. and POSNER, R. 1994. Learning to understand software using cultural algorithms. In *3rd Annual Conf on Evolutionary Programming*, San Diego, CA, 150-157.
RICE, R.W. 2010. The Economics of Testing Rice Consulting, 5.
ROBINSON, H. 1999. Graph Theory Techniques in Model-Based Testing. In *International Conference on Testing Computer Software*, 12.
ROBINSON, H. 2000. Intelligent Test Automation. *Software Testing & Quality Engineering*, 24-32.
RODRIGUES, R.W.S. 2000. Formalising UML Activity Diagrams using Finite State Processes. In *Proceedings of the International Workshop on Dynamic Behavior in UML Models: Semantic Questions*, Oct 2000, A.K. G. REGGIO, B. RUMPE, B. SELIC AND R. WIERINGA Ed. Inst. f. Informatik, Ludwig-Maximilians-Universität, München, 92-98.
ROGER, F. and BOGDAN, K. 1996. The chaining approach for software test data generation. *ACM Transactions on Software Engineering and Methodology 5*, 63-86.
ROPER, M. 1996. CAST with GAs (Genetic Algorithms) - Automatic Test Data Generation via. Evolutionary Computation. In *IEE Colloquium on Computer Aided Software Testing Tools*.
ROSARIA, S. and ROBINSON, H. 2000. Applying Models in your Testing Process. *Information and Software Technology 42*, 16.
ROTHERMEL, G. and ELBAUM, S. 2003. Putting your best tests forward. *Software, IEEE 20*, 74-77.
ROTHERMEL, G. and HARROLD, M.J. 1993. A safe, efficient algorithm for regression test selection. In *Proceedings of the Conference on Software Maintenance*, Sept 1993, 358-367.
ROTHERMEL, G., HARROLD, M.J., OSTRIN, J. and HONG, C. 1998. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *Proceedings of the Conference on Software Maintenance*, Nov 1998, 34-43.
ROTHERMEL, G., UNTCH, R.H., CHENGYUN, C. and HARROLD, M.J. 1999. Test case prioritization: an empirical study. In *Proceedings of the IEEE International Conference on Software Maintenance*, 30 Aug - 3 Sept 1999, 179-188.
ROTHERMEL, G., UNTCH, R.H., CHENGYUN, C. and HARROLD, M.J. 2001. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering 27*, 929-948.
ROTHLAUF, F. 2006. *Representations for Genetic and Evolutionary Algorithms*. Springer.
ROUNTEV, A., KAGAN, S. and SAWIN, J. 2005. Coverage criteria for testing of object interactions in sequence diagrams. In *Proceedings of the Fundamental Approaches to Software Engineering*, Apr 2005 Springer, 282-297.
SAMI, K. and BATAREKH, A. 1990. Heuristics for the Integer Knapsack Problem. In *Proceedings of the Xth International Computer Science Conference*, Santiago, Chile, June 1990, 161-172.
SAMI, K., THOMAS, B. and HEIKOTTER, J. 1994. The zero/one multiple knapsack problem and genetic algorithms. In *Proceedings of the ACM symposium on Applied computing*, Phoenix, Arizona, United States, July 1994 ACM.
SCHAFFER, J.D. 1985. Multiple objective optimization with vector evaluated genetic algorithms. In *Proceedings of the First International Conference on Genetic Algorithms*1985, 93-100.
SCHAFFER, J.D., CARUANA, R.A., ESHELMAN, L.J. and DAS, R. 1989. A study of control parameters affecting online performance of genetic algorithms for function optimization. In *Proceedings of the 3rd International Conference on Genetic Algorithms*, Fairfax, VA, June 1989 Morgan Kaufmann, 51-60.

SCHATTKOWSKY, T. and FORSTER, A. 2007. On the Pitfalls of UML 2 Activity Modeling. In *Proceedings of the International Workshop on Modeling in Software Engineering* May 2007 IEEE Computer Society.
SCHATTKOWSKY, T. and FORSTER, A. 2007. On the Pitfalls of UML 2 Activity Modeling. In *Proceedings of the International Workshop on Modeling in Software Engineering*, May 2007 IEEE Computer Society.
SCHÜRR, A., WINTER, A.J. and ZÜNDORF, A. 1997. The Progres Approach: Language And Environment. In *Handbook of graph grammars and computing by graph transformation: Foundations*, H. EHRIG, H.-J.K. G. ENGELS and G. ROZENBERG Eds. World Scientific Publishing, 487-550.
SELIC, B. 2008. Filling in the Whitespace: A Strategy for Research in Model-Driven Development Malina Software Corp., Carleton, Canada, 38.
SENDALL, S. and STROHMEIER, A. 2002. Using OCL and UML to Specify System Behavior. *Object Modeling with the OCL, LNCS 2263*, 250-279.
SHEN, W., COMPTON, K. and HUGGINS, J. 2001. A UML Validation Toolset Based on Abstract State Machines. In *16th IEEE International Conference on Automated Software Engineering (ASE'01)*, San Diego, USA, 315-318.
SHIBA, T., TSUCHIYA, T. and KIKUNO, T. 2004. Using Artificial Life Techniques to Generate Test Cases for Combinatorial Testing. In *28th Annual International Computer Software and Applications Conference (COMPSAC'04)* IEEE, Hong Kong, 72-77.
SHOUSHA, M., BRIAND, L. and LABICHE, Y. 2008. A UML/SPT Model Analysis Methodology for Concurrent Systems Based on Genetic Algorithms Carleton University, Ottawa, 18.
SILVER, E.A., VIDAL, R.V.V. and WERRA, D.D. 1980. A tutorial on heuristic methods. *European Journal of Operational Research 5*, 153-162.
SINHA, A., WILLIAMS, C.E. and SANTHANAM, P. 2006. A measurement framework for evaluating model-based test generation tools. *IBM Systems Journal 45*, 501-514.
SIVARAJ, H. and GOPALAKRISHNAN, G. 2003. RandomWalk Based Heuristic Algorithms for Distributed Memory Model Checking University of Utah, School of Computing, Salt Lake City.
SNYERS, D. and PETILLOT, Y. 1995. Image processing optimization by genetic algorithm with a new coding scheme. *Pattern Recogn. Lett. 16*, 843-848.
SOUZA, S.D.R.S.D., MALDONADO, J.C., FABBRI, S.C.P.F. and SOUZA, W.L.D. 1999. Mutation Testing Applied to Estelle Specifications. *Software Quality Control 8*, 285-301.
SRIVASTAVA, A. and THIAGARAJAN, J. 2002. Effectively prioritizing tests in development environment. *SIGSOFT Softw. Eng. Notes 27*, 97-106.
STACHOWIAK, H. 1973. *Allgemeine Modelltheorie*. Wien: Springer.
STANDARD, E. 2001. EN Std 50128 - 2001 BSEN. In *EN Std 50128 - 2001 BS EN*, 0_1-110.
STATSOFT 2011. StatSoft Electronic Statistics Textbook StatSoft, Inc, Tulsa, OK.
STHAMER, H. 1996. The Automatic Generation of Software Test Data Using Genetic Algorithms. University of Glamorgan, Pontyprid, Wales, Great Britain.
STOBIE, K. 2005. Model Based Testing in Practice at Microsoft. *Electronic Notes in Theoretical Computer Science 111*, 5-12.
STOCKS, P.A. 1993. Applying formal methods to software testing. In *Department of computer science* University of Queensland, Brisbane.
STÖRRLE, H. 2004. Semantics of Control-Flow in UML 2.0 Activities. In *IEEE Symposium on Visual Languages and Human-Centric Computing* IEEE Computer Society, 235-242.
STÖRRLE, H. 2004. Structured Nodes in UML 2.0 Activities. *Nordic Journal of Computing 11*, 279-302.
STÖRRLE, H. and HAUSMANN, J.H. 2005. Semantics and Verification of Data Flow in UML 2.0 Activities. *Electronic Notes Theoratical Computer Science 127*, 35-52.
STÖRRLE, H. and HAUSMANN, J.H. 2005. Towards a Formal Semantics of UML 2.0 Activities. In *Proceedings of the Software Engineering*, Essen, March 8-11 2005, 117-128.

SUGETA, T., MALDONADO, J.C. and WONG, W.E. 2004. Mutation Testing Applied to Validate SDL Specifications. In *TestCom* Springer, Oxford, UK, 193-208.
SYSTEMS, S. 2008. Enterprise Architect Sparx Systems, Creswick, Australia, CASE Tool.
T. Y. CHEN and LAU, M.F. 2003. On the divide-and-conquer approach towards test suite reduction. *Information sciences 152*, 89-119.
TAGHI, M.K., YI, L. and NAEEM, S. 2004. Module-Order Modeling using an Evolutionary Multi-Objective Optimization Approach. In *Proceedings of the 10th International Symposium on Software Metrics*, Sept 2004 IEEE Computer Society.
TAI, K. 1989. Testing of concurrent software. In *13th Annual International Computer Software and Applications Conference*, 62-64.
TALBI, E.-G. 2009. *Metaheuristics: from design to implementation*. John Wiley & Sons.
TALLAM, S. and GUPTA, N. 2005. A Concept Analysis Inspired Greedy Algorithm for Test Suite Minimization. In *Workshop on Program Analysis for Software Tools and Engineering* ACM, Lisbon, Portugal.
TAN, G. 2009. A Collection of Well-Known Software Failures Lehigh University.
TASSEY, G. 2002. The Economic Impacts of Inadequate Infrastructure for Software Testing National Institute of Standards & Technology (NIST), Gaithersburg, 309.
TATE, D.M. and SMITH, A.E. 1992. A Genetic Approach to the Quadratic Assignment Problem. *Computers & Operations Research*, 24.
TAVARES, J., PEREIRA, F.B. and COSTA, E. 2008. Multidimensional Knapsack Problem: A Fitness Landscape Analysis *IEEE Trans. Systems, Man, and Cybernatics 38*, 604-616.
TENNANT, A. and CHAMBERS, B. 1994. Adaptive optimisation techniques for the design of microwave absorbers. In *Proceedings of the ACEDC'94*, Plymouth, UK, Aug 1994, 44-49.
THOMAS, D. 2004. MDA: revenge of the modelers or UML utopia? *Software, IEEE 21*, 15-17.
TOBIAS, B. and LOTHAR, T. 1995. A Mathematical Analysis of Tournament Selection. In *Proceedings of the International Conference on Genetic Algorithms*, July 1995 Morgan Kaufmann Publishers Inc.
TRACEY, N., CLARK, J. and MANDER, K. 1998. Automated program flaw finding using simulated annealing. In *International Symposium on Software Testing and Analysis* ACM/SIGSOFT, 73-81.
TRACEY, N., CLARK, J., MANDER, K. and MCDERMID, J. 1998. An Automated Framework for Structural Test-Data Generation, 4 pages.
UTTING, M. 2005. Position paper: Model-Based Testing. In *Proceedings of the Verified Software: Theories, Tools, Experiments*, ETH Zürich, 10-15 October 2005 Springer.
UTTING, M. 2008. The Role of Model-Based Testing. *Lecture Notes in Computer Science 4171*, 510-517.
UTTING, M. and LEGEARD, B. 2006. *Practical Model-Based Testing - A Tools Approach*. Morgan Kaufmann.
UTTING, M. and LEGEARD, B. 2007. Commercial MBT Tools Waikato University, 2.
UTTING, M., PRETSCHNER, A. and LEGEARD, B. 2006. A Taxonomy of model-based testing. In *Working paper series* Department of Computer Science, 18.
VARRO, D., VARRO, G. and PATARICZA, A. 2002. Designing the automatic transformation of visual languages. *Sci. Comput. Program. 44*, 205-227.
W3C 2010. The Extensible Stylesheet Language Family (XSL) World Wide Web Consortium (W3C).
WAGNER, S. 2006. A Literature Survey of the Quality Economics of Defect-Detection Techniques. In *Proceedings of the International Symposium on Empirical Software Engineering* Rio de Janeiro, Brazil, Sept. 21-22 2006 ACM, 194-204.
WANG, L., YUAN, J., YU, X., HU, J., LI, X. and ZHENG, G. 2004. Generating Test Cases from UML Activity Diagram based on Gray-Box Method. In *Asia-Pacific Software Engineering Conference (APSEC'04)*, Busan, Korea, 284-291.

WATANABE, H. and KUDOH, T. 1995. Test Suite Generation Methods for Concurrent Systems Based on Colored Petri Nets. In *Proceedings of the Second Asia Pacific Software Engineering Conference*, Dec 1995 IEEE Computer Society.
WATKINS, A. 1995. The automatic generation of test data using genetic algorithms. In *4th Software Quality Conf*, Dundee, UK, 300-309.
WEGENER, J., BARESEL, A. and STHAMER, H. 2001. Evolutionary test environment for automatic structural testing. *Information and Software Technology - Special Issue on Software Engineering using Metaheuristic Innovative Algorithms 43*, 841-854.
WEGENER, J., EYRES, D.E., STHAMER, H. and JONES, B.F. 1997. Testing real-time systems using genetic algorithms. *Software Quality 6*, 127–135.
WEINBERGER, E.D. 1990. Correlated and Uncorrelated landscape and How to tell the difference. *Biological Cybernatics 63*, 325-336.
WEYUKER, E. 1990. The cost of data flow testing: An empirical study. *IEEE Transactions on Software Engineering 16*, 121-127.
WHITE, L.J. and COHEN, E.I. 1980. A domain strategy for computer program testing. *IEEE Transactions on Software Engineering 6*, 247-257.
WHITLEY, D. 1993. A Genetic Algorithm Tutorial Colorado State University, Colorado, 37.
WILLIAMS, C.E. 1999. Software Testing and the UML. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE'99)*, Boca Raton, November 1-4 1999, 2.
WILLIAMS, T.W., MERCER, M.R., MUCHA, J.P. and KAPUR, R. 2001. Code Coverage, What Does It Mean in Terms of Quality? In *Proceedings of the Annual Symposium on Reliability and Maintainability*, Jan 2001, 420-424.
WILSON, S.W. 1987. The Genetic Algorithm and biological developmenturce code to solve the Prisoner's Dilemma. In *Proceedings of the Second International Conference on Genetic Algorithms and their Applications*, July 1987, 247-251.
WOLPERT, D.H. and MACREADY, W.G. 1997. No free lunch theorems for optimization. *Evolutionary Computation, IEEE Transactions on 1*, 67-82.
WONG, W.E., HORGAN, J.R., LONDON, L. and MATHUR, A.P. 1995. Effect of Test Set Minimization on Fault Detection Effectiveness. In *17th International Conference on Software Engineering*, 41-50.
WONG, W.E., HORGAN, J.R., LONDON, S. and MATHUR, A.P. 1994. Effect of Test Set Size and Block Coverage on the Fault Detection Effectiveness. In *5th International Symposium on Software Reliability Engineering*, 230-238.
WONG, W.E., HORGAN, J.R., MATHUR, A.P. and PASQUINI, A. 1997. Test Set Size Minimization and Fault Detection Effectiveness: A Case Study in a Space Application In *Proceedings of the Annual International Computer Software & Applications Conference*1997, 522-528.
WONG, W.E., HORGAN, J.R., MATHUR, A.P. and PASQUINI, A. 1999. Test set size minimization and fault detection effectiveness: A case study in a space application *Journal of Systems and Software 48*, 79-89.
WOODWARD, M.R., HEDLEY, D. and HENNELL, M.A. 1980. Experience with path analysis and testing of programs. *IEEE Transactions on Software Engineering 6*, 278-286.
XIE, T., MARINOV, D. and NOTKIN, D. 2004. Rostra: A Framework for Detecting Redundant Object-Oriented Unit Tests. In *Automated Software Engineering* IEEE, Linz, Austria, 196-205.
XU, D., LI, H. and LAM, C.P. 2005. Using Adaptive Agents to Automatically Generate Test Scenarios from the UML Activity Diagrams. In *Proceedings of the 12th Asia-Pacific Software Engineering Conference*, Dec 2005 IEEE Computer Society.
YANG, D. and ZHANG, S. 2003. Using π-calculus to formalize UML activity diagram for business process modeling. In *Engineering of Computer-Based Systems, 2003. Proceedings. 10th IEEE International Conference and Workshop on the*, 47-54.

YANG, R.-D. and CHUNG, C.-G. 1990. A path analysis approach to concurrent program testing. In *Ninth Annual International Phoenix Conference on Computers and Communications* IEEE Computer Society, Scottsdale, AZ, USA, 425-432.
YANPING, C., ROBERT, L.P. and HASAN, U. 2007. Regression test suite reduction using extended dependence analysis. In *Proceedings of the Fourth international workshop on Software quality assurance: in conjunction with the 6th ESEC/FSE joint meeting*, Dubrovnik, Croatia, Sept 2007 ACM.
YE, R. and MALAIYA, Y.K. 2002. Relationship Between Test Effectiveness and Coverage. In *Proceedings of the International Symposium on Sftware Reliability Engineering*, November 2002, 159-160.
YOO, S. and HARMAN, M. 2007. Pareto efficient multi-objective test case selection. In *International Symposium on Software Testing and Analysis*, London, United Kingdom, 140-150.
YOUNG, M. and TAYLOR, R.N. 1989. Rethinking the taxonomy of fault detection techniques. In *Proceedings of the International conference on Software engineering*, Pittsburgh, Pennsylvania, United States, May 1989 ACM.
YU-SEUNG, M., JEFF, O. and YONG RAE, K. 2005. MuJava: an automated class mutation system: Research Articles. *Softw. Test. Verif. Reliab. 15*, 97-133.
YVES, V., PETER, R. and YOLANDE, B. 2008. Genetic algorithm-based optimization of service composition and deployment. In *Proceedings of the International workshop on Services integration in pervasive environments*, Sorrento, Italy, July 2008 ACM.
ZHAN, Y. and CLARK, J.A. 2005. Search-based mutation testing for *Simulink* models. In *Proceedings of the Conference on Genetic and evolutionary computation*, Washington DC, USA, June 2005 ACM.
ZHANG, Y., HARMAN, M. and MANSOURI, S.A. 2007. The multi-objective next release problem. In *Proceedings of the Annual conference on Genetic and evolutionary computation*, London, England, July 2007 ACM.
ZHU, H., HALL, P.A.V. and MAY, J.H.R. 1997. Software unit test coverage and adequacy. *ACM Computing Surveys 29*, 366-427.
ZHU, H. and HE, X. 2002. A methodology of testing high-level Petri nets. *Information and Software Technology 44*, 473-489.
ZITZLER, E. and LOTHAR, T. 1999. Multiobjective Evolutionary Algorithms: A Comparative Case Study and the Strength Pareto Approach. *IEEE Transactions on Evolutionary Computation 3*, 257-271.

Appendix A. AD to CPN Transformation

1. Implementation

In Chapter 3, the AD to CPN transformation methodology and selected technology are described. The implementation of proposed three step methodology is described as follows:

1. At first step, the typical ambiguities are identified and removed from AD model.
2. Secondly, the refined AD model is transformed into CPN model, and
3. Finally, enhancements in the CPN model are performed in order to make it executable.
4. Most of the CPN tools use XML based files to store all the information about the models.

XSLT is a standard technology for transformation between XML or other structured documents. Extensible Stylesheet Language (XSL) is an XML-based language to specify the transformation rules also known as templates. An XSLT engine takes XML based input document and XSL Stylesheet, and produces target document. On loading input document, the engine prepares source tree from it and then starts acting on the templates in the Stylesheet sequentially by navigating to the nodes in the source tree as specified in templates. One finding an exact matching, the engine then either creates one or more nodes in the result tree, or processes them in the source tree according to the instructions in each template. The output is finally derived from the result tree. A typical implementation of a XSLT based transformation consists of four parts:

1. One or more XML based input documents
2. One or more transformation templates specified in XML based Stylesheet
3. A XSLT templates execution engine.
4. One or more XML based output documents

In order to realize the proposed XSL based transformation of AD models into CPN models, they were needed in XML format. So the case study AD models were exported into XMI file. The transformation rules were implemented in XSL Stylesheet and specified in the following

section. For the execution of transformation templates, the home edition of 'Saxon' [Kay 2010], an open source XSLT execution engine was used. It provides the implementations of latest versions of XSLT, XQuery, and XPath at the basic level of conformance as defined by W3C [W3C 2010].

2. XSL templates for AD to CPN Transformation

In this section, the XSL templates are presented. The templates are based on the transformation rules defined in Chapter 3 for AD to CPN transformation. The first template is main template which calls the templates related with pre-transformation, transformation and post-transformation steps. The ActivitytoNet template performs the transformation step and calls the rest of the templates on each occurrence of the associated element in AD model (XML format).

1. AD to CPN Template

```
<xsl:template match="/" >
  <workspaceElements>
  <generator tool="CPN Tools" version="2.2.0" format="6"/>
    <cpnet>
      <!-- Pre Transformation step-->
      <xsl:call-template name="Pre-TransformationStep-Template"/>
      <!-- Transformation step-->
      <xsl:apply-templates select="xmi:XMI/uml:Model/packagedElement/packagedElement[@xmi:type='uml:Activity
      <!-- Post Transformation step-->
      <xsl:call-template name="Post-TransformationStep-Template"/>
    </cpnet>
  </workspaceElements>
</xsl:template>
```

2. Pre-Transformation Step Template

```
<xsl:template name="Pre-TransformationStep-Template">
      <xsl:call-template name="Implicit2ExplicitFork-Template"/>
      <xsl:call-template name="Implicit2ExplicitJoin-Template"/>
      <xsl:call-template name="Implicit2ExplicitDecision-Template"/>
      <xsl:call-template name="Implicit2ExplicitMerge-Template"/>
</xsl:template>
```

3. Activity to Net Template

```
<xsl:template name="Activity2Net" match="xmi:XMI/uml:Model/packagedElement/packagedElement[@xmi:type='uml:Activity']">
  <page>
      <xsl:attribute name="id">
          <xsl:value-of select="@xmi:id" />
      </xsl:attribute>
      <pageattr>
          <xsl:attribute name="name">
          <xsl:value-of select="@name" />
          </xsl:attribute>
      </pageattr>

      <xsl:apply-templates select="node[@xmi:type='uml:InitialNode']"/>
      <xsl:apply-templates select="node[@xmi:type='uml:FinalNode']"/>
      <xsl:apply-templates select="node[@xmi:type='uml:MergeNode' or @xmi:type='uml:DecisionNode']"/>
      <xsl:apply-templates select="node[@xmi:type='uml:ForkNode' or @xmi:type='uml:JoinNode']"/>
      <xsl:apply-templates select="node[@xmi:type='uml:Action']"/>
      <xsl:apply-templates select="edge[local:nodeType(@source)='place' and local:nodeType(@target)='transition']" mode="PandT"/
      <xsl:apply-templates select="edge[local:nodeType(@source)='transition' and local:nodeType(@target)='place']" mode="TandP"/
      <xsl:apply-templates select="edge[local:nodeType(@source)='transition' and local:nodeType(@target)='transition']" mode="Ta
      <xsl:apply-templates select="edge[local:nodeType(@source)='place' and local:nodeType(@target)='place']" mode="PandP"/>
  </page>
</xsl:template>
```

4. Action to Transition Template

```
<xsl:template name="Action2Transition" match="node[@xmi:type='uml:Action']">
  <trans>
      <xsl:variable name="TransitionNodes" select="." />
      <xsl:attribute name="id">
          <xsl:value-of select="@xmi:id" />
      </xsl:attribute>
      <posattr>
          <xsl:variable name="sPlacePosition" select="$DiagramElements[@subject=$TransitionNodes/@xmi:id]/@geometr
          <xsl:variable name="nodePos1" select='local:split($sPlacePosition)'  />
          <xsl:attribute name="x">
              <xsl:value-of select='$nodePos1[6]' />
          </xsl:attribute>
          <xsl:attribute name="y">
              <xsl:value-of select='$nodePos1[11]' />
          </xsl:attribute>
      </posattr>
      <text>
          <xsl:variable name="tName" select="(if (@name)  then @name else concat('F', position()) )"/>
              <xsl:value-of select="$tName" />
      </text>
  </trans>
</xsl:template>
```

5. Fork or Join to Transition Template

```
<xsl:template name="ForknJoin2Transition" match="node[@xmi:type='uml:ForkNode' or @xmi:type='uml:JoinNode']">
  <trans>
      <xsl:variable name="TransitionNodes" select="." />
      <xsl:attribute name="id">
          <xsl:value-of select="@xmi:id" />
      </xsl:attribute>
      <posattr>
          <xsl:variable name="sPlacePosition" select="$DiagramElements[@subject=$TransitionNodes/@xmi:id]/@geometr
          <xsl:variable name="nodePos1" select='local:split($sPlacePosition)'  />
          <xsl:attribute name="x">
              <xsl:value-of select='$nodePos1[6]' />
          </xsl:attribute>
          <xsl:attribute name="y">
              <xsl:value-of select='$nodePos1[11]' />
          </xsl:attribute>
      </posattr>
      <text>
          <xsl:variable name="tName" select="(if (@name)  then @name else concat('F', position()) )"/>
              <xsl:value-of select="$tName" />
      </text>
  </trans>
</xsl:template>
```

6. Merge or Decision to Place Template

```
<xsl:template name="MergenDecisionNode2Place" match="node[@xmi:type='uml:MergeNode' or @xmi:type='uml:DecisionNod
  <place>
    <xsl:variable name="PlaceNodes" select="." />
    <xsl:attribute name="id">
      <xsl:value-of select="@xmi:id" />
    </xsl:attribute>
    <posattr>
      <xsl:variable name="sPlacePosition" select="$DiagramElements[@subject=$PlaceNodes/@xmi:id]/@geometry" /
      <xsl:variable name="nodePos1" select='local:split($sPlacePosition)'  />
      <xsl:attribute name="x">
        <xsl:value-of select='$nodePos1[6]' />
      </xsl:attribute>
      <xsl:attribute name="y">
        <xsl:value-of select='$nodePos1[11]' />
      </xsl:attribute>
    </posattr>
    <text>
      <xsl:value-of select="@name" />
    </text>
    <type>
      <text>
        <xsl:value-of select="'TKN'" />
      </text>
    </type>
  </place>
</xsl:template>
```

7. Initial Node to Place Template

```
<xsl:template name="InitialNode2Place" match="node[@xmi:type='uml:InitialNode']">
  <place>
    <xsl:variable name="PlaceNodes" select="." />
    <xsl:attribute name="id">
      <xsl:value-of select="@xmi:id" />
    </xsl:attribute>
    <posattr>
      <xsl:variable name="sPlacePosition" select="$DiagramElements[@subject=$PlaceNodes/@xmi:id]/@geometr
      <xsl:variable name="nodePos1" select='local:split($sPlacePosition)'  />
      <xsl:attribute name="x">
        <xsl:value-of select='$nodePos1[6]' />
      </xsl:attribute>
      <xsl:attribute name="y">
        <xsl:value-of select='$nodePos1[11]' />
      </xsl:attribute>
    </posattr>
    <text>
      <xsl:value-of select="@name" />
    </text>
    <type>
      <text>
        <xsl:value-of select="'TKN'" />
      </text>
    </type>
    <initmark>
      <text>
        <xsl:if test="$PlaceNodes[@xmi:type='uml:InitialNode']">
        <xsl:value-of select='$initToken' />
        </xsl:if>
      </text>
    </initmark>
  </place>
</xsl:template>
```

8. Final Node to Place Template

```
<xsl:template name="FinalNode2Place" match="node[@xmi:type='uml:FinalNode']">
  <place>
      <xsl:variable name="PlaceNodes" select="." />
      <xsl:attribute name="id">
          <xsl:value-of select="@xmi:id" />
      </xsl:attribute>
      <posattr>
          <xsl:variable name="sPlacePosition" select="$DiagramElements[@subject=$PlaceNodes/@xmi:id]/@geometr
          <xsl:variable name="nodePos1" select='local:split($sPlacePosition)'  />
          <xsl:attribute name="x">
              <xsl:value-of select='$nodePos1[6]' />
          </xsl:attribute>
          <xsl:attribute name="y">
              <xsl:value-of select='$nodePos1[11]' />
          </xsl:attribute>
      </posattr>
      <text>
          <xsl:value-of select="@name" />
      </text>
      <type>
          <text>
              <xsl:value-of select="'TKN'" />
          </text>
      </type>
  </place>
</xsl:template>
```

9. Edge to Arc Template (Place to Transition)

```
<xsl:template name="Edge2Arc0" match="edge[local:nodeType(@source)='place' and local:nodeType(@target)='transition']" mode="P
  <arc>
      <xsl:attribute name="id">
        <xsl:value-of select="@xmi:id" />
      </xsl:attribute>
      <xsl:variable name="edgeSrc" select="@source" />
      <xsl:attribute name="orientation">
                  <xsl:value-of select="if (local:nodeType(@source)='place') then 'PtoT' else 'TtoP'"/>
      </xsl:attribute>
      <transend>
          <xsl:attribute name="idref">
          <xsl:value-of select="@target" />
          </xsl:attribute>
      </transend>
      <placeend>
          <xsl:attribute name="idref">
          <xsl:value-of select="@source" />
          </xsl:attribute>
      </placeend>
      <annot>
        <xsl:attribute name="id">
          <xsl:value-of select="generate-id()" />
        </xsl:attribute>
        <text>
          <xsl:value-of select="'s'" />
        </text>
      </annot>
   </arc>
</xsl:template>
```

10. Edge to Arc Template (Transition to Place)

```
<xsl:template name="Edge2Arc1" match="edge[local:nodeType(@source)='transition' and local:nodeType(@target)='place']" mode="T
  <arc>
    <xsl:attribute name="id">
      <xsl:value-of select="@xmi:id" />
    </xsl:attribute>
    <xsl:variable name="edgeSrc" select="@source" />
    <xsl:attribute name="orientation">
            <xsl:value-of select="if (local:nodeType(@source)='place') then 'PtoT' else 'TtoP'"/>
    </xsl:attribute>
    <transend>
        <xsl:attribute name="idref">
        <xsl:value-of select="@source" />
        </xsl:attribute>
    </transend>
    <placeend>
        <xsl:attribute name="idref">
        <xsl:value-of select="@target" />
        </xsl:attribute>
    </placeend>
    <annot>
      <xsl:attribute name="id">
        <xsl:value-of select="generate-id()" />
      </xsl:attribute>
      <text>
        <xsl:value-of select="'s'" />
      </text>
    </annot>
  </arc>
</xsl:template>
```

11. Edge to Arc Template (Transition to Transition)

```
<xsl:template name="Edge2Arc2" match="edge[local:nodeType(@source)='transition' and local:nodeType(@target)='transition']" mode="T
  <xsl:variable name="newNodeID" select="generate-id()"/>
  <arc>
    <xsl:attribute name="id">
        <xsl:value-of select="@xmi:id" />
    </xsl:attribute>
    <xsl:variable name="edgeSrc" select="@source" />
    <xsl:attribute name="orientation">
        <xsl:value-of select="if (local:nodeType(@source)='place') then 'PtoT' else 'TtoP'"/>
    </xsl:attribute>
    <transend>
        <xsl:attribute name="idref">
        <xsl:value-of select="@source" />
        </xsl:attribute>
    </transend>
    <placeend>
        <xsl:attribute name="idref">
        <xsl:value-of select="$newNodeID" />
        </xsl:attribute>
    </placeend>
    <annot>
        <xsl:attribute name="id">
            <xsl:value-of select="concat($newNodeID,'annot')" />
        </xsl:attribute>
        <text>
            <xsl:value-of select="'s'" />
        </text>
    </annot>
  </arc>
  <xsl:call-template name="CreateDummyPlace">
    <xsl:with-param name="newNodeID" select="$newNodeID" />
  </xsl:call-template>
  <xsl:call-template name="CreateDummyEdge">
    <xsl:with-param name="transend" select="@target" />
    <xsl:with-param name="placeend" select="$newNodeID" />
    <xsl:with-param name="orientation" select="'PtoT'" />
  </xsl:call-template>
</xsl:template>
```

12. Edge to Arc Template (Place to Place)

```
<xsl:template name="Edge2Arc3" match="edge[local:nodeType(@source)='place' and local:nodeType(@target)='place']" mode="Pa
   <xsl:variable name="newNodeID" select="generate-id()"/>
   <arc>
      <xsl:attribute name="id">
         <xsl:value-of select="@xmi:id" />
      </xsl:attribute>
      <xsl:variable name="edgeSrc" select="@source" />
      <xsl:attribute name="orientation">
         <xsl:value-of select="if (local:nodeType(@source)='place') then 'PtoT' else 'TtoP'"/>
      </xsl:attribute>
      <transend>
         <xsl:attribute name="idref">
            <xsl:value-of select="$newNodeID" />
         </xsl:attribute>
      </transend>
      <placeend>
         <xsl:attribute name="idref">
            <xsl:value-of select="@source" />
         </xsl:attribute>
      </placeend>
      <annot>
         <xsl:attribute name="id">
            <xsl:value-of select="concat($newNodeID,'annot')" />
         </xsl:attribute>
         <text>
            <xsl:value-of select="'s'" />
         </text>
      </annot>
   </arc>
   <xsl:call-template name="CreateDummyTransition">
      <xsl:with-param name="newNodeID" select="$newNodeID" />
   </xsl:call-template>
   <xsl:call-template name="CreateDummyEdge">
      <xsl:with-param name="transend" select="$newNodeID" />
      <xsl:with-param name="placeend" select="@target" />
      <xsl:with-param name="orientation" select="'TtoP'" />
   </xsl:call-template>
</xsl:template>
```

13. Post-Transformation Step Template

```
<xsl:template name="Post-TransformationStep-Template">
      <xsl:call-template name="globbox-Template"/>
      <xsl:call-template name="instances-Template"/>
      <xsl:call-template name="options-Template"/>
      <xsl:call-template name="binders-Template"/>
      <xsl:call-template name="monitorblock-Template"/>
      <xsl:call-template name="IndexNode-Template"/>
</xsl:template>
```

3. CPN Model of Case Studies

Following are the CPN models (without marking) produced from the case study AD models using model transformation methodology mentioned in Chapter 3.

a. Enterprise Customer Commerce System (ECCS)

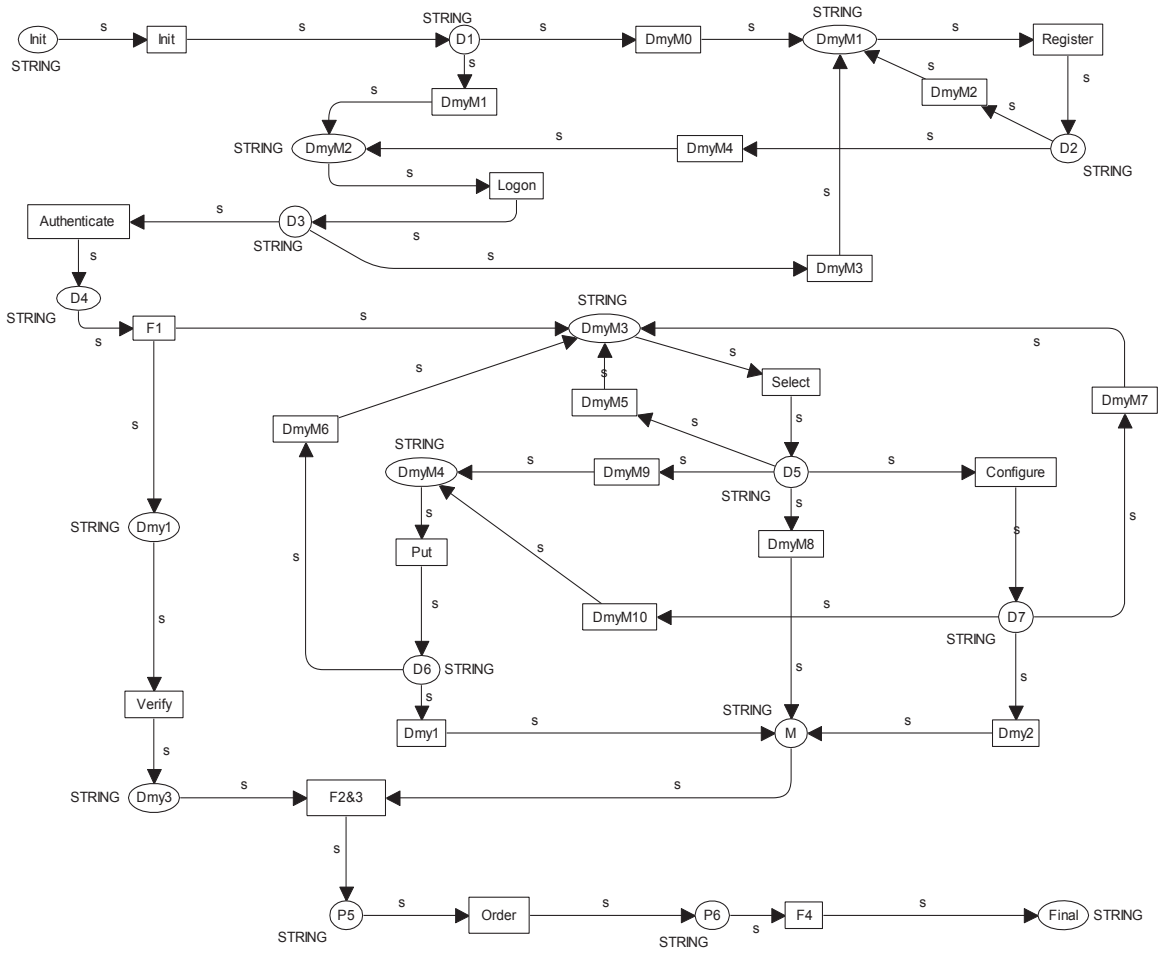


Figure A-1: ECCS CPN model

b. Automated Teller Machine (ATM)

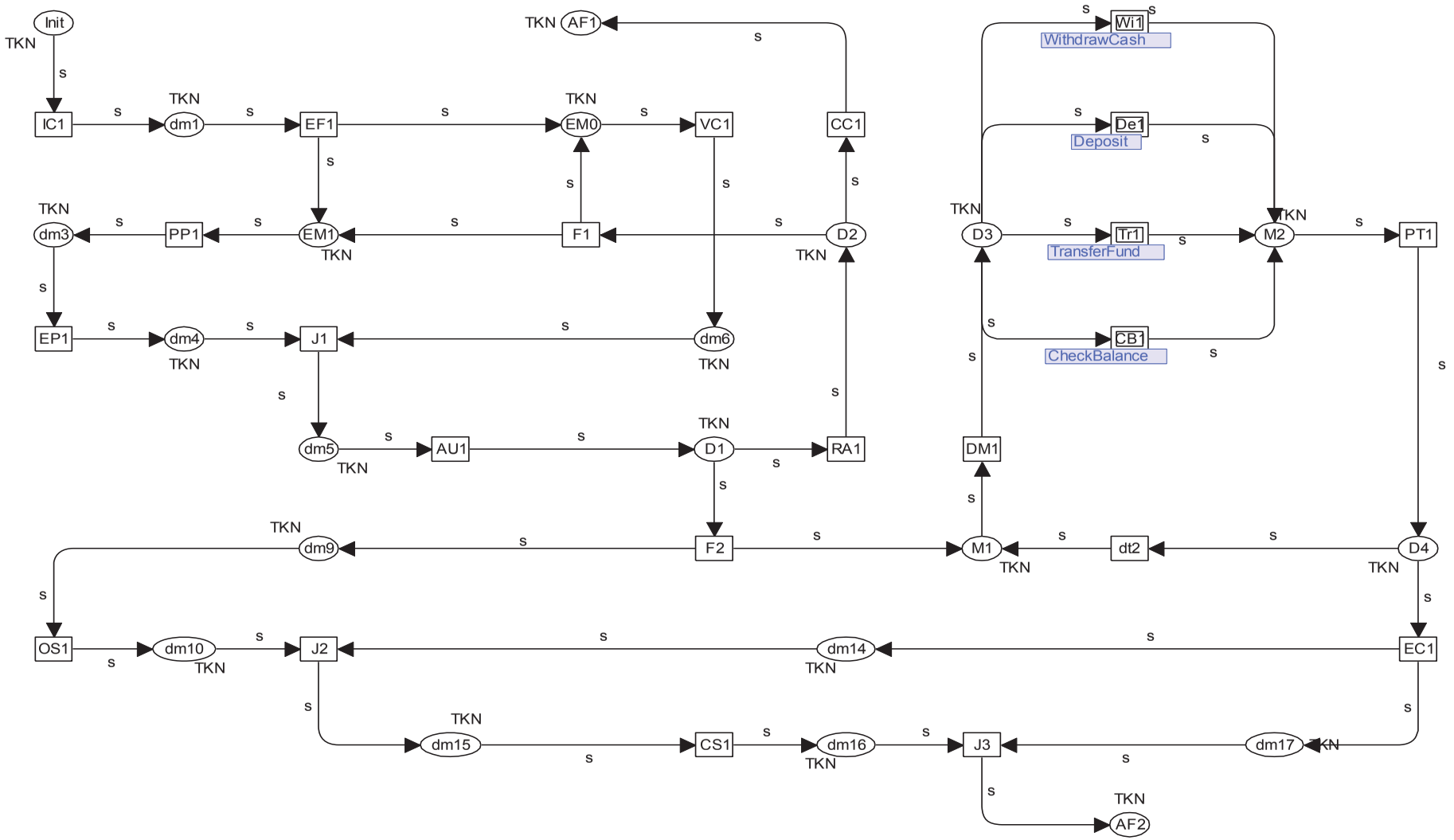


Figure A-2: ATM CPN model

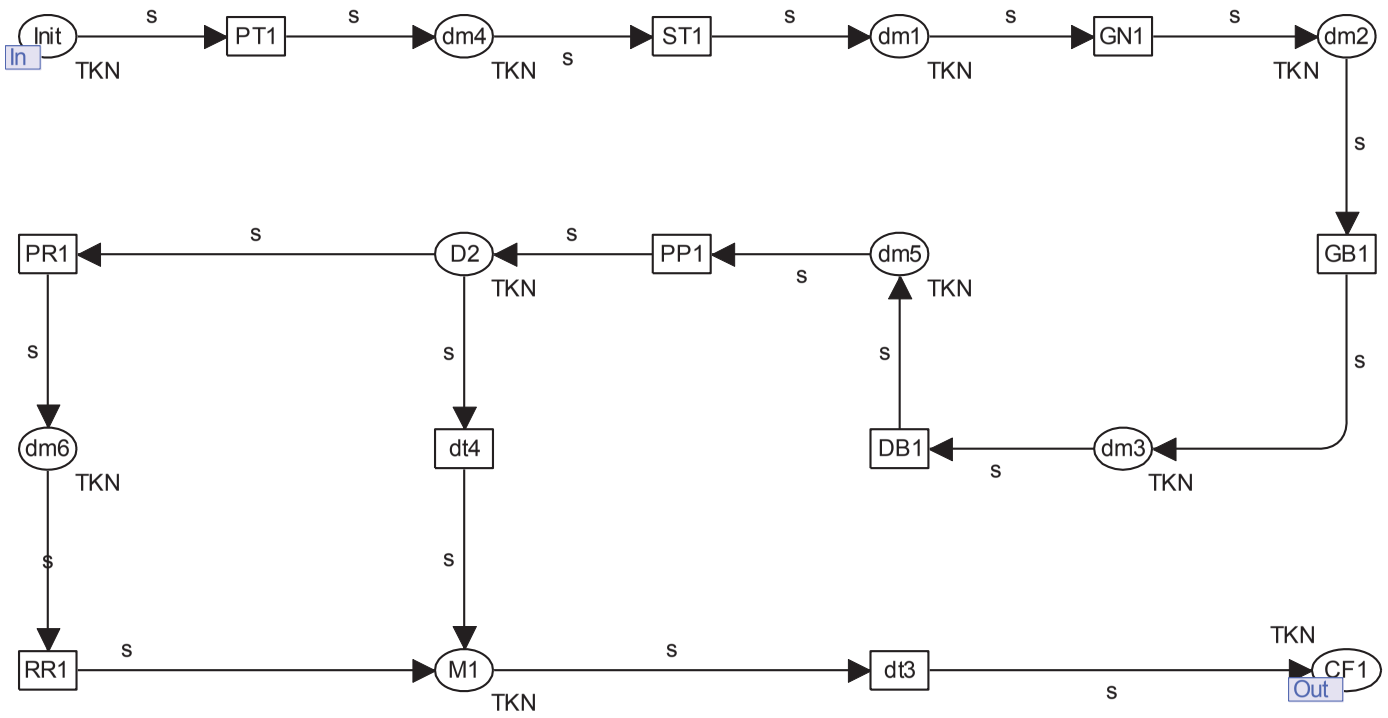


Figure A-3: Check balance

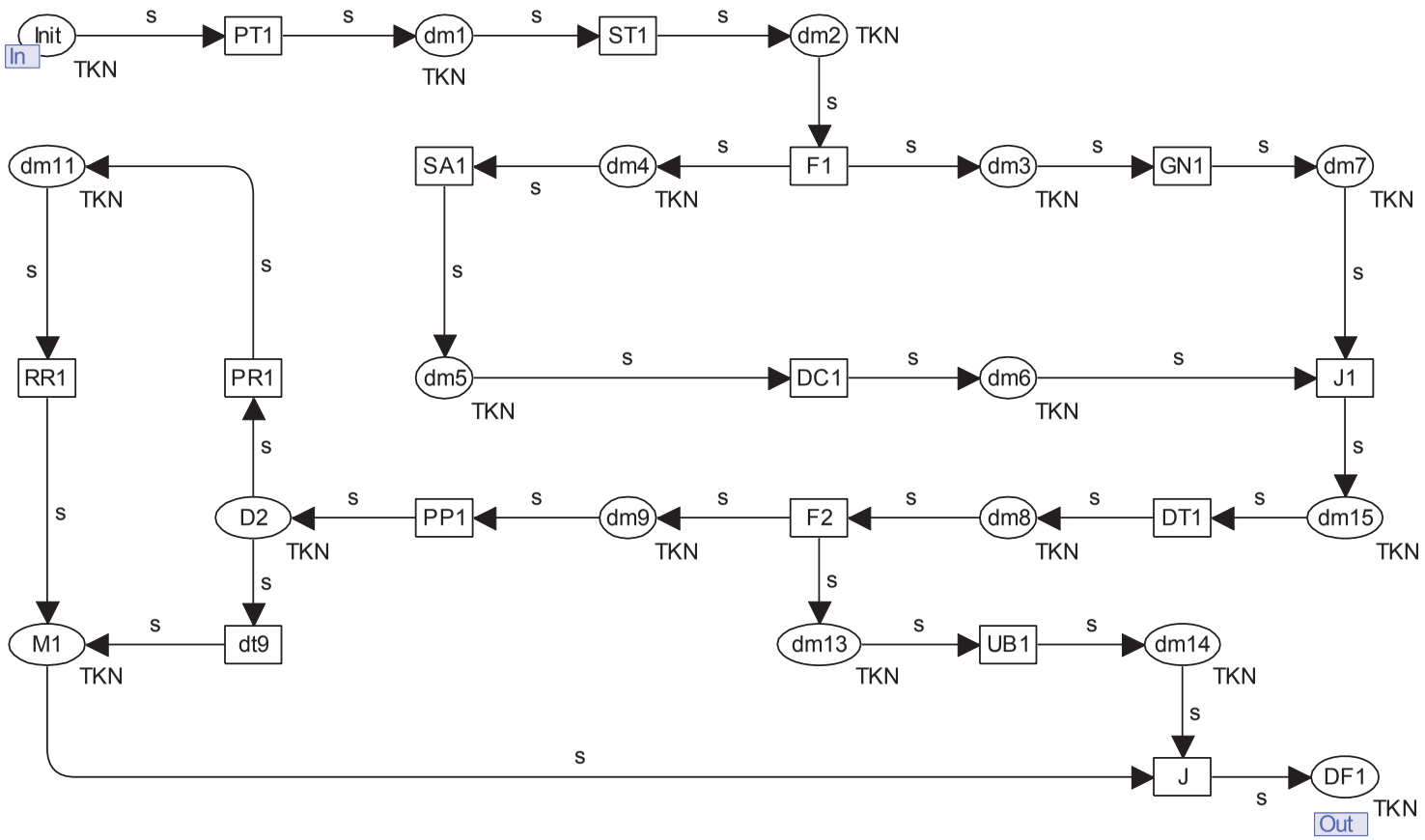


Figure A-4: Deposit cash

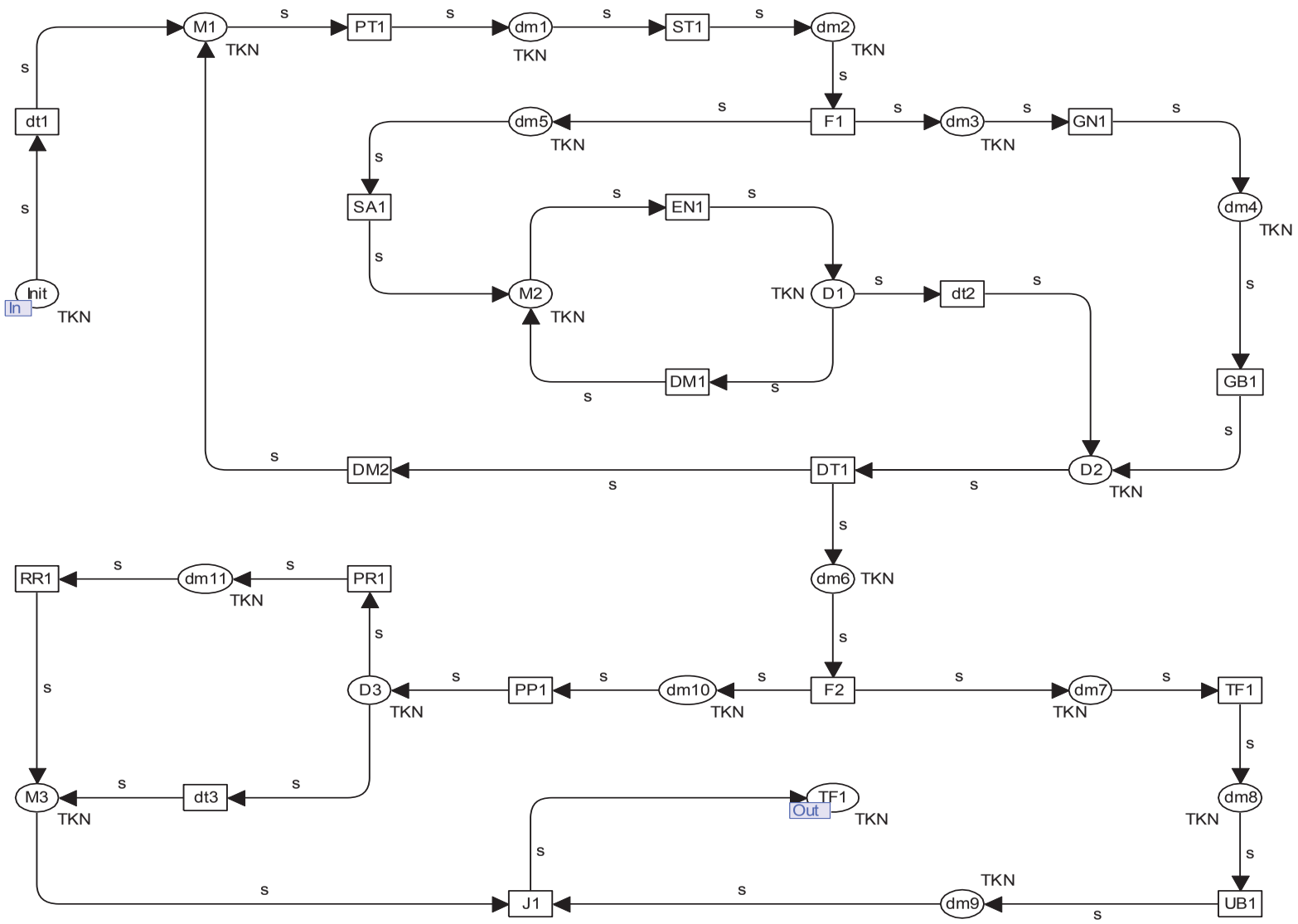


Figure A-5: Transfer funds

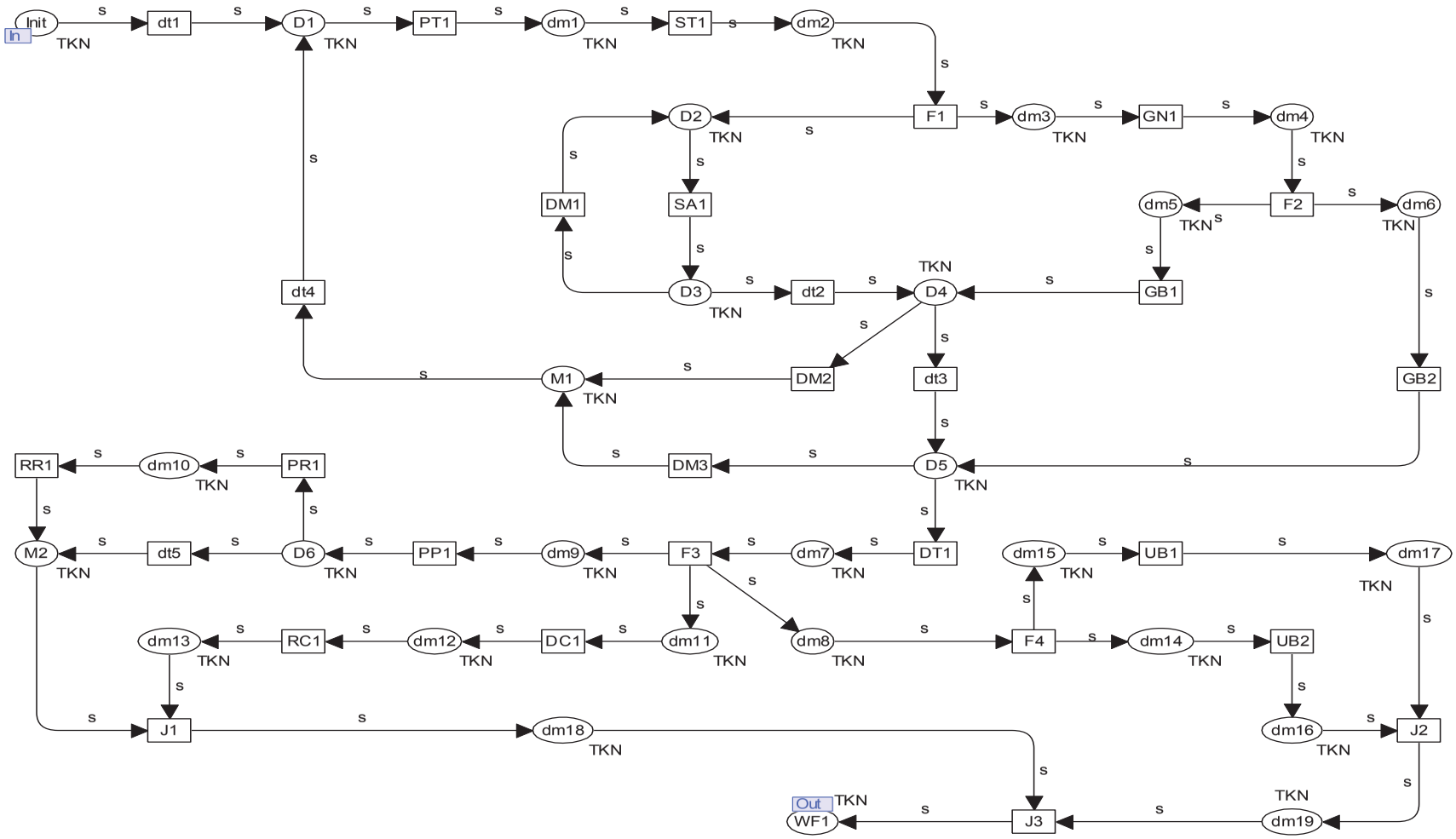


Figure A-6: Withdraw cash

c. Edit Trend Properties

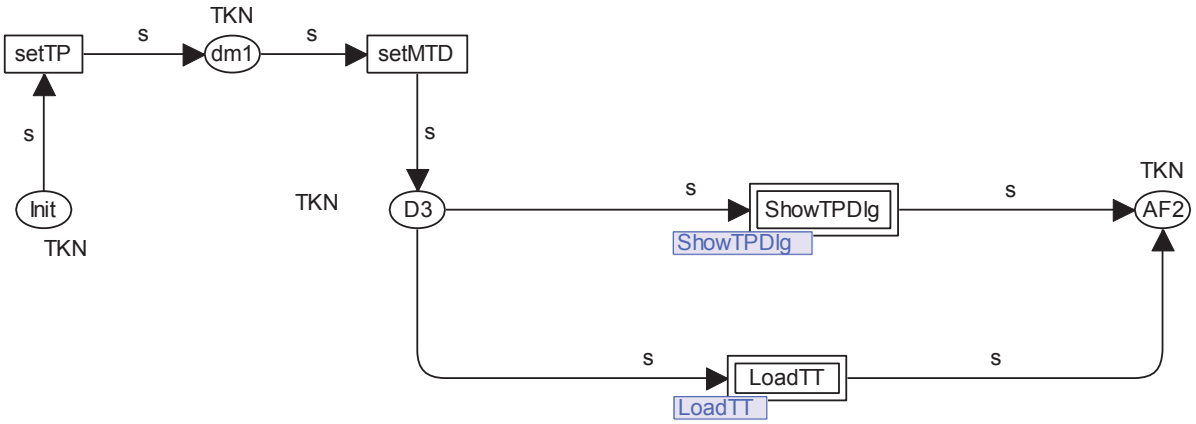


Figure A-7: ETP CPN model

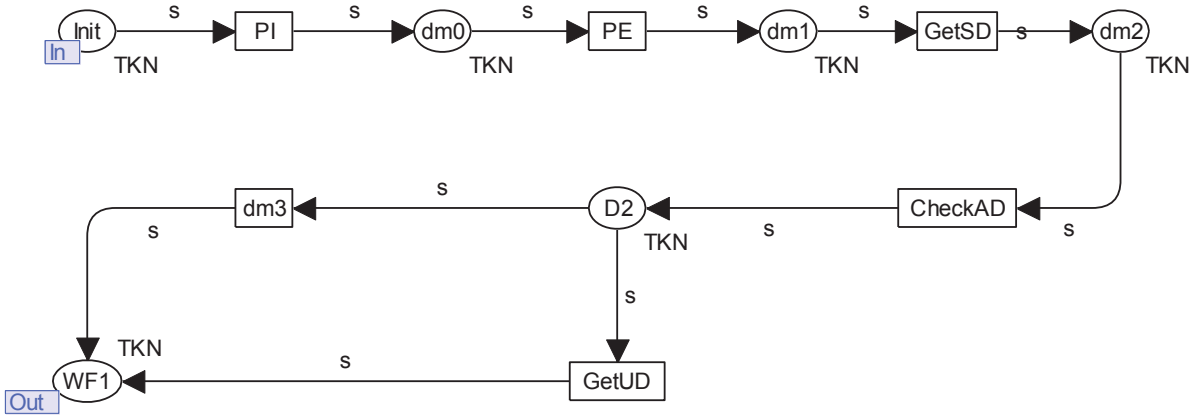


Figure A-8: ShowTPDialog

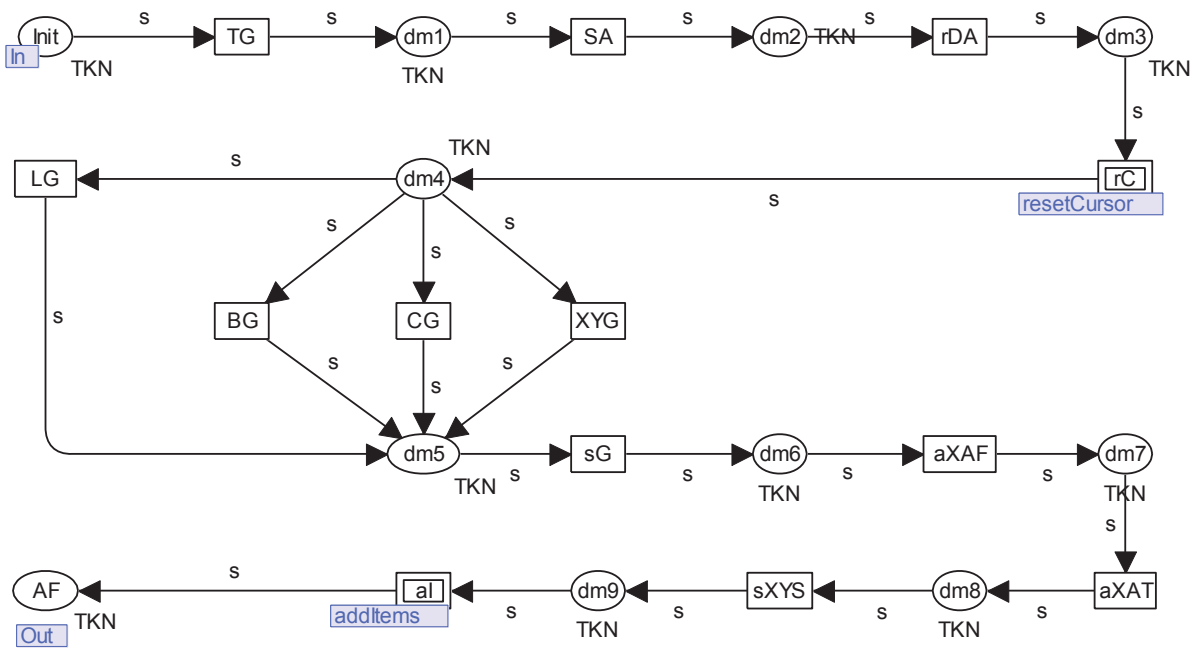


Figure A-9: LoadTT

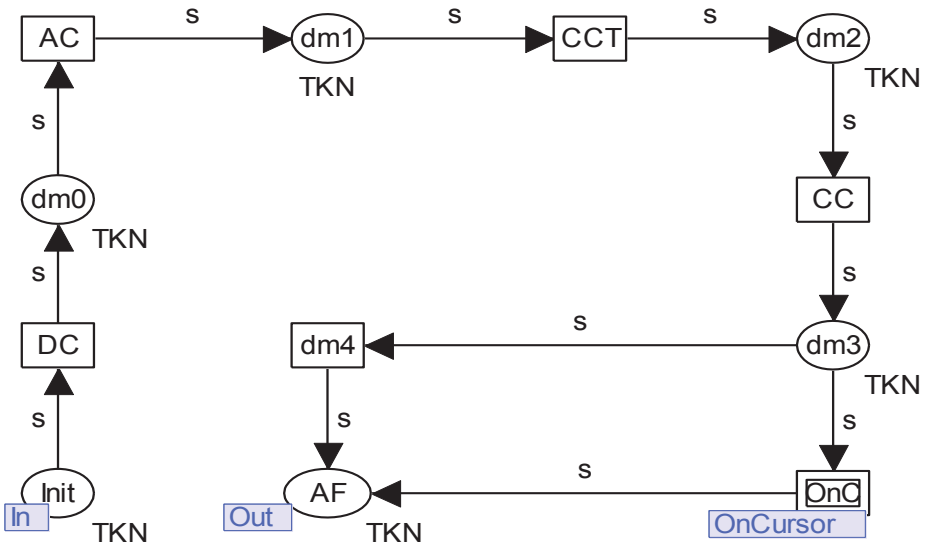


Figure A-10: ResetCursor

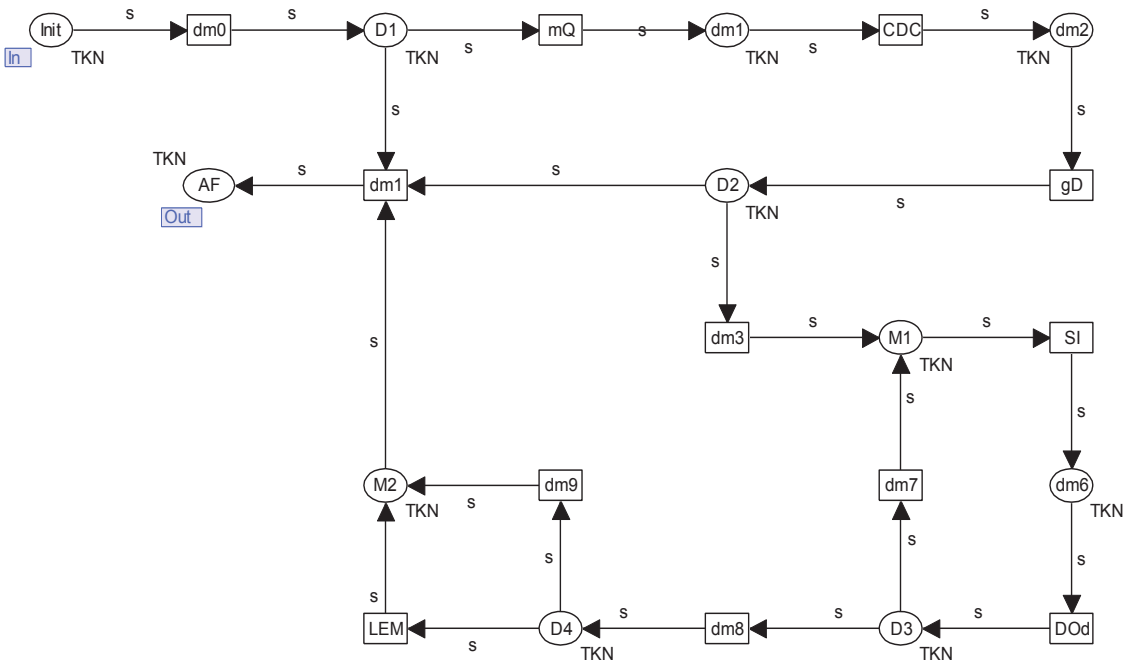


Figure A-11: OnCursor

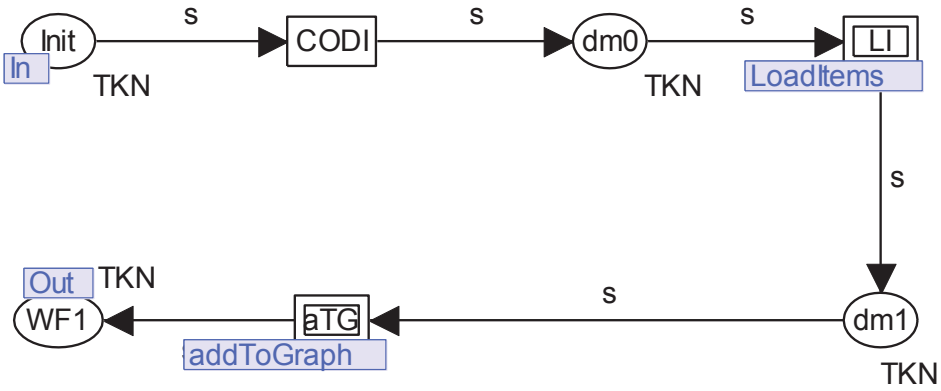


Figure A-12: AddItems

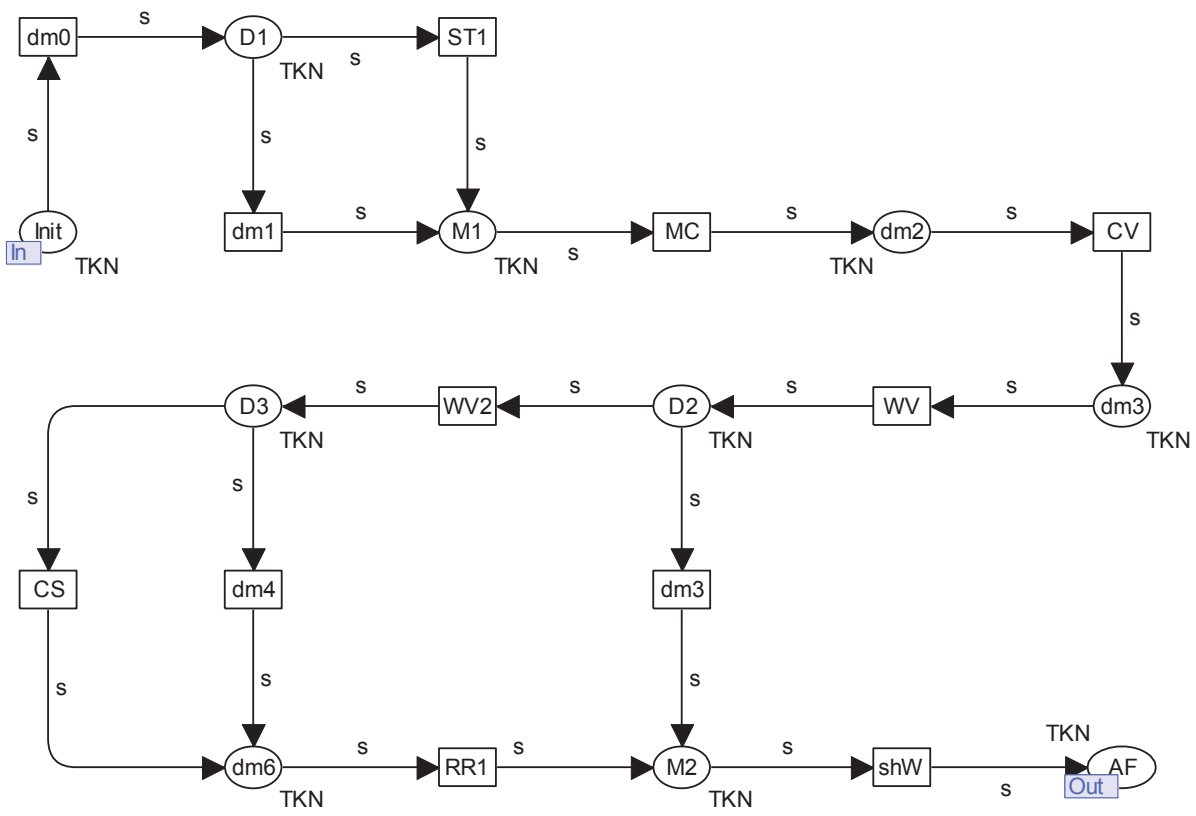


Figure A-13: LoadItems

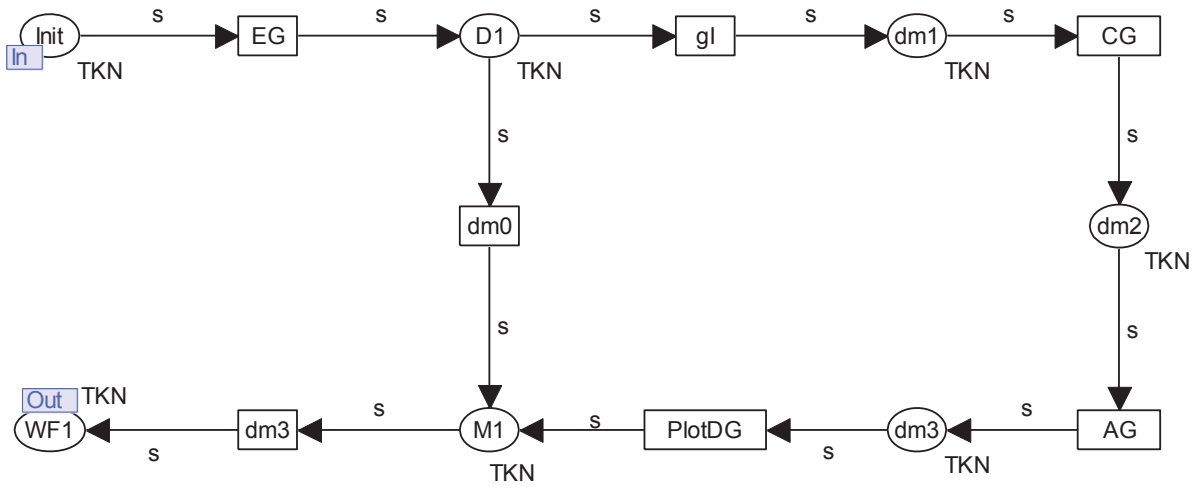


Figure A-14: addToGraph

d. Delete Trend Properties

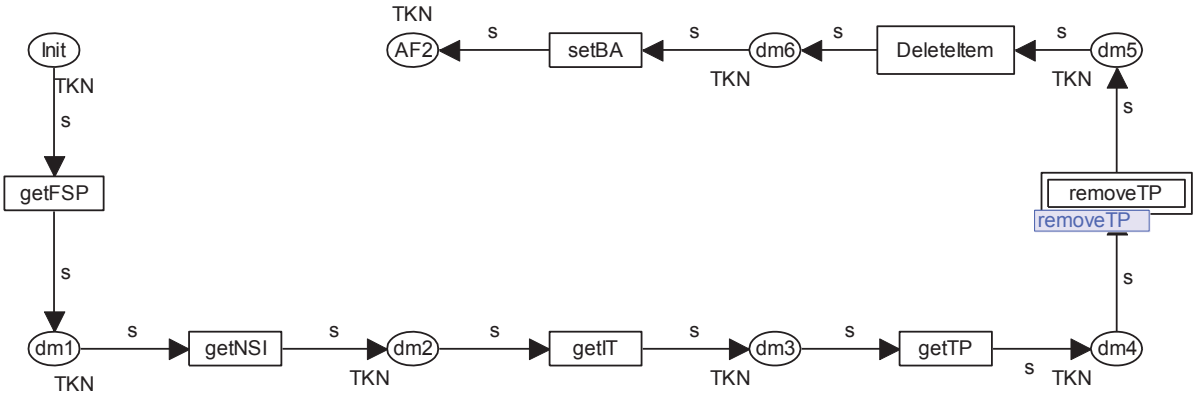


Figure A-15: Delete Trend Properties CPN model

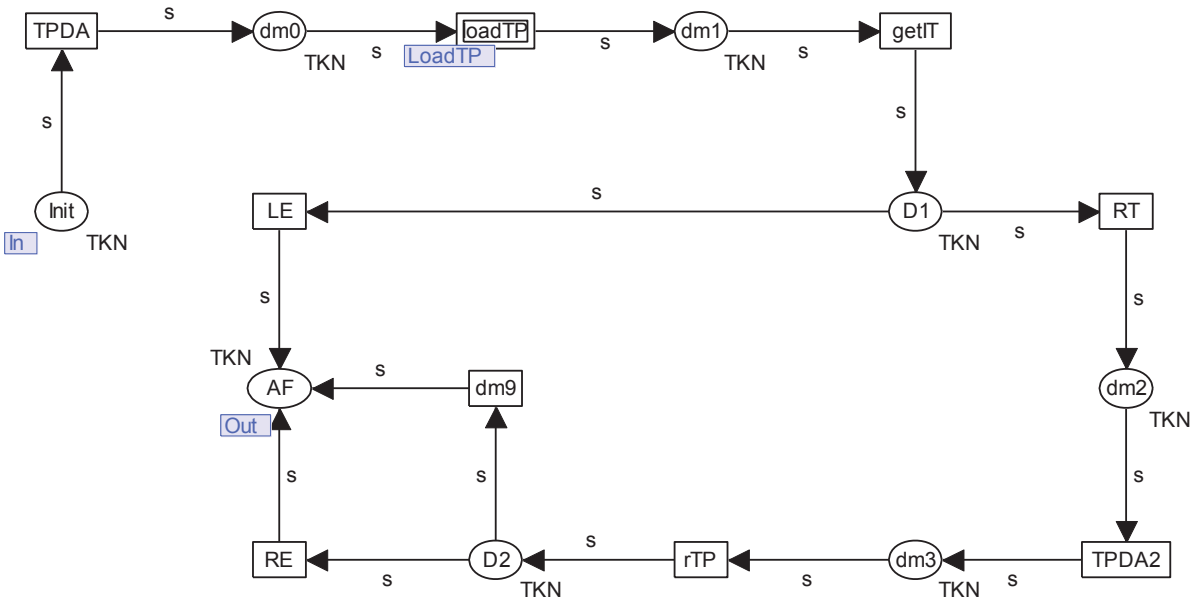


Figure A-16: Remove trend properties

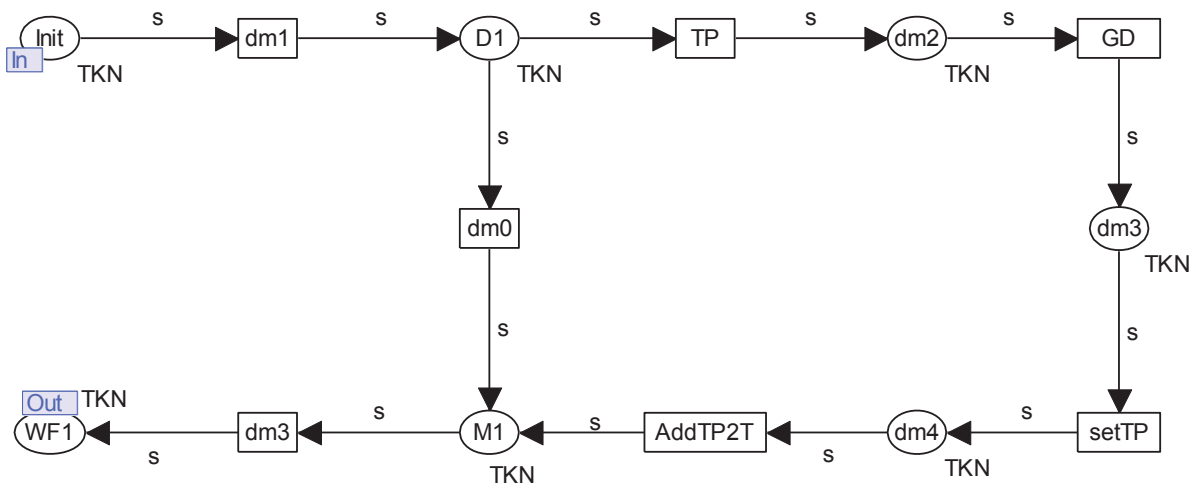


Figure A-17: Load trend properties

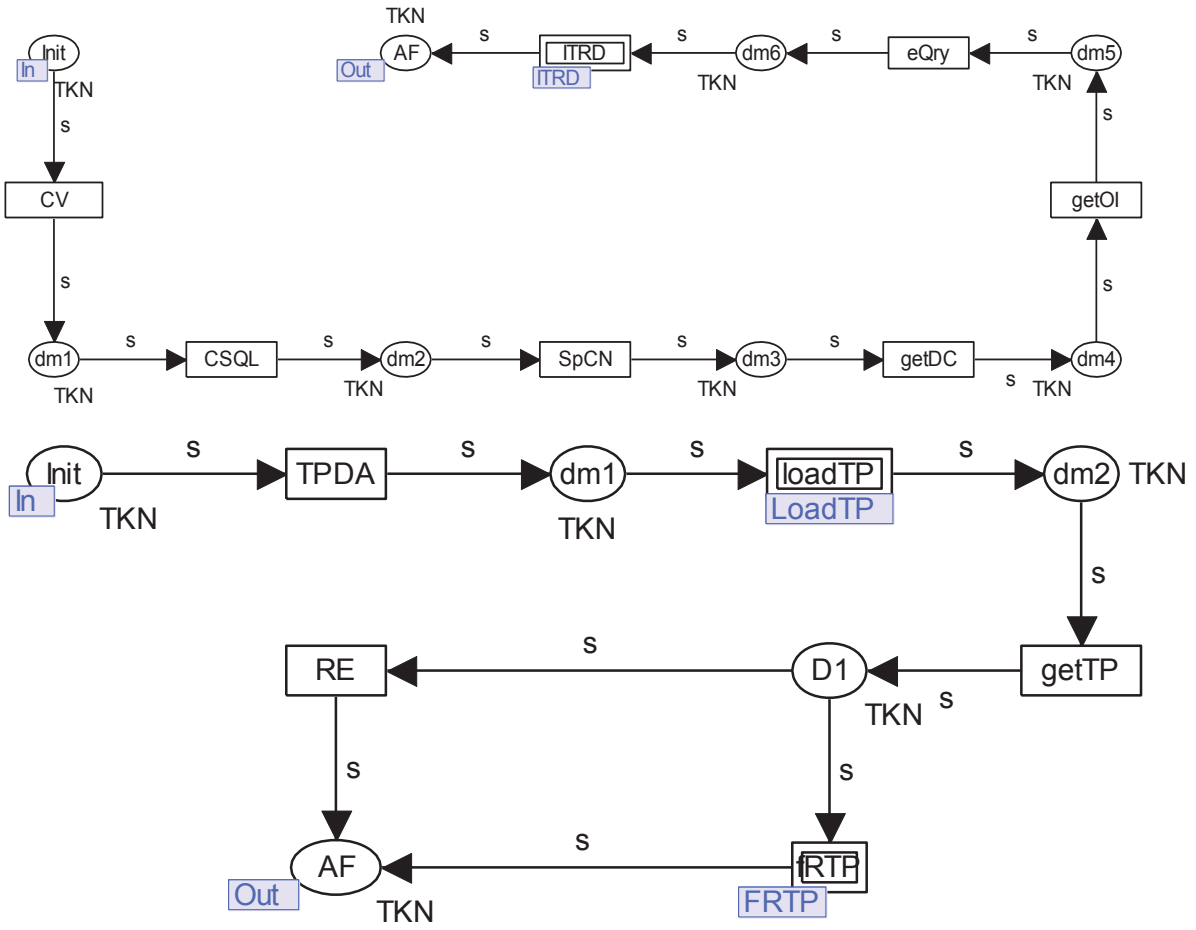


Figure A-18: Force remove trend properties

Appendix B. Mutant Model Generation

In Chapter 4, the mutation analysis is proposed for adequacy analysis of a test suite generated from an AD model. In order to perform such analysis, it is necessary to be able to generate mutant models by seeding simple faults into the original model. The set of fault types that can be seeded into a model constitute as the mutant operators. A mutation operator is a set of rules that describe syntactic changes into the model under test (MUT) to seed a particular type of faults. In the following, the XSLT-based implementation of the AD-based mutation operators defined in Chapter 4 is presented.

XML Metamodel Interchange (XMI) is an industry standard format for UML-based model document. Previously it is mentioned that the XSLT is a standard technology for transformation of XML based documents or artifacts and XSL is a standard language for specifying the output style and format which is mostly used along with XSLT. Here the XSL templates are used to define the mutation operators in the form of XSL transformation rules. The least advantage of using this approach is that the templates can be used with any programming language that has well defined Application Programming Interfaces (APIs) for XSL (i.e. .Net and JAXP) or with any independent XSL engine like the Saxon and the AltovaXML.

Mutating an AD model is similar to mutating a program source and a mutant model is generated from an original model for each application of a mutant operator. The XSL Processor parse the input model and produces output model according to the transformation rules (mutant operators). Thus, the implementation of mutant model generation of AD model effectively consists of three main components:

1) Original model in XML format – XMI file
2) One or more transformation rules (XSL templates)
3) One or more mutant models in XML format – XMI file

In the following section, an XSL template is defined for each mutant operator defined in Chapter 4. The XMI standard allows the tool specific data included in the <Extension> element. In order to view the mutant models in Enterprise Architecture templates are also provided to update the extension data related to changes made in <Model> element [EA 2008].

1. Mutant Operator Templates

1. Missing Action Operator Template

```
<!--     Missing Action Mutation     -->
  <xsl:template name="MissingActionMutation">
    <xsl:for-each select="$ModelActions">
    <xsl:variable name="MutantModel">
        <xsl:apply-templates select="$model" mode="MAO">
            <xsl:with-param name="NodeID" select="./@xmi:id" />
        </xsl:apply-templates>
    </xsl:variable>
    <xsl:result-document  method="xml"  href="OMO{position()}-AD-Mutant.xml">
        <xsl:sequence select="$MutantModel" />
    </xsl:result-document>
    </xsl:for-each>
  </xsl:template>
  <xsl:template match="*" mode="MAO">
  <xsl:param name="NodeID" />
    <xsl:choose>
        <xsl:when test="name()='edge' and ./@target=$NodeID">
            <xsl:call-template name="update-edges-for-MAO">
                <xsl:with-param name="NodeID" select="$NodeID" />
            </xsl:call-template>
        </xsl:when>
        <xsl:when test="name()='target' and ./@xmi:idref=$NodeID">
            <xsl:call-template name="update-target-for-MAO">
                <xsl:with-param name="NodeID" select="$NodeID" />
            </xsl:call-template>
        </xsl:when>
        <xsl:otherwise>
        <xsl:copy>
            <xsl:copy-of select="@*"/>
            <xsl:apply-templates mode="MAO">
                <xsl:with-param name="NodeID" select="$NodeID" />
            </xsl:apply-templates>
        </xsl:copy>
        </xsl:otherwise>
    </xsl:choose>
  </xsl:template>
  <xsl:template name="update-edges-for-MAO">
    <xsl:param name="NodeID" />
    <xsl:variable name="EdgesWithNodeAsSource" select="$ModelEdges[@source=$NodeID]" />
    <xsl:if test="./@target=$NodeID">
        <xsl:copy>
            <xsl:copy-of select="@* except @target"/>
            <xsl:attribute name="target">
                <xsl:value-of select="$EdgesWithNodeAsSource[1]/@target" />
            </xsl:attribute>
        </xsl:copy>
    </xsl:if>
  </xsl:template>
```

```
<!-- template to see changes in EA -->
<xsl:template name="update-target-for-MAO">
  <xsl:param name="NodeID" />
  <xsl:variable name="EdgesWithNodeAsSource" select="$ModelEdges[@source=$NodeID]" />
  <xsl:if test="./@xmi:idref=$NodeID">
      <xsl:copy>
          <xsl:copy-of select="@* except @xmi:idref"/>
          <xsl:attribute name="xmi:idref">
              <xsl:value-of select="$EdgesWithNodeAsSource[1]/@target" />
          </xsl:attribute>
          <xsl:apply-templates mode="MAO" />
      </xsl:copy>
  </xsl:if>
</xsl:template>
<!--    End Missing Action Mutation    -->
```

2. Actions Exchanged Operator Template

```
<!--     Actions Exchanged Mutation     -->
 <xsl:template name="ActionsExchangedMutation">
   <xsl:for-each select="$ModelActions">
      <xsl:variable name="MutantModel">
      <xsl:apply-templates select="$model" mode="AEO">
          <xsl:with-param name="NodeID" select="./@xmi:id" />
          <xsl:with-param name="RandomNodeID">
              <xsl:variable name="RandomNodePosition">
                  <xsl:call-template name="randomVal">
                      <xsl:with-param name="bound" select="count($ModelActions)" />
                  </xsl:call-template>
              </xsl:variable>
              <xsl:value-of select="$ModelActions[number($RandomNodePosition)]/@xmi:id" />
          </xsl:with-param>
      </xsl:apply-templates>
   </xsl:variable>
   <xsl:result-document  method="xml"  href="AEO{position()}-AD-Mutant.xml">
      <xsl:sequence select="$MutantModel" />
   </xsl:result-document>
   </xsl:for-each>
 </xsl:template>
 <xsl:template match="*" mode="AEO">
   <xsl:param name="NodeID" />
   <xsl:param name="RandomNodeID" />
   <xsl:choose>
      <xsl:when test="name()='edge' and (./@source=$NodeID or ./@source=$RandomNodeID
      or ./@target=$NodeID or ./@target=$RandomNodeID)">
          <xsl:call-template name="update-edges-for-AEO">
              <xsl:with-param name="NodeID" select="$NodeID" />
              <xsl:with-param name="RandomNodeID" select="$RandomNodeID" />
          </xsl:call-template>
      </xsl:when>
      <xsl:when test="(name()='source' or name()='target') and (./@xmi:idref=$NodeID or ./@xmi:idref=$RandomNodeID)">
          <xsl:call-template name="update-sourceANDtarget-for-AEO">
              <xsl:with-param name="NodeID" select="$NodeID" />
              <xsl:with-param name="RandomNodeID" select="$RandomNodeID" />
          </xsl:call-template>
      </xsl:when>
      <xsl:otherwise>
      <xsl:copy>
          <xsl:copy-of select="@*"/>
          <xsl:apply-templates mode="AEO">
              <xsl:with-param name="NodeID" select="$NodeID" />
              <xsl:with-param name="RandomNodeID" select="$RandomNodeID" />
          </xsl:apply-templates>
      </xsl:copy>
      </xsl:otherwise>
   </xsl:choose>
 </xsl:template>
```

```
<xsl:template name="update-edges-for-AEO">
  <xsl:param name="NodeID" />
  <xsl:param name="RandomNodeID" />
  <xsl:choose>
      <xsl:when test="./@source=$NodeID">
          <xsl:copy>
              <xsl:copy-of select="@* except @source"/>
              <xsl:attribute name="source">
                  <xsl:value-of select="$RandomNodeID" />
              </xsl:attribute>
          </xsl:copy>
      </xsl:when>
      <xsl:when test="./@source=$RandomNodeID">
          <xsl:copy>
              <xsl:copy-of select="@* except @source"/>
              <xsl:attribute name="source">
                  <xsl:value-of select="$NodeID" />
              </xsl:attribute>
          </xsl:copy>
      </xsl:when>
      <xsl:when test="./@target=$NodeID">
          <xsl:copy>
              <xsl:copy-of select="@* except @target"/>
              <xsl:attribute name="target">
                  <xsl:value-of select="$RandomNodeID" />
              </xsl:attribute>
          </xsl:copy>
      </xsl:when>
      <xsl:when test="./@target=$RandomNodeID">
          <xsl:copy>
              <xsl:copy-of select="@* except @target"/>
              <xsl:attribute name="target">
                  <xsl:value-of select="$NodeID" />
              </xsl:attribute>
          </xsl:copy>
      </xsl:when>
      <xsl:otherwise />
  </xsl:choose>
</xsl:template>
```

```
<!-- template to see changes in EA -->
<xsl:template name="update-sourceANDtarget-for-AEO">
  <xsl:param name="NodeID" />
  <xsl:param name="RandomNodeID" />
  <xsl:choose>
      <xsl:when test="./@xmi:idref=$NodeID">
          <xsl:copy>
              <xsl:copy-of select="@* except @xmi:idref"/>
              <xsl:attribute name="xmi:idref">
                  <xsl:value-of select="$RandomNodeID" />
              </xsl:attribute>
              <xsl:apply-templates mode="AEO" />
          </xsl:copy>
      </xsl:when>
      <xsl:when test="./@xmi:idref=$RandomNodeID">
          <xsl:copy>
              <xsl:copy-of select="@* except @xmi:idref"/>
              <xsl:attribute name="xmi:idref">
                  <xsl:value-of select="$NodeID" />
              </xsl:attribute>
              <xsl:apply-templates mode="AEO" />
          </xsl:copy>
      </xsl:when>
      <xsl:otherwise />
  </xsl:choose>
</xsl:template>
<!--   End Actions eXchange Mutation   -->
```

3. Extra Inflow Operator Template

```
<!--     Extra Inflow Mutation     -->
 <xsl:template name="ExtraInflowMutation">
   <xsl:for-each select="$ModelActions">
       <xsl:variable name="MutantModel">
       <xsl:apply-templates select="$model" mode="EIO">
           <xsl:with-param name="NodeID" select="./@xmi:id" />
           <xsl:with-param name="RandomNodeID">
               <xsl:variable name="RandomNodePosition">
                   <xsl:call-template name="randomVal">
                       <xsl:with-param name="bound" select="count($ModelActions)" />
                   </xsl:call-template>
               </xsl:variable>
               <xsl:value-of select="$ModelActions[number($RandomNodePosition)]/@xmi:id" />
           </xsl:with-param>
       </xsl:apply-templates>
   </xsl:variable>
   <xsl:result-document  method="xml"  href="EIO{position()}-AD-Mutant.xml">
       <xsl:sequence select="$MutantModel" />
   </xsl:result-document>
   </xsl:for-each>
 </xsl:template>
 <xsl:template match="*" mode="EIO">
   <xsl:param name="NodeID" />
   <xsl:param name="RandomNodeID" />
   <xsl:choose>
       <xsl:when test="name()='edge' and (./@target=$NodeID)">
           <xsl:call-template name="update-edges-for-EIO">
               <xsl:with-param name="NodeID" select="$NodeID" />
               <xsl:with-param name="RandomNodeID" select="$RandomNodeID" />
           </xsl:call-template>
       </xsl:when>
       <xsl:otherwise>
       <xsl:copy>
           <xsl:copy-of select="@*"/>
           <xsl:apply-templates mode="EIO">
               <xsl:with-param name="NodeID" select="$NodeID" />
               <xsl:with-param name="RandomNodeID" select="$RandomNodeID" />
           </xsl:apply-templates>
       </xsl:copy>
       </xsl:otherwise>
   </xsl:choose>
 </xsl:template>

 <xsl:template name="update-edges-for-EIO">
   <xsl:param name="NodeID" />
   <xsl:param name="RandomNodeID" />
   <xsl:choose>
       <xsl:when test="./@target=$NodeID">
           <xsl:copy>
               <xsl:copy-of select="@*"/>
           </xsl:copy>
           <xsl:copy>
               <xsl:copy-of select="@* except @xmi:id, @source"/>
               <xsl:attribute name="xmi:id">
                   <xsl:value-of select="$RandomNodeID" />
               </xsl:attribute>
               <xsl:attribute name="source">
                   <xsl:value-of select="$RandomNodeID" />
               </xsl:attribute>
           </xsl:copy>
       </xsl:when>
       <xsl:otherwise />
   </xsl:choose>
 </xsl:template>
<!--   End Extra Inflow Mutation   -->
```

4. Extra Inflow Operator Template

```
<!--     Extra Outflow Mutation     -->
 <xsl:template name="ExtraOutflowMutation">
   <xsl:for-each select="$ModelActions">
       <xsl:variable name="MutantModel">
       <xsl:apply-templates select="$model" mode="EOO">
           <xsl:with-param name="NodeID" select="./@xmi:id" />
           <xsl:with-param name="RandomNodeID">
               <xsl:variable name="RandomNodePosition">
                   <xsl:call-template name="randomVal">
                       <xsl:with-param name="bound" select="count($ModelActions)" />
                   </xsl:call-template>
               </xsl:variable>
               <xsl:value-of select="$ModelActions[number($RandomNodePosition)]/@xmi:id" />
           </xsl:with-param>
       </xsl:apply-templates>
   </xsl:variable>
   <xsl:result-document  method="xml"  href="EOO{position()}-AD-Mutant.xml">
       <xsl:sequence select="$MutantModel" />
   </xsl:result-document>
   </xsl:for-each>
 </xsl:template>
 <xsl:template match="*" mode="EOO">
   <xsl:param name="NodeID" />
   <xsl:param name="RandomNodeID" />
   <xsl:choose>
       <xsl:when test="name()='edge' and (./@source=$NodeID)">
           <xsl:call-template name="update-edges-for-EOO">
               <xsl:with-param name="NodeID" select="$NodeID" />
               <xsl:with-param name="RandomNodeID" select="$RandomNodeID" />
           </xsl:call-template>
       </xsl:when>
       <xsl:otherwise>
       <xsl:copy>
           <xsl:copy-of select="@*"/>
           <xsl:apply-templates mode="EOO">
               <xsl:with-param name="NodeID" select="$NodeID" />
               <xsl:with-param name="RandomNodeID" select="$RandomNodeID" />
           </xsl:apply-templates>
       </xsl:copy>
       </xsl:otherwise>
   </xsl:choose>
 </xsl:template>

 <xsl:template name="update-edges-for-EOO">
   <xsl:param name="NodeID" />
   <xsl:param name="RandomNodeID" />
   <xsl:choose>
       <xsl:when test="./@source=$NodeID">
           <xsl:copy>
               <xsl:copy-of select="@*"/>
           </xsl:copy>
           <xsl:copy>
               <xsl:copy-of select="@* except @xmi:id, @target"/>
               <xsl:attribute name="xmi:id">
                   <xsl:value-of select="$RandomNodeID" />
               </xsl:attribute>
               <xsl:attribute name="target">
                   <xsl:value-of select="$RandomNodeID" />
               </xsl:attribute>
           </xsl:copy>
       </xsl:when>
       <xsl:otherwise />
   </xsl:choose>
 </xsl:template>
<!--    End Extra Outflow Mutation    -->
```

5. Inflow Exchanged Operator Template

```
<!--      Inflow Exchanged Mutation      -->
 <xsl:template name="InflowExchangedMutation">
   <xsl:for-each select="$ModelActions">
       <xsl:variable name="MutantModel">
       <xsl:apply-templates select="$model" mode="IEO">
           <xsl:with-param name="NodeID" select="./@xmi:id" />
           <xsl:with-param name="RandomNodeID">
               <xsl:variable name="RandomNodePosition">
                   <xsl:call-template name="randomVal">
                       <xsl:with-param name="bound" select="count($ModelActions)" />
                   </xsl:call-template>
               </xsl:variable>
               <xsl:value-of select="$ModelActions[number($RandomNodePosition)]/@xmi:id" />
           </xsl:with-param>
       </xsl:apply-templates>
   </xsl:variable>
   <xsl:result-document  method="xml"  href="IEO{position()}-AD-Mutant.xml">
       <xsl:sequence select="$MutantModel" />
   </xsl:result-document>
   </xsl:for-each>
 </xsl:template>
 <xsl:template match="*" mode="IEO">
   <xsl:param name="NodeID" />
   <xsl:param name="RandomNodeID" />
   <xsl:choose>
       <xsl:when test="name()='edge' and (./@target=$NodeID or ./@target=$RandomNodeID)">
           <xsl:call-template name="update-edges-for-IEO">
               <xsl:with-param name="NodeID" select="$NodeID" />
               <xsl:with-param name="RandomNodeID" select="$RandomNodeID" />
           </xsl:call-template>
       </xsl:when>
       <xsl:when test="(name()='target') and (./@xmi:idref=$NodeID or ./@xmi:idref=$RandomNodeID)">
           <xsl:call-template name="update-target-for-IEO">
               <xsl:with-param name="NodeID" select="$NodeID" />
               <xsl:with-param name="RandomNodeID" select="$RandomNodeID" />
           </xsl:call-template>
       </xsl:when>
       <xsl:otherwise>
       <xsl:copy>
           <xsl:copy-of select="@*"/>
           <xsl:apply-templates mode="IEO">
               <xsl:with-param name="NodeID" select="$NodeID" />
               <xsl:with-param name="RandomNodeID" select="$RandomNodeID" />
           </xsl:apply-templates>
       </xsl:copy>
       </xsl:otherwise>
   </xsl:choose>
 </xsl:template>
```

```
<xsl:template name="update-edges-for-IEO">
  <xsl:param name="NodeID" />
  <xsl:param name="RandomNodeID" />
  <xsl:choose>
      <xsl:when test="./@source=$NodeID">
          <xsl:copy>
              <xsl:copy-of select="@* except @source"/>
              <xsl:attribute name="source">
                  <xsl:value-of select="$RandomNodeID" />
              </xsl:attribute>
          </xsl:copy>
      </xsl:when>
      <xsl:when test="./@source=$RandomNodeID">
          <xsl:copy>
              <xsl:copy-of select="@* except @source"/>
              <xsl:attribute name="source">
                  <xsl:value-of select="$NodeID" />
              </xsl:attribute>
          </xsl:copy>
      </xsl:when>
      <xsl:when test="./@target=$NodeID">
          <xsl:copy>
              <xsl:copy-of select="@* except @target"/>
              <xsl:attribute name="target">
                  <xsl:value-of select="$RandomNodeID" />
              </xsl:attribute>
          </xsl:copy>
      </xsl:when>
      <xsl:when test="./@target=$RandomNodeID">
          <xsl:copy>
              <xsl:copy-of select="@* except @target"/>
              <xsl:attribute name="target">
                  <xsl:value-of select="$NodeID" />
              </xsl:attribute>
          </xsl:copy>
      </xsl:when>
      <xsl:otherwise />
  </xsl:choose>
</xsl:template>

<!-- template to see changes in EA -->
<xsl:template name="update-target-for-IEO">
  <xsl:param name="NodeID" />
  <xsl:param name="RandomNodeID" />
  <xsl:choose>
      <xsl:when test="./@xmi:idref=$NodeID">
          <xsl:copy>
              <xsl:copy-of select="@* except @xmi:idref"/>
              <xsl:attribute name="xmi:idref">
                  <xsl:value-of select="$RandomNodeID" />
              </xsl:attribute>
              <xsl:apply-templates mode="IEO" />
          </xsl:copy>
      </xsl:when>
      <xsl:when test="./@xmi:idref=$RandomNodeID">
          <xsl:copy>
              <xsl:copy-of select="@* except @xmi:idref"/>
              <xsl:attribute name="xmi:idref">
                  <xsl:value-of select="$NodeID" />
              </xsl:attribute>
              <xsl:apply-templates mode="IEO" />
          </xsl:copy>
      </xsl:when>
      <xsl:otherwise />
  </xsl:choose>
</xsl:template>
<!--   End Inflow Exchange Mutation   -->
```

6. Missing Join Operator Template

```
<!--      Missing Join Mutation      -->
 <xsl:template name="MissingJoinMutation">
   <xsl:for-each select="$ModelJoins">
   <xsl:variable name="MutantModel">
       <xsl:apply-templates select="$model" mode="MJO">
           <xsl:with-param name="NodeID" select="./@xmi:id" />
       </xsl:apply-templates>
   </xsl:variable>
   <xsl:result-document  method="xml"  href="MJO{position()}-AD-Mutant.xml">
       <xsl:sequence select="$MutantModel" />
   </xsl:result-document>
   </xsl:for-each>
 </xsl:template>
 <xsl:template match="*" mode="MJO">
 <xsl:param name="NodeID" />
   <xsl:choose>
       <xsl:when test="name()='edge' and ./@target=$NodeID">
           <xsl:call-template name="update-edges-for-MJO">
               <xsl:with-param name="NodeID" select="$NodeID" />
           </xsl:call-template>
       </xsl:when>
       <xsl:when test="name()='target' and ./@xmi:idref=$NodeID">
           <xsl:call-template name="update-target-for-MJO">
               <xsl:with-param name="NodeID" select="$NodeID" />
           </xsl:call-template>
       </xsl:when>
       <xsl:otherwise>
       <xsl:copy>
           <xsl:copy-of select="@*"/>
           <xsl:apply-templates mode="MJO">
               <xsl:with-param name="NodeID" select="$NodeID" />
           </xsl:apply-templates>
       </xsl:copy>
       </xsl:otherwise>
   </xsl:choose>
 </xsl:template>
 <xsl:template name="update-edges-for-MJO">
   <xsl:param name="NodeID" />
   <xsl:variable name="EdgesWithNodeAsSource" select="$ModelEdges[@source=$NodeID]" />
   <xsl:if test="./@target=$NodeID">
       <xsl:copy>
           <xsl:copy-of select="@* except @target"/>
           <xsl:attribute name="target">
               <xsl:value-of select="$EdgesWithNodeAsSource[1]/@target" />
           </xsl:attribute>
       </xsl:copy>
   </xsl:if>
 </xsl:template>
```

```
<!-- template to see changes in EA -->
<xsl:template name="update-target-for-MJO">
  <xsl:param name="NodeID" />
  <xsl:variable name="EdgesWithNodeAsSource" select="$ModelEdges[@source=$NodeID]" />
  <xsl:if test="./@xmi:idref=$NodeID">
      <xsl:copy>
          <xsl:copy-of select="@* except @xmi:idref"/>
          <xsl:attribute name="xmi:idref">
              <xsl:value-of select="$EdgesWithNodeAsSource[1]/@target" />
          </xsl:attribute>
          <xsl:apply-templates mode="MJO" />
      </xsl:copy>
  </xsl:if>
</xsl:template>
<!--   End Missing Join Mutation   -->
```

7. Invalid Synchronization Operator Template

```
<!--     Invalid Synchronization Mutation     -->
<xsl:template name="InvalidSynchronizationMutation">
  <xsl:for-each select="$ModelJoins">
      <xsl:variable name="MutantModel">
      <xsl:apply-templates select="$model" mode="ISO">
          <xsl:with-param name="EdgeID" select="./incoming/@xmi:idref" />
          <xsl:with-param name="RandomNodeID">
              <xsl:variable name="RandomNodePosition">
                  <xsl:call-template name="randomVal">
                      <xsl:with-param name="bound" select="count($ModelDecisions)" />
                  </xsl:call-template>
              </xsl:variable>
              <xsl:value-of select="$ModelDecisions[number($RandomNodePosition)]/@xmi:id" />
          </xsl:with-param>
      </xsl:apply-templates>
  </xsl:variable>
  <xsl:result-document  method="xml"  href="ISO{position()}-AD-Mutant.xml">
      <xsl:sequence select="$MutantModel" />
  </xsl:result-document>
  </xsl:for-each>
</xsl:template>
<xsl:template match="*" mode="ISO">
  <xsl:param name="EdgeID" />
  <xsl:param name="RandomNodeID" />
  <xsl:choose>
      <xsl:when test="name()='edge' and (./@xmi:id=$EdgeID)">
          <xsl:call-template name="update-edges-for-ISO">
              <xsl:with-param name="RandomNodeID" select="$RandomNodeID" />
          </xsl:call-template>
      </xsl:when>
      <xsl:otherwise>
      <xsl:copy>
          <xsl:copy-of select="@*"/>
          <xsl:apply-templates mode="ISO">
              <xsl:with-param name="EdgeID" select="$EdgeID" />
              <xsl:with-param name="RandomNodeID" select="$RandomNodeID" />
          </xsl:apply-templates>
      </xsl:copy>
      </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

```
<xsl:template name="update-edges-for-ISO">
  <xsl:param name="RandomNodeID" />
  <xsl:copy>
      <xsl:copy-of select="@*"/>
  </xsl:copy>
  <xsl:copy>
      <xsl:copy-of select="@* except @xmi:id, @source"/>
      <xsl:attribute name="xmi:id">
          <xsl:value-of select="$RandomNodeID" />
      </xsl:attribute>
      <xsl:attribute name="source">
          <xsl:value-of select="$RandomNodeID" />
      </xsl:attribute>
  </xsl:copy>
</xsl:template>
<!--    End Invalid Synchronization Mutation    -->
```

8. Extra Branch Operator Template

```
<!--      Extra Branch Mutation      -->
<xsl:template name="ExtraBranchMutation">
  <xsl:for-each select="$ModelDecisions">
      <xsl:variable name="MutantModel">
      <xsl:apply-templates select="$model" mode="EBO">
          <xsl:with-param name="EdgeID" select="./outgoing/@xmi:idref" />
          <xsl:with-param name="RandomNodeID">
              <xsl:variable name="RandomNodePosition">
                  <xsl:call-template name="randomVal">
                      <xsl:with-param name="bound" select="count($ModelActions)" />
                  </xsl:call-template>
              </xsl:variable>
              <xsl:value-of select="$ModelActions[number($RandomNodePosition)]/@xmi:id" />
          </xsl:with-param>
      </xsl:apply-templates>
  </xsl:variable>
  <xsl:result-document  method="xml"  href="EBO{position()}-AD-Mutant.xml">
      <xsl:sequence select="$MutantModel" />
  </xsl:result-document>
  </xsl:for-each>
</xsl:template>
<xsl:template match="*" mode="EBO">
  <xsl:param name="EdgeID" />
  <xsl:param name="RandomNodeID" />
  <xsl:choose>
      <xsl:when test="name()='edge' and (./@xmi:id=$EdgeID)">
          <xsl:call-template name="update-edges-for-EBO">
              <xsl:with-param name="RandomNodeID" select="$RandomNodeID" />
          </xsl:call-template>
      </xsl:when>
      <xsl:otherwise>
      <xsl:copy>
          <xsl:copy-of select="@*"/>
          <xsl:apply-templates mode="EBO">
              <xsl:with-param name="EdgeID" select="$EdgeID" />
              <xsl:with-param name="RandomNodeID" select="$RandomNodeID" />
          </xsl:apply-templates>
      </xsl:copy>
      </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

```
 <xsl:template name="update-edges-for-EBO">
   <xsl:param name="RandomNodeID" />
   <xsl:copy>
       <xsl:copy-of select="@*"/>
   </xsl:copy>
   <xsl:copy>
       <xsl:copy-of select="@* except @xmi:id, @target"/>
       <xsl:attribute name="xmi:id">
           <xsl:value-of select="$RandomNodeID" />
       </xsl:attribute>
       <xsl:attribute name="target">
           <xsl:value-of select="$RandomNodeID" />
       </xsl:attribute>
   </xsl:copy>
 </xsl:template>
<!--    End Extra Branch Mutation    -->
```

9. Missing Branch Operator Template

```
<!--     Missing Branch Mutation     -->
<xsl:template name="MissingBranchMutation">
  <xsl:for-each select="$BranchEdges">
      <xsl:variable name="MutantModel">
      <xsl:variable name="branchID" select="./@xmi:idref" />
      <xsl:apply-templates select="$model" mode="MBO">
          <xsl:with-param name="NodeID" select="$ModelEdges[@xmi:id=$branchID]/@source" />
          <xsl:with-param name="EdgeID" select="./@xmi:idref" />
      </xsl:apply-templates>
  </xsl:variable>
  <xsl:result-document  method="xml"  href="MBO{position()}-AD-Mutant.xml">
      <xsl:sequence select="$MutantModel" />
  </xsl:result-document>
  </xsl:for-each>
</xsl:template>
<xsl:template match="*" mode="MBO">
  <xsl:param name="NodeID" />
  <xsl:param name="EdgeID" />
  <xsl:choose>
      <xsl:when test="name()='edge' and (./@xmi:id=$EdgeID)">
          <xsl:call-template name="update-edges-for-MBO" />
      </xsl:when>
      <xsl:when test="name()='outgoing' and (./@xmi:idref=$EdgeID)" />
      <xsl:when test="name()='source' and ./@xmi:idref=$NodeID and (../@xmi:idref=$EdgeID)">
          <xsl:call-template name="update-source-for-MBO" />
      </xsl:when>
      <xsl:otherwise>
      <xsl:copy>
          <xsl:copy-of select="@*"/>
          <xsl:apply-templates mode="MBO">
              <xsl:with-param name="NodeID" select="$NodeID" />
              <xsl:with-param name="EdgeID" select="$EdgeID" />
          </xsl:apply-templates>
      </xsl:copy>
      </xsl:otherwise>
  </xsl:choose>
</xsl:template>
<xsl:template name="update-edges-for-MBO">
  <xsl:copy>
      <xsl:copy-of select="@* except @source"/>
      <xsl:attribute name="source">
          <xsl:value-of select="@target" />
      </xsl:attribute>
  </xsl:copy>
</xsl:template>
```

```
<!-- template to see changes in EA -->
<xsl:template name="update-source-for-MBO">
  <xsl:copy>
    <xsl:attribute name="xmi:idref">
      <xsl:value-of select="following-sibling::*[1]/@xmi:idref" />
    </xsl:attribute>
    <xsl:apply-templates mode="MBO" />
  </xsl:copy>
</xsl:template>
<!--   End Missing Branch Mutation   -->
```

10. Missing Merge Operator Template

```
<!--     Missing Merge Mutation     -->
<xsl:template name="MissingMergeMutation">
  <xsl:for-each select="$ModelMerges">
  <xsl:variable name="MutantModel">
    <xsl:apply-templates select="$model" mode="MMO">
      <xsl:with-param name="NodeID" select="./@xmi:id" />
    </xsl:apply-templates>
  </xsl:variable>
  <xsl:result-document  method="xml"  href="MMO{position()}-AD-Mutant.xml">
    <xsl:sequence select="$MutantModel" />
  </xsl:result-document>
  </xsl:for-each>
</xsl:template>
<xsl:template match="*" mode="MMO">
<xsl:param name="NodeID" />
  <xsl:choose>
    <xsl:when test="name()='edge' and ./@target=$NodeID">
      <xsl:call-template name="update-edges-for-MMO">
        <xsl:with-param name="NodeID" select="$NodeID" />
      </xsl:call-template>
    </xsl:when>
    <xsl:when test="name()='target' and ./@xmi:idref=$NodeID">
      <xsl:call-template name="update-target-for-MMO">
        <xsl:with-param name="NodeID" select="$NodeID" />
      </xsl:call-template>
    </xsl:when>
    <xsl:otherwise>
    <xsl:copy>
      <xsl:copy-of select="@*"/>
      <xsl:apply-templates mode="MMO">
        <xsl:with-param name="NodeID" select="$NodeID" />
      </xsl:apply-templates>
    </xsl:copy>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
<xsl:template name="update-edges-for-MMO">
  <xsl:param name="NodeID" />
  <xsl:variable name="EdgesWithNodeAsSource" select="$ModelEdges[@source=$NodeID]" />
  <xsl:if test="./@target=$NodeID">
    <xsl:copy>
      <xsl:copy-of select="@* except @target"/>
      <xsl:attribute name="target">
        <xsl:value-of select="$EdgesWithNodeAsSource[1]/@target" />
      </xsl:attribute>
    </xsl:copy>
  </xsl:if>
</xsl:template>
```

```
<!-- template to see changes in EA -->
<xsl:template name="update-target-for-MMO">
  <xsl:param name="NodeID" />
  <xsl:variable name="EdgesWithNodeAsSource" select="$ModelEdges[@source=$NodeID]" />
  <xsl:if test="./@xmi:idref=$NodeID">
      <xsl:copy>
          <xsl:copy-of select="@* except @xmi:idref"/>
          <xsl:attribute name="xmi:idref">
              <xsl:value-of select="$EdgesWithNodeAsSource[1]/@target" />
          </xsl:attribute>
          <xsl:apply-templates mode="MMO" />
      </xsl:copy>
  </xsl:if>
</xsl:template>
<!--   End Missing Merge Mutation   -->
```

11. Negation of Condition Operator Template

```
<!--     Negation of Condition Mutation     -->
<xsl:template name="NegationOfConditionMutation">
  <xsl:for-each select="$GuardConditions">
  <xsl:variable name="MutantModel">
      <xsl:apply-templates select="$model" mode="NCO">
          <xsl:with-param name="GuardID" select="./@xmi:id" />
      </xsl:apply-templates>
  </xsl:variable>
  <xsl:result-document  method="xml"  href="NCO{position()}-AD-Mutant.xml">
      <xsl:sequence select="$MutantModel" />
  </xsl:result-document>
  </xsl:for-each>
</xsl:template>
<xsl:template match="*" mode="NCO">
<xsl:param name="GuardID" />
  <xsl:choose>
      <xsl:when test="name()='guard' and @xmi:id=$GuardID">
          <xsl:call-template name="update-guard-for-NCO" />
      </xsl:when>
      <xsl:otherwise>
      <xsl:copy>
          <xsl:copy-of select="@*"/>
          <xsl:apply-templates mode="NCO">
              <xsl:with-param name="GuardID" select="$GuardID" />
          </xsl:apply-templates>
      </xsl:copy>
      </xsl:otherwise>
  </xsl:choose>
</xsl:template>
<xsl:template name="update-guard-for-NCO">
  <xsl:copy>
      <xsl:copy-of select="@* except @body"/>
      <xsl:attribute name="body">
          <xsl:value-of select="concat('!(',@body,')')" />
      </xsl:attribute>
  </xsl:copy>
</xsl:template>
<!--   End Negation of Condition Mutation   -->
```

2. Coverage analysis of ATM test suite

The following table presents the coverage analysis (Coverage (Cov) % and Cumulative Coverage (CC) %) of a test suite generated for ATM model using the Branch coverage, Interleaving Node (IN) coverage and Interleaving Edge (IE) coverage criteria.

Table B-1: Coverage analysis of ATM test suite

	Branch Coverage		IN Coverage		IE Coverage	
Tests	**Cov %**	**CC %**	**Cov %**	**CC %**	**Cov %**	**CC %**
TS-1	3.57	3.57	2.65	2.65	1.77	1.77
TS-2	10.71	14.29	4.42	5.31	3.98	3.98
TS-3	3.57	14.29	2.65	6.19	1.77	3.98
TS-4	3.57	14.29	2.65	6.19	1.77	3.98
TS-5	17.86	28.57	7.52	9.29	6.19	6.64
TS-6	10.71	32.14	4.87	9.29	3.98	6.64
TS-7	25.00	53.57	15.49	19.47	12.39	15.49
TS-8	21.43	57.14	15.49	21.68	12.83	17.26
TS-9	3.57	57.14	2.65	21.68	1.77	17.26
TS-10	17.86	57.14	13.27	21.68	11.06	17.26
TS-11	3.57	57.14	2.65	21.68	1.77	17.26
TS-12	3.57	57.14	2.65	21.68	1.77	17.26
TS-13	14.29	57.14	5.75	21.68	3.98	17.26
TS-14	3.57	57.14	2.65	21.68	1.77	17.26
TS-15	10.71	57.14	5.75	21.68	3.98	17.26
TS-16	25.00	60.71	8.85	21.68	6.64	17.26
TS-17	7.14	60.71	2.65	21.68	1.77	17.26
TS-18	3.57	60.71	3.98	21.68	2.21	17.26
TS-19	3.57	60.71	2.65	21.68	1.77	17.26
TS-20	3.57	60.71	2.65	21.68	1.77	17.26
TS-21	3.57	60.71	2.65	21.68	1.77	17.26
TS-22	3.57	60.71	3.98	21.68	2.21	17.26
TS-23	28.57	60.71	9.29	21.68	7.08	17.26
TS-24	10.71	60.71	5.31	21.68	3.54	17.26
TS-25	17.86	67.86	10.18	23.01	7.96	18.14
TS-26	10.71	67.86	5.31	23.01	3.54	18.14
TS-27	17.86	67.86	8.85	23.01	6.64	18.14
TS-28	32.14	67.86	10.62	23.01	7.96	18.14
TS-29	3.57	67.86	2.65	23.01	1.77	18.14
TS-30	3.57	67.86	2.65	23.01	1.77	18.14
TS-31	3.57	67.86	2.65	23.01	1.77	18.14
TS-32	14.29	67.86	9.73	23.01	7.52	18.14
TS-33	21.43	67.86	14.60	23.01	11.95	18.14
TS-34	17.86	67.86	9.73	23.01	7.52	18.14
TS-35	25.00	67.86	9.73	23.01	7.52	18.14
TS-36	25.00	67.86	9.29	23.01	7.08	18.14
TS-37	14.29	67.86	9.73	23.01	7.52	18.14
TS-38	14.29	67.86	5.75	23.01	3.98	18.14
TS-39	21.43	67.86	14.16	23.01	11.95	18.14
TS-40	7.14	67.86	2.65	23.01	1.77	18.14
TS-41	10.71	67.86	5.75	23.01	3.98	18.14
TS-42	17.86	67.86	8.85	23.01	6.64	18.14
TS-43	14.29	67.86	9.73	23.01	7.52	18.14
TS-44	3.57	67.86	2.65	23.01	1.77	18.14
TS-45	3.57	67.86	2.65	23.01	1.77	18.14

TS-46	3.57	67.86	2.65	23.01	1.77	18.14
TS-47	3.57	67.86	2.65	23.01	1.77	18.14
TS-48	3.57	67.86	2.65	23.01	1.77	18.14
TS-49	14.29	67.86	9.73	23.01	7.52	18.14
TS-50	25.00	67.86	10.62	23.01	7.96	18.14
TS-51	14.29	67.86	9.73	23.01	7.52	18.14
TS-52	3.57	67.86	2.65	23.01	1.77	18.14
TS-53	3.57	67.86	2.65	23.01	1.77	18.14
TS-54	3.57	67.86	2.65	23.01	1.77	18.14
TS-55	14.29	67.86	8.85	23.01	6.64	18.14
TS-56	3.57	67.86	2.65	23.01	1.77	18.14
TS-57	3.57	67.86	2.65	23.01	1.77	18.14
TS-58	3.57	67.86	2.65	23.01	1.77	18.14
TS-59	3.57	67.86	2.65	23.01	1.77	18.14
TS-60	14.29	67.86	9.73	23.01	7.52	18.14
TS-61	10.71	67.86	5.75	23.01	3.98	18.14
TS-62	17.86	67.86	5.75	23.01	3.98	18.14
TS-63	25.00	67.86	13.27	23.01	11.06	18.14
TS-64	3.57	67.86	2.65	23.01	1.77	18.14
TS-65	7.14	67.86	2.65	23.01	1.77	18.14
TS-66	25.00	67.86	10.62	23.01	7.96	18.14
TS-67	10.71	67.86	5.75	23.01	3.98	18.14
TS-68	17.86	67.86	8.41	23.01	6.64	18.14
TS-69	3.57	67.86	2.65	23.01	1.77	18.14
TS-70	14.29	67.86	9.29	23.01	7.52	18.14
TS-71	10.71	67.86	6.19	23.01	3.98	18.14
TS-72	17.86	67.86	8.85	23.01	6.64	18.14
TS-73	10.71	67.86	5.75	23.01	3.98	18.14
TS-74	10.71	67.86	5.75	23.01	3.98	18.14
TS-75	3.57	67.86	2.65	23.01	1.77	18.14
TS-76	21.43	67.86	14.16	23.01	11.95	18.14
TS-77	3.57	67.86	2.65	23.01	1.77	18.14
TS-78	3.57	67.86	2.65	23.01	1.77	18.14
TS-79	3.57	67.86	2.65	23.01	1.77	18.14
TS-80	3.57	67.86	2.65	23.01	1.77	18.14
TS-81	7.14	67.86	2.65	23.01	1.77	18.14
TS-82	3.57	67.86	2.65	23.01	1.77	18.14
TS-83	3.57	67.86	2.65	23.01	1.77	18.14
TS-84	3.57	67.86	2.65	23.01	1.77	18.14
TS-85	3.57	67.86	2.65	23.01	1.77	18.14
TS-86	3.57	67.86	2.65	23.01	1.77	18.14
TS-87	25.00	89.29	17.26	34.07	14.60	29.20
TS-88	3.57	89.29	2.65	34.07	1.77	29.20
TS-89	28.57	100.00	22.12	37.61	19.91	34.51