

MODEL-BASED TESTING FOR SOFTWARE PRODUCT LINES


by

Erika Mir Olimpiew
A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
in Partial Fulfillment of
The Requirements for the Degree
of
Doctor of Philosophy
Information Technology



Committee:



Dr. Hassan Gomaa, Dissertation
Director



Dr. Don Gantz, Committee Member

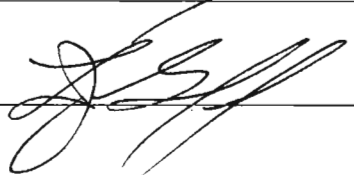



Dr. Jeff Offutt, Committee Member

Dr. David Rine, Committee Member



Dr. Daniel Menascé, Associate Dean for
Research and Graduate Studies



Dr. Lloyd J. Griffiths, Dean, The Volgenau
School of Information Technology and
Engineering

Date: April 30th, 2008

Spring Semester 2008
George Mason University
Fairfax, Virginia

Model-Based Testing for Software Product Lines

A dissertation submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy at George Mason University

By

Erika Mir Olimpiew

Master of Science, Virginia Commonwealth University, 1997
Bachelor of Science, Virginia Commonwealth University, 1995

Director: Dr. Hassan Gomaa, Chairman and Professor
Department of Information and Software Engineering
Volgenau School of Information Technology & Engineering

Spring 2008
George Mason University
Fairfax, VA

Copyright 2008
Erika Mir Olimpiew
All Rights Reserved

DEDICATION

This dissertation is dedicated to my father Alexandre Olimpiew, my mother Dorothea Renata Mir Olimpiew, my siblings Igor, Astrid, Christian, Monika, Alex, and in the memory of my youngest brother Andrew. I would also like to dedicate this dissertation in the memory of my grandmother Zylnah C.L. Olimpiew.

ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. Hassan Gomaa for his patient, thorough and valuable help for many years, from the development of the idea for this dissertation to its conclusion. I would also like to thank Dr. Gantz, Dr. Jeff Offutt, Dr. David Rine, and Dr. Jon Whittle for serving in my committee.

Also, I would like to thank all participants of all applied research projects from the SWE 796 Directed Reading, and SWE 721 / IT 821 Reusable Software Architecture classes: Jung-Woo Peter An, Dave Anderson, Lima Beauvais, Dwight Donaldson, Ahmed Elkhodary, Hatim Hussein, Hugo Kang, Frederic Kneisel, Chris Magrin, Jason Pepper, Rasheed Rabbi, and Rich Thornett. Thanks are also due to Diana Webber who developed the initial version of the Banking System SPL and to Vinesh Vonteru who implemented the version that was tested as described in this dissertation.

Last, but not least, I would like to thank all my friends at GMU that participated in the Informal Ph.D. sessions and the numerous discussions about this research. I would like to thank Mazen Saleh for helping to start the Informal Ph.D. sessions, for his help explaining the separation of concerns method, and for the use and support of a tool he developed to implement the method.

TABLE OF CONTENTS

	Page
LIST OF TABLES	xi
LIST OF FIGURES	xiv
ABSTRACT	xvii
1 Introduction	1
1.1 Motivation and Scope of Research	2
2 Problem Statement and Research Approach	5
2.1 Problem Statement	5
2.2 Thesis Statement	5
2.3 Overview of Approach	5
3 Related Research	9
3.1 Introduction	9
3.2 Software Models and Modeling Methods	9
3.3 Software Product Lines	11
3.4 Variability Mechanisms	16
3.5 Separation of Concerns	18
3.6 Software Testing of Single Systems	21
3.6.1 White Box Testing	22
3.6.2 Black Box or Requirements-based Testing	23
3.6.3 Regression Testing	25
3.6.4 Model-Based Testing	25
3.6.5 Test Management	27
3.7 Software Testing of Software Product Lines	28
3.7.1 A Testing Process for Software Product Lines	28
3.7.2 Systematic Reuse of Use Case-Based Tests in a Software Product Line	29

3.7.3	Variability Management with Separation of Concerns.....	33
3.7.4	Selecting Representative Applications to Test	33
3.8	Comparison and Analysis of Related Research on Software Testing SPLs	35
4	Extending Model-Based Testing for Software Product Lines	40
4.1	Incorporating CADeT and CADeT-SoC within a SPL Development Process.....	40
4.2	Model-Based Testing for a Single Application	43
4.3	Model-Based Testing for a SPL.....	45
5	CADeT: A Model-Based Testing Method for SPLs	48
5.1	Developing Customizable Test Specifications During SPL Engineering.....	48
5.2	Phase I: Creating Activity Diagrams from Use Cases During SPL Engineering	50
5.2.1	Role Stereotypes in CADeT.....	51
5.2.2	Reuse Stereotypes and Feature Conditions in CADeT	52
5.2.3	Creating Activity Diagrams from Use Cases.....	55
5.2.4	Analyzing the Impact of Features on the Activity Diagrams	56
5.2.5	Example of Creating Activity Diagrams from Use Cases	58
5.3	Phase II: Creating Decision Tables and Test Specifications from Activity Diagrams During SPL Engineering.....	71
5.3.1	Example of Creating Decision Tables from Activity Diagrams	75
5.4	Phase III: Defining Feature-Based Test Plan.....	77
5.4.1	Analysis of Feature Model.....	78
5.4.2	Analysis of Relationships between Features and Test Specifications	80
5.4.3	Applying a Feature-Based Coverage Criterion.....	81
5.4.4	Example of Defining a Feature-Based Test Plan.....	82
5.5	Phase IV: Applying the Parameterization Variability Mechanism to Decision Tables and Test Specifications During SPL Engineering	85
5.5.1	Tool Support for Parameterization Variability Mechanism.....	86
5.5.2	Binding Times Supported by CADeT Tools	87
5.5.3	Description of Approach.....	88
5.5.4	Example of Applying Parameterization Mechanism	89
5.6	Customizing Test Specifications During Application Engineering.....	92

5.7	Phase V: Customizing the Decision Tables and Test Specifications Using the Parameterization Variability Mechanism	93
5.7.1	Selecting Values of Feature Conditions.....	94
5.7.2	Example of Selecting Values of Feature Conditions	94
5.7.3	Applying Test Specification Generator Tool.....	97
5.7.4	Example of Applying Test Specification Generator Tool	97
5.7.5	Applying Test Procedure Definition Tool	98
5.7.6	Example of Applying Test Procedure Definition Tool.....	102
5.7.7	Applying System Test Generator Tool	104
5.7.8	Example of Applying System Test Generator Tool.....	105
5.8	Phase VI: Selecting Input Data	107
5.8.1	Creating Database Structure from Static Entity Model of Application	107
5.8.2	Selecting Input Data to Satisfy Database Constraints.....	108
5.8.3	Example of Selecting Input Data for Database.....	109
5.8.4	Selecting Input Data to Satisfy Execution Conditions in System Tests	111
5.8.5	Example of Selecting Input Data for System Tests	113
5.9	Phase VII: Testing Application.....	115
5.10	Summary	116
6	CADeT-SoC: Extending CADeT with Separation of Concerns	118
6.1	Separation of Concerns Variability Mechanism in CADeT-SoC	120
6.2	Phase IV _{SoC} : Applying Separation of Concerns to Test Specifications During SPL Engineering.....	121
6.2.1	Extending SCAC for SCT.....	121
6.2.2	Applying SCT to the Test Specifications of a SPL.....	125
6.2.3	Example of Applying SCT to the Test Specifications of a SPL	127
6.2.4	Phase V _{SoC} : Applying Feature-Based Test Derivation using Separation of Concerns.....	130
6.2.5	Example of Applying Feature-Based Test Derivation using SCT.....	132
6.3	Comparison of CADeT and CADeT-SoC	134
6.4	Summary	135
7	Evaluation of CADeT and CADeT-SoC.....	137
7.1	Rationale for Selecting Case Study Research Method	138

7.2	Description of Evaluation	139
7.3	Preliminary Study to Evaluate Feasibility of Initial Version of CADeT	140
7.3.1	Description of Study	140
7.3.2	Results.....	142
7.3.3	Interpretation of Results.....	144
7.4	Evaluate Feasibility of Creating and Customizing Test Specifications Using CADeT (Phases I-V)	145
7.4.1	Description of Study	146
7.4.2	Characteristics of Requirement Models.....	146
7.5	Application of Phases I-V: Creating and Customizing Test Specifications for the AHTS SPL	147
7.5.1	Coverage of All Use Case Scenarios and All Features in AHTS SPL	147
7.5.2	Coverage of All Relevant Feature Combinations in AHTS SPL.....	152
7.5.3	Coverage of All Use Case Scenarios in each Application of the AHTS SPL	156
7.6	Application of Phases I-V: Creating and Customizing Test Specifications for the Banking System SPL	159
7.6.1	Coverage of All Use Case Scenarios and All Features in Banking System SPL	159
7.6.2	Coverage of All Relevant Feature Combinations in Banking System SPL	164
7.6.3	Coverage of All Use Case Scenarios in each Application of the Banking System SPL	166
7.6.4	Number of Applications Configured for each SPL.....	167
7.6.5	Comparison of Number of Test Specifications Created using CADeT with alternative approaches.....	167
7.6.6	Number of Test Specifications Created for AHTS SPL	169
7.6.7	Number of Test Specifications Created for Banking System SPL	170
7.7	Evaluate Feasibility and Effort of Customizing Test Specifications and Testing Applications Using CADeT and CADeT-SoC	172
7.7.1	Description of Study	173
7.7.2	Results of Applying Each Phase.....	180
7.8	Creating and Customizing Test Specifications Using Parameterization	183
7.8.1	Results for Phase IV: Apply the Parameterization Variability Mechanism	183

7.8.2	Results for Phase V: Customize Test Specifications for Two Applications using the Parameterization Variability Mechanism	185
7.9	Selecting Test Data for Customized Test Specifications.....	187
7.9.1	Results for Phase VI: Select Test Data for Two Applications.....	187
7.10	Results for Phase VII: Test Two Applications	190
7.10.1	Results of Executing the Tests.....	190
7.10.2	Coverage of All Use Case Scenarios and All Features.....	194
7.10.3	Faults Discovered.....	198
7.11	Applying Separation of Concerns.....	201
7.11.1	Results for Phase IV _{SoC} : Learn and Apply Separation of Concerns Variability Mechanism to Test Specifications	201
7.11.2	Results for Phase V _{SoC} : Customize Test Specifications for Two Applications using the Separation of Concerns Variability Mechanism	203
7.12	Applying a Pragmatic Approach.....	205
7.12.1	Results of Applying Pragmatic Approach to Create Test Specifications for Two Applications	205
7.13	Results of Questionnaire.....	206
7.14	Interpretation of 3 rd Study Results	207
7.15	Comparison of CADeT with Previous Research on SPL Testing Methods ...	208
8	Conclusions.....	210
8.1	Contributions.....	211
8.1.1	Application of a Feature-Based Coverage Criterion with a Use Case-Based Coverage Criterion	212
8.1.2	Distinguishing Between Coarse-Grained and Fine-Grained Functional Variability	212
8.1.3	Using Separation of Concerns to Customize the Test Specifications of a SPL	213
8.1.4	Prototype Tools to Customize SPL Test Specifications	213
8.1.5	An Evaluation of CADeT and CADeT-SoC on Two SPLs.....	214
8.2	Further Study	214
8.2.1	Determining a Break-Even Point.....	214
8.2.2	Automating More Phases of CADeT and CADeT-SoC	215

8.2.3	Incorporating Feature-Based and Use Case Scenario-Based Coverage Criteria with Unit and Integration Testing Criteria.....	215
8.2.4	Evaluating the Impact of SPL Evolution	216
8.2.5	Resolving Inconsistencies between Requirement Models.....	216
8.2.6	Incremental Testing of SPL	216
8.2.7	Integrating Additional Variability Mechanisms	217
8.2.8	Detecting Feature-Based Faults	217
8.2.9	Evaluating CAdET and CAdET-SoC on Industrial SPLs	217
Appendix A: Banking System SPL case study		218
A.1	Requirement Models for Banking System SPL.....	218
A.2	Example of Phase I: Create Activity Diagrams during SPL Engineering.....	222
A.3	Example of Phase II: Create Decision Tables and Test Specifications from Activity Diagrams.....	229
A.4	Example of Phase III: Define Feature-Based Test Plan	232
A.5	Example of Phase IV: Apply Parameterization Variability Mechanism	237
A.6	Example of Phase V: Customize Decision Tables and Test Specifications using Parameterization Variability Mechanism	241
A.7	Example of Phase VI: Select Test Data.....	248
A.8	Example of Phase VII: Test Application	250
A.9	Example of Applying Separation of Concerns Variability Mechanism in CAdET-SoC	253
Appendix B: Glossary		258
REFERENCES		266

LIST OF TABLES

Table	Page
Table 1 Comparison of SPL testing methods	39
Table 2 Feature condition selection values	55
Table 3 Feature list for AHTS SPL.....	66
Table 4 Excerpt of feature to use case relationship table for the AHTS SPL.....	67
Table 5 Structure of decision table	73
Table 6 Decision table for "Enter through transponder enabled booth" use case.....	76
Table 7 Number of possible feature selections for feature conditions.....	79
Table 8 Excerpt of feature / test specification relationship table for AHTS SPL.....	83
Table 9 A feature-based combinatorial test plan for the AHTS SPL	85
Table 10 Example of parameterization mechanism applied to decision table.....	91
Table 11 Feature selections for TS1	95
Table 12 Example of a customized decision table for TS1	96
Table 13 Example of a test specification generated for TS1	97
Table 14 Example of system tests from test procedure document of TS1.....	104
Table 15 Excerpt of system tests document for TS1	106
Table 16 Relationship between static model notation and database structure.....	108
Table 17 Example of input data selected for database of TS1.....	111
Table 18 Conventions for representing the selection of variable values	112
Table 19 Example of input data selected for a system test.....	114
Table 20 Example of pass / fail status in a test specification.....	117
Table 21 Feature description language	123
Table 22 Relationship between insertion points, test specifications and variable test step file	124
Table 23 Representing interacting features in variable file	125

Table 24 Number of variable test steps defined for variation points in AHTS SPL decision tables	134
Table 25 Studies used to evaluate CADeT and CADeT-SoC.....	140
Table 26 Requirements models of AHTS SPL	141
Table 27 Assessment of initial version of CADeT	142
Table 28 Results of questionnaire for first applied project.....	144
Table 29 Characteristics of requirement models created for each SPL	147
Table 30 Features / test specifications relationships in AHTS SPL	150
Table 31 Relevant feature combinations in AHTS SPL	153
Table 32 Test specifications selected for the applications in the AHTS SPL test plan ..	155
Table 33 Test procedure for TS2	158
Table 34 Features / test specifications relationships in Banking System SPL	163
Table 35 Relevant feature combinations in the Banking System SPL	164
Table 36 Number of application configurations for each SPL	167
Table 37 Number of test specifications created for AHTS SPL.....	169
Table 38 Number of test specifications created for Banking System SPL	171
Table 39 Time log template	176
Table 40 Total time in man-hours spent learning and applying each phase.....	182
Table 41 Time in man-hours to learn and apply Phase IV	184
Table 42 Time in man-hours to learn and apply Phase V	186
Table 43 Time in man-hours to learn and apply Phase VI	189
Table 44 Time in man-hours to learn and apply Phase VII	191
Table 45 Test results assigned by participants for test cases	192
Table 46 Corrected test results.....	193
Table 47 Features associated with applications of the Banking System SPL.....	195
Table 48 Number of test cases executed against each application	196
Table 49 Faults found in the applications of the Banking System SPL.....	199
Table 50 Time in man-hours to learn and apply Phase IV-SoC	202
Table 51 Time in man-hours to learn and apply Phase V-SoC.....	204
Table 52 Time in man-hours to learn and apply pragmatic approach	206
Table 53 Results of Questionnaire.....	207

Table 54 Feature list for Banking System SPL.....	224
Table 55 Feature to use case relationship table for Banking System SPL.....	225
Table 56 Excerpt of decision table for “Validate pin” use case	230
Table 57 Excerpt of feature / test specification relationships in Banking System SPL..	235
Table 58 Pair-wise coverage criterion applied to Banking System SPL	237
Table 59 Excerpt from modified “Validate pin” decision table	240
Table 60 Feature selections for application TS1 from the Banking System SPL.....	241
Table 61 Excerpt of customized decision table for “Validate pin” use case	243
Table 62 Card is valid test specification	244
Table 63 Example of system test sequences for TS1	246
Table 64 Excerpt from “System test 1” of TS1	246
Table 65 Example of input data selected for database of TS1	249
Table 66 Test results for “Card Is Valid” test case in system test 1	252
Table 67 Insertion points in “Validate pin” decision table	254

LIST OF FIGURES

Figure	Page
Figure 1 SPL development processes used with PLUS	15
Figure 2 Incorporating CADeT within the SPL development process of PLUS	42
Figure 3 Meta-model describing model-based testing for a single application	45
Figure 4 Feature-oriented model-based testing for a SPL	47
Figure 5 SPL test development activities.....	50
Figure 6 Example of role stereotypes	52
Figure 7 Example of reuse stereotypes	53
Figure 8 Feature model for AHTS SPL	59
Figure 9 “Enter toll road” use case and extension use cases	60
Figure 10 “Enter toll road” use case description	61
Figure 11 “Enter through transponder-enabled booth” use case description.....	62
Figure 12 “Enter through ticket-issuing booth” use case description.....	63
Figure 13 Initial system level diagram for AHTS SPL.....	64
Figure 14 Modified "Enter toll road" use case activity diagram.....	69
Figure 15 Activity diagram referenced by “Enter through toll booth” activity node	70
Figure 16 Sub-activity diagrams for adaptable activity nodes.....	71
Figure 17 Example of a simple path trace	77
Figure 18 Excerpt of AHTS model with implicit feature dependency	84
Figure 19 Association between features and test specifications	87
Figure 20 Incorporating CADeT within an application engineering process	93
Figure 21 Graph building algorithm	100
Figure 22 System test definition tool	101
Figure 23 Excerpt of test order graph for TS1	103
Figure 24 Excerpt of static model for AHTS SPL	110
Figure 25 Association of the alarm feature with a variable test step	122

Figure 26 Application of SCT during SPL engineering	126
Figure 27 Example of test insertion points and variable test step file	128
Figure 28 Excerpt of variable feature file for AHTS SPL	129
Figure 29 Application of SCT during feature-based test derivation.....	131
Figure 30 Code weaver tab in SPLET tool	132
Figure 31 Test specification for Invalid Transponder.....	133
Figure 32 Example of customized test specification	134
Figure 33 Relationship between use case scenarios and test specifications of AHTS SPL	149
Figure 34 Test execution sequence graph for TS2.....	157
Figure 35 Execution sequence for system test 1	159
Figure 36 Relationship between use case scenarios and test specifications of Banking System SPL	161
Figure 37 Total time in man-hours spent learning and applying each phase.....	181
Figure 38 Time in man-hours to learn and apply Phase IV	184
Figure 39 Time in man-hours to learn and apply Phase V	186
Figure 40 Time in man-hours to learn and apply Phase VI	188
Figure 41 Time in man-hours to learn and apply Phase VII.....	191
Figure 42 Time in man-hours to learn and apply Phase IV-SoC	202
Figure 43 Time in man-hours to learn and apply Phase V-SoC	204
Figure 44 Time in man-hours to learn and apply pragmatic approach	206
Figure 45 Feature model for Banking System SPL	219
Figure 46 Use case model for Banking System SPL	220
Figure 47 Validate pin use case description	221
Figure 48 System level activity diagram for Banking System SPL.....	223
Figure 49 Activity diagram for “Validate pin” use case.....	227
Figure 50 Sub-activity diagrams for adaptable nodes in “Validate pin” activity diagram	228
Figure 51 “Display welcome message” adaptable node	235
Figure 52 Dependencies between test specifications of TS1	246
Figure 53 Debit card class.....	248
Figure 54 Static model for Banking System SPL	248

Figure 55 ATM user interface for TS1	251
Figure 56 Excerpt of variable test step file for Banking System SPL	255
Figure 57 “Card is valid” test specification	257
Figure 58 “Card is valid” test specification customized for TS1.....	257

ABSTRACT

MODEL-BASED TESTING FOR SOFTWARE PRODUCT LINES

Erika Mir Olimpiew, Ph.D.

George Mason University, 2008

Dissertation Director: Dr. Hassan Gomaa, Chairman & Professor

A Software Product Line (SPL), or family of systems, is a collection of applications that have so many features in common that it is worthwhile to study and analyze the common features as well as analyzing the features that differentiate these applications. Model-based design and development for SPLs extends modeling concepts for single applications to model the commonality and variability among the members of the SPL.

Previous research on model-based functional testing methods for SPLs use existing requirement models, such as feature and use case models, to create reusable test specifications that can be configured for applications derived from a SPL. Feature-based test coverage criteria can be applied to determine what applications to test, when it is not feasible to test all possible applications of a SPL. However, previous research on functional testing methods for SPLs does not apply feature-based test coverage criteria together with a use case-based approach of creating reusable test specifications for a SPL.

This research describes a functional test design method for SPLs (Customizable Activity diagrams, Decision tables and Test specifications, or CAdET) that applies feature-based test coverage criteria together with a use case-based approach of creating reusable test specifications for a SPL. Features from a feature model are associated with test models created from the use cases of a SPL using feature condition variables. The values of a feature condition represent possible feature selections, so that selecting a value for the feature condition selects and customizes the test models associated with that feature.

With CAdET, activity diagrams are created from the use case descriptions of a SPL. Reusable test specifications are traced from the use case activity diagrams and described in decision tables. The relationships of features to activity diagrams are also portrayed in decision tables, and then analyzed to apply a feature-based test coverage criterion to the SPL. Representative applications configurations are generated to cover all features, all use case scenarios, and all relevant feature combinations of a SPL. Reusable test specifications are selected and customized for each application configuration, and then used to test the corresponding application implementation.

Furthermore, CAdET is extended to use separation of concerns to customize the reusable test specifications during feature-based test derivation (CAdET-SoC). Instead of using feature conditions to customize these test specifications, CAdET-SoC separates the variable test steps from the test specifications, and then weaves selected test steps with these test specifications during feature-based test derivation. CAdET-SoC is more suitable than CAdET for customizing the test specifications of a SPL with many variation

points repeated across several use cases. Using CAdET-SoC reduced the effort needed to define variable test steps for the variation points in test specifications in each SPL.

The feasibility of the CAdET and CAdET-SoC methods was evaluated in three studies on two SPLs: an Automated Highway Toll System (AHTS) SPL, and a Banking System SPL. The results of these studies show that CAdET and CAdET-SoC can be used to create reusable test specifications to cover all use case scenarios, all features, and all relevant feature combinations on each of these two SPLs. The feature model of each SPL, and the relationships of features to test specifications were analyzed to determine the relevant feature combinations, and a feature-based coverage criterion was applied to reduce the number of application configurations to test. Using CAdET also reduced the number of test specifications needed to satisfy these criteria, as compared with using two alternative approaches.

The contribution of this research is CAdET, a model-based test specification design method, and CAdET-SoC, an extension of CAdET that uses separation of concerns to customize the test specifications for an application derived from the SPL. CAdET and CAdET-SoC can help a test engineer create reusable test specifications to cover all use case scenarios, features and relevant feature combinations of a SPL. These test specifications can be customized during feature-based test derivation for a set of applications derived from a SPL. Using CAdET and CAdET-SoC reduces the number of application configurations and test specifications that need to be created to cover all use case scenarios, features and relevant feature combinations in a SPL.

1 Introduction

Software applications are developed to fulfill the needs of different users in various business domains, such as the customers and operators of a banking system. Over time, a business may develop and deploy several similar applications and configure each application in a different environment, with variations in language, business rules, operating system, and hardware features. A *Software Product Line (SPL)*, or family of systems, is a collection of applications that have so many features in common that it is worthwhile to study and analyze the common features as well as analyzing the features that differentiate these applications, in order to more effectively reuse software assets across members of the family (Parnas 1978; Clements and Northrop 2002). A Software Product Line (SPL) development method proactively designs a family of applications with similar characteristics, in order to reuse common features across the members of a SPL and also to distinguish between the features, or requirements, that differentiate these applications. Developing a SPL requires more time and resources than developing a single application. Over time, this additional investment is expected to pay off by reducing the time to market and costs of deriving and configuring new applications, which are members of the SPL (Clements and Northrop 2002).

Most research on SPL development methods has investigated the development of requirements, software models and implementation of a SPL (Kang, Kim et al. 1998; Weiss and Lai 1999; Clements and Northrop 2002; Gomaa 2005). Although a few researchers have addressed the management of testing processes and the development of test specifications for a SPL (McGregor 2001), there are still many open problems in this area of research. For instance, it is not clear how a testing process should fit within a SPL development method; how SPL models can be used to create reusable test specifications that can be configured for any application derived from the SPL; and how to analyze these models to define an adequate feature-based test coverage criterion.

Managing the testing processes and developing test specifications for a SPL requires a test design method that is in concert with the SPL development method used to create the requirements and software models of the SPL. If possible, a functional test design method for a SPL should leverage existing models, such as the requirements models, to create black-box system test specifications. Care must be taken to identify which models should be used; how to extend these models to assist in the planning and design of test specifications; how to analyze these models to define an adequate feature-based coverage criterion; and how to apply a mechanism to automate the configuration of these test specifications for an application derived from the SPL.

1.1 Motivation and Scope of Research

Managing features is an essential part of SPL development. The emphasis in feature modeling is in capturing the SPL variability, as given by optional and alternative features, since these features differentiate one member of the family from the others. Use

cases, on the other hand, are a means of describing the functional requirements of an application in terms of informal, narrative descriptions of interactions between the actor (application user) and application (Jacobson, Christerson et al. 1992). Use cases can also serve to describe the functional requirements of a SPL. The goal of the use case analysis is to get a good understanding of the functional requirements whereas the goal of feature analysis is to enable reuse (M. L. Griss, J. Favaro et al. 1998). Using a feature-oriented approach in a SPL-based functional test design method can help a requirements analyst and test engineer to represent, analyze and manage the relationships of features to functional requirements, and the test specifications created from these requirements.

Use case-based test design methods for SPLs address the problem of systematically reusing functional test specifications for the applications derived from an SPL (Bertolino and Gnesi 2003; Nebut, Fleurey et al. 2003; Reuys, Kamsties et al. 2005). These methods identify and automatically configure the variability in the test specifications for an application derived from the SPL, but do not provide a feature-oriented approach to systematically represent and manage the relationships between the features and test specifications of a SPL.

A few requirements-based SPL testing methods address the problem of selecting representative application configurations to test from the configuration space of a SPL, in situations where the set of applications derived from the SPL is not pre-determined and is likely to change (McGregor 2001; Scheidemann 2006). For example, a SPL for a mobile phone may contain several optional features, which can be selected by a prospective customer. These methods provide a feature-oriented approach to select representative

applications from a SPL, but do not extend this approach to reuse and customize the test specifications for these applications.

This research builds on previous research on requirements-based test design methods for a SPL by investigating the problem of representing, analyzing and managing the relationships of common, optional and alternative features to the use cases of a SPL. Managing these relationships enables the development of reusable, functional test specifications that can be configured during feature-based test derivation for an application derived from the SPL. Reusable test specifications reduce the number of test specifications that need to be created for a SPL. Furthermore, managing these relationships reduces the number of application configurations to test, by enabling the application of a feature-based coverage criterion to cover all features, use case scenarios, and relevant feature combinations of a SPL.

A feature-oriented functional test design method is proposed in this research to combine a use case scenario-based test coverage criterion to provide functionality coverage together with a feature-based test coverage criterion to provide variability coverage of a SPL. This method systematically represents, analyzes and manages the relationships of common and variable features to the test specifications of a SPL, so that these test specifications can be customized during feature-based test derivation to test a set of applications derived from the SPL.

2 Problem Statement and Research Approach

2.1 Problem Statement

This dissertation investigates and proposes a solution to the problem of testing applications developed from a SPL in which the functional requirements are expressed as use cases:

There is a need for a test design method to create reusable and functional test specifications to satisfy use case-based and feature-based coverage criteria for a SPL, where these test specifications can be configured during feature-based test derivation to test a set of applications derived from the SPL.

2.2 Thesis Statement

A test design method can be developed to create reusable and functional test specifications to satisfy use case-based and feature-based coverage criteria for a SPL, where these test specifications can be configured during feature-based test derivation to test a set of applications derived from the SPL.

2.3 Overview of Approach

The proposed solution is Creating Customizable Activity Diagrams, Decision Tables, and Test Specifications (CADET). CADeT is a model-based test design method that enables a test engineer to create reusable and configurable test specifications to cover

all use case scenarios and features for a SPL, which can be automatically selected and configured during feature-based test derivation to test a set of applications derived from the SPL.

A test engineer uses CAdET to create customizable activity diagrams, decision tables, and test specifications from the feature and use case models of the Product Line UML based Software engineering (PLUS) method (Gomaa 2005). PLUS is a UML-based design method that uses both feature modeling and use case modeling to describe the requirements of a SPL. CAdET can also be used with other SPL development methods that use both feature and use case models to describe the SPL requirements.

The CAdET method is divided into four phases during SPL engineering, and three phases during application engineering. These phases are outlined below:

- Phase I: Create activity diagrams from use cases
- Phase II: Create decision tables and test specifications from activity diagrams
- Phase III: Create a feature-based test plan
- Phase IV: Apply a variability mechanism to customize decision tables.

The remaining three phases are done during application engineering for each application in the feature-based test plan. During application engineering, phase V is automated, and phases VI and VII include manual activities. These phases are outlined below:

- Phase V: Select and customize test specifications for a given application
- Phase VI: Select test data for the application
- Phase VII: Test application.

CADeT-SoC extends CADeT to use a separation of concerns variability mechanism in phases IV and V of CADeT. CADeT-SoC replaces phases IV and V of CADeT with the following phases.

During SPL engineering:

- Phase IV_{SoC}: Apply a separation of concerns variability mechanism

The remaining phase is done during application engineering for each application:

- Phase V_{SoC}: Select and customize test specifications using separation of concerns for a given application.

CADeT and CADeT-SoC were evaluated on two SPLs: an Automated Highway Toll System SPL and a Banking System SPL. Reusable test specifications were created for each SPL using CADeT and CADeT-SoC. Then, a set of representative application configurations was selected to cover features and selected feature combinations in each SPL. The reusable test specifications were customized for each application configuration in each SPL using CADeT and CADeT-SoC. The remaining phases (VI “Select test data” and VII “Test application”) were applied to the Banking System SPL. A set of applications was derived from a Banking System SPL implementation and then tested using the customized test specifications.

Chapter 3 describes related research in the area of software modeling, software testing and software product lines. Chapter 4 describes how a model-based testing method for a single application is extended to create CADeT and CADeT-SoC. The CADeT method is described in chapter 5, and CADeT-SoC is described in chapter 6.

Chapter 7 describes the evaluation of these methods on the case studies. Chapter 8 contains the conclusions, contributions, and further study.

3 Related Research

3.1 Introduction

This chapter starts with a broad overview of related research in the more general areas of software modeling, software product lines, and software testing, and then ends with a more in-depth review and comparison of related research in the area of software testing SPLs.

3.2 Software Models and Modeling Methods

A model of a software system is an abstraction of the system from a particular viewpoint, described with a graphical or textual notation (Rumbaugh, Jacobson et al. 2005). A software modeling method describes how to develop software models using a modeling notation. Structured analysis and design methods emphasize the functions and data flow aspects of a system (Yourdon 1989), while object-oriented methods (Rumbaugh 1991; Jacobson, Christerson et al. 1992; Jacobson, Martin Griss et al. 1997) emphasize the encapsulation, grouping and categorization of objects in a system. Bouzeghoub et al survey some well known object-oriented modeling methods (Bouzeghoub, Gardarin et al. 1997).

Many modeling approaches use several modeling views of a software system, referred to as multiple-view modeling. Multiple-view modeling includes a context

modeling view to describe the interface of the software system to external devices, actors and systems, a functional modeling view to describe the functions of the system, a static modeling view to provide a structural perspective of the system, and a dynamic modeling view to provide a behavioral perspective of the system. The Concurrent Object Modeling and architectural design mEThod (COMET) method (Gomaa 2000) is an UML-based object-oriented software development method that uses multiple-view modeling. A software engineer uses COMET to develop a use case model that describes the functional software requirements. In the analysis modeling phase the software engineer develops a static model, communication diagrams and statecharts from the use cases. Then, in the design modeling phase, the engineer develops the software application architecture from these models.

The use case model, first introduced by Jacobson et al. describes use cases, where a use case groups sequences of interactions that provide a service of value to an outside user (actor) of the system (Jacobson, Christerson et al. 1992). The 4+1 view model of software architecture emphasizes the importance of use case modeling as a driver to determine architectural elements, and as the starting point for testing the system. In the analysis phase, use cases relate to classes in the static model, object interaction diagrams in the dynamic model, and test specifications in the test model (Krutchen 1995).

The Unified Modeling Language (UML) is a graphical modeling language which fully incorporates use case diagrams for modeling requirements, class diagrams for static modeling, sequence and communication diagrams for inter-object dynamic modeling, and state diagrams for intra-object dynamic modeling. A UML profile extends the UML

language to a particular domain. A UML profile defines domain-specific stereotypes, tagged values and constraints on a subset of the UML model elements in terms of the UML meta-model. A UML meta-model describes relationships between modeling elements that are applicable to all domains. A stereotype is a classification mechanism that defines additional semantics for a model element, a tagged value is a set of keyword-value pairs that define additional properties for a model element, and a constraint is a restriction on the semantics or value of a model element. The UML notation is used with a software development method (Rumbaugh, Jacobson et al. 2005; OMG 2007).

In 1997, OMG developed the Object Constraint Language (OCL) to help formalize the UML models. OCL is a declarative, typed language that is free of side effects. The OCL language contains several different types of constraints that help formalize the UML models: invariants, and pre and post conditions. Invariants are expressions that represent rules, or conditions that are true for a set of model element instances in an UML model. Precondition and postcondition constraints are associated with a class operation or other behavioral feature. Precondition constraints specify conditions that must be true before the operation is executed, while postcondition constraints specify conditions that must be true after the operation is executed (Warmer and Kleppe 1999).

3.3 Software Product Lines

A *Software Product Line (SPL)*, or family of systems, is a collection of applications that have so many features in common that it is worthwhile to study and analyze the common features as well as analyzing the features that differentiate these

applications, in order to more effectively reuse software assets across members of the family (Parnas 1978; Clements and Northrop 2002). Several SPL development methods have been developed in related research (Weiss and Lai 1999; Clements and Northrop 2002; Gomaa 2005; Saleh and Gomaa 2005). Also, the Software Engineering Institute (SEI) has defined a set of guidelines for SPL development that is based on the framework developed by Clements and Northrop in (Clements and Northrop 2002).

SPL development consists of SPL engineering and application engineering. SPL engineering is the development of core assets for a family of systems that comprise the application domain. Core assets are the requirement models, design models, implementation, documentation, test specifications and any other artifacts used in the development of the software product line. Application engineering is the selection and customization of these assets for an application of the family.

Some SPL modeling methods use several modeling views of a SPL, referred to as multiple-view modeling. Multiple-view modeling includes a functional modeling view to describe the functions of the SPL, a static modeling view to provide a structural perspective of the SPL, and a dynamic modeling view to provide a behavioral perspective of the SPL (Gomaa and Shin 2004). The functional modeling view can be described with both feature and use case models. The feature model, first introduced by Kang, is a diagram that distinguishes between the commonalities and variabilities among the applications of a SPL (Kang 1990). A feature model describes common, optional and alternative features, and relationships between these features. A *feature* is a requirement or characteristic that is provided by one or more applications of a SPL. A *common* feature

is present in every application of the SPL; an *optional* feature is present in some applications of the SPL, and an *alternative* feature is mutually exclusive with other features in the SPL (Kang 1990; Gomaa 2005).

The Product Line UML-Based Software Engineering (PLUS) method describes a feature-oriented modeling method, process and notation for SPLs based on the UML notation. The PLUS modeling method is broken down into three phases: requirements, analysis and design (Gomaa 2005).

In the requirements modeling phase, a SPL engineer creates a feature model, use case model and a table describing the feature to use case relationships (Gomaa 2005). The feature model in PLUS is based on the feature model introduced by Kang in (Kang 1990), but is described using the meta-classes and stereotypes of the UML notation. With PLUS, UML stereotypes are applied to differentiate between «common feature», «optional feature» and «alternative feature» (Gomaa 2005). Furthermore, feature groups, which place a constraint on how certain features can be selected for a SPL member, such as mutually exclusive features, are also modeled using meta-classes and given stereotypes, e.g., «zero-or-one-of feature group» or «exactly-one-of feature group» (Gomaa 2005). The use cases in PLUS are labeled with the stereotypes «kernel», «optional» or «alternative» (Gomaa 2005). A kernel use case is required by all members of the SPL. Other use cases are optional, in that they are required by some but not all members of the SPL. Some use cases may be alternative, that is different versions of the use case are required by different members of the SPL. In addition, variation points specify locations in the use case where variability can be introduced (Jacobson, Martin

Griss et al. 1997; Gomaa and Webber 2004; Gomaa 2005). A feature to use case relationship table is used to associate features with use cases and use case variation points in PLUS (Gomaa 2005).

In the analysis modeling phase of PLUS, a SPL engineer creates a static model, communication diagrams, and statecharts from the use case model. Reuse stereotypes in the static model categorize classes as kernel, optional, or variant. A feature to class relationship table is used to associate features with classes, and feature conditions are used to associate a feature to model elements in the communication diagrams and statecharts. A feature condition is a variable that associates a model element to a feature in a feature model, where the values of the feature condition represent possible feature selections. Selecting values for the feature conditions configures the models for an application derived from the SPL (Gomaa 2005).

In the design phase of PLUS, a SPL engineer defines the software architecture in terms of components and connectors. Feature conditions are also used in this phase to relate features to the model elements in the software architecture (Gomaa 2005).

Figure 1 is an overview of the SPL development processes used with the PLUS method. A SPL engineer creates reusable requirements models, analysis models and architecture during SPL engineering and then stores these models into a SPL repository. Then an application engineer reuses and configures these models for an application derived from the SPL. The process is iterative, meaning that any unsatisfied requirements, errors, and adaptations discovered during application engineering are sent

back to the product line engineer, who evolves the SPL models and updates the SPL repository (Gomaa 2005).

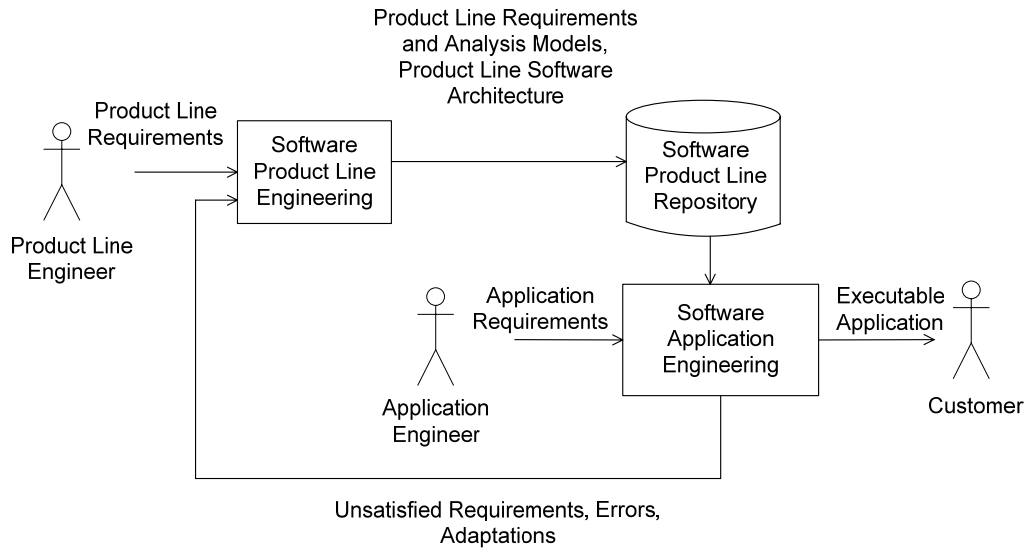


Figure 1 SPL development processes used with PLUS

The Variation Point Model (VPM) is a SPL development method that maps use case variation points to an application architecture. In VPM, a variation point identifies one or more locations at which change will occur, and the mechanism for a reuser to extend it. VPM describes how the parameterization, inheritance, and callback variability mechanisms are used on the class and sequence diagrams of the system architecture. In the class diagram, parameterization associates a parameter in a class operation to a variation point, and parameter values to variation point values. Inheritance associates an abstract class to a variation point, and its specialized classes to variation point values. In the sequence diagram, callback associates the interface of a class to a variation point, and its realization to a variation point value (Webber 2001). The next section describes these

variability mechanisms and other variability mechanisms used to configure the applications of a SPL.

3.4 Variability Mechanisms

A variability mechanism is a technique that enables automatic configuration of the variability in an application's requirements, models, implementation and test specifications. A variability mechanism is used with a SPL development method to automate the configuration of the applications of a SPL. Variability mechanisms have been applied to SPL requirements (Jarzabek 2003), SPL implementations (Anastasopoulos and Gacek 2001; Muthig and Patzke 2003), and SPL tests (McGregor, Sodhani et al. 2004) . Some examples of variability mechanisms are:

- *Aggregation*: Aggregate objects implement the common functionality and reference contained objects that implement the variant functionality
- *Inheritance*: Base classes implement the common functionality and specialized classes implement the variant functionality
- *Parameterization*: Parameters relate to variation points in the core assets, and the parameter values correspond to a variation point variant
- *Frames*: Common code is separated from variant code in separate frames. Frames are assembled to create application-specific assets (Bassett 1996; Zhang and Jarzabek 2004).
- *Aspects*: An aspect facilitates separation of concerns by separating variant code into an aspect file. Aspect weaving merges the code from the aspect file with the common code (Kiczales, Lamping et al. 1997).

Several quality criteria have been described for evaluating variability mechanisms with respect to the construction of product line assets (Anastasopoulos and Gacek 2001; McGregor 2001; Tirila 2002). Some of these quality criteria are:

- *Binding time*: The time at which the variability is bound to the asset, which can be at pre-compile time, at compile time, at initialization time, and at run-time.
- *Scope*: The smallest entity of variability supported by the mechanism
- *Flexibility*: The binding times supported by the variability mechanism
- *Efficiency*: The overhead required to support the variability in the asset using the variability mechanism
- *Separation of Concerns*: The ease with which the variability and commonality in the assets can be decoupled using the variability mechanism.
- *Traceability*: The ease with which the assets can be traced to the features and requirements of the SPL.
- *Modifiability or adaptability*: The ease with which the assets can be modified during product line evolution using the variability mechanism
- *Configurability*: The ease with which the assets can be combined and configured for different application configurations of a product line using the variability mechanism

Separation of concerns and its impact on the SPL development process is described in more detail in the next section.

3.5 Separation of Concerns

Separation of concerns is the principle that a given problem involves different kinds of concerns, or aspects, which should be identified and separated in order to achieve the required engineering quality factors such as robustness, adaptability, maintainability, and reusability (Aksit, Tekinerdogan et al. 1996). Variability mechanisms that emphasize separation of concerns, such as Aspect Oriented Programming (AOP) (Kiczales 1996; Kiczales, Lamping et al. 1997) and frames (Bassett 1996), have been created to achieve cohesion and decoupling of concerns in the software models and implementation of a single system. These variability mechanisms facilitate separation of concerns by explicitly naming, separating and associating parts of the code with the concern.

AOP identifies and separates those system aspects, or characteristics, that cut across each other and executable code. These system aspects, also called cross-cutting concerns, increase the complexity of the implementation due to code tangling and scattering of the concern. Code tangling means that the concern is interwoven with the functional code of the modules and with the code of other concerns in the system. Scattering means that the code of a concern is spread out over the system modules, rather than being localized, or grouped into one location.

AOP uses join points, pointcuts and advice can to modularize and physically separate these cross-cutting concerns from the code. Join points are the locations in the code that will be modified by a cross-cutting concern; a pointcut is a predicate over join points, and describes the join points that will be modified by a concern; an advice

consists of a pointcut and a body, which encapsulates the implementation of a concern. An aspect weaver executes the code in the body of the advice at the join point locations described by the pointcut (Kiczales 1996).

Besides AOP, frame technology has been used to achieve separation of concerns in program code. Frame technology decomposes the solution for a problem into frames, or default parts that can be reused, adapted, and composed to describe a group of similar solutions. Frames are organized into a frame hierarchy. The more specific and context-sensitive frames at the top of the hierarchy can adapt or select the more general and context-free frames at lower levels in the hierarchy (Bassett 1987; Bassett 1996).

Both Frames and AOP have been extended for SPLs. Zhang and Jarzabek created the XVCL frames variability mechanism for handling variants in SPLs. A product line engineer applies separation of concerns principles to identify the commonality and variability in the textual documents a product line, such as the requirements and implementation. The engineer then uses XVCL to organize these assets into a layered frame hierarchy called an x-framework. An application engineer describes the configuration of an application of the SPL in a specification x-frame (SPC), and then runs the XVCL processor on this framework to configure the requirements and implementation assets for that application (Zhang and Jarzabek 2004).

Loughran et al combined concepts from AOP and Frames into a language called Framed Aspects to better address the configuration and evolution of features in a SPL. Framed Aspects uses a subset of XVCL together with AOP's join points, pointcuts and advice. Framed Aspects allows the variability of an implementation to be set at pre-

compile time using Frames, and also allows program behavior to be modified at run-time using AOP (Loughran and Rashid 2004; Loughran, Rashid et al. 2004). XVCL and Framed Aspects define feature models, and then delineate frames according to the features in the feature model.

Lee et al also use feature models in their approach, and describe how Feature Oriented Analysis (FOA) can be combined with AOP in order to enhance the reusability, adaptability and configurability of the core assets of a SPL. An engineer uses FOA to develop a feature model that distinguishes between the common and variable features in the SPL, maps these features to code and then uses AOP to separate the code of the variable features and feature dependencies from the code of the common features. Lee et al point out that in general the common features map to the structure of the base modular components, but it is possible for a common feature to map to an aspect if it cannot be encapsulated in a modular component. Also, in general the variable features map to aspects, but it is possible for a variable feature to map to a modular component (Lee, Kang et al. 2006).

These approaches use feature models together with separation of concerns, but do not explicitly model the relationships between features and code. Saleh and Gomaa developed two feature-oriented approaches that explicitly model these relationships, and use separation of concerns to customize the clients of a web-service based SPL: the Static Customization of Client applications method (SCAC), and the Dynamic Customization of Client applications (DCAC) method. Both methods use a feature description language

and SPLET tool to explicitly associate and group variable code with optional or alternative features in the SPL feature model.

An engineer can apply either one of these customization methods to generate applications from the SPL implementation. The SCAC method binds the optional and alternative source code selected for an application with the common code at pre-compile time, while the DCAC method integrates parameterized variable code with the common code, and then reads a customization file during system initialization to enable the code associated with the features selected for an application of the SPL (Gomaa and Saleh 2005; Saleh 2005; Saleh and Gomaa 2005).

3.6 Software Testing of Single Systems

The process of software testing involves choosing one or more software testing strategies, using the strategies to generate test cases, executing the System Under Test (SUT) with the test cases, and verifying that the output is correct for all test inputs. A software testing strategy constructs a model that captures some properties of a SUT, applies an analysis technique to identify those properties in the SUT, and provides a means of generating test cases to exercise those properties. The analysis technique identifies a test coverage criterion, a description of the properties of a program that must be exercised to constitute a thorough test. A collection of test cases, or test suite, is adequate if it exercises all of the properties captured by the analysis technique.

Software testing strategies can be black box, white box or gray box, or a combination of both. White box techniques examine the structure of the system to create tests, while black box techniques examine the software requirements without knowledge

of the internal structure of the system. Gray box techniques are a combination of white box and black box techniques. Software testing techniques can be applied at the unit, module test, integration test, subsystem test, system test and acceptance test levels (Beizer 1990).

3.6.1 White Box Testing

White box testing techniques analyze the control-flow or data-flow properties of a program, and select tests that cover these properties. White box testing techniques model the control-flow properties of a program using a control-flow graph, which is a directed graph that represents possible execution paths in the program. Path coverage testing strategies generate and select tests for various path coverage testing criteria of the control-flow model (Miller, Paige et al. 1974).

Data flow testing strategies create tests that will detect data flow errors. These strategies use a data flow graph to model the data flow relationships of the variables defined and used in a program (Laski and Korel 1983; Rapps and Weyuker 1985; Clarke, Podgurski et al. 1989).

Predicate testing strategies create tests that will detect faults in the predicate logic of a program. Branch coverage testing is a type of predicate testing strategy that creates a test suite to cover both the true and false branches of each predicate in a program at least once. Other predicate testing strategies create a test suite to cover each simple predicate in compound predicates, or create a test suite to detect faults in compound predicates (Tai 1996).

Fault-based testing strategies examine the introduction and propagation of errors in a program's data state. Fault-based testing strategies focus on revealing certain classes of syntactic and semantic faults (DeMillo, Lipton et al. 1978; Morell 1990; Offutt 1992; Morell and Murrill 1993; Murrill, Morell et al. 2002). An example of a syntactic fault is a typing error, like the substitution of one operator for another. An example of a semantic fault is an error in the computed data state, which consists of the values of all program variables, registers (including the program counter), and file descriptors.

3.6.2 Black Box or Requirements-based Testing

Black box testing strategies test the execution of the software system against the requirements of that system. Some of these strategies, like category-partition and pair-wise testing, define heuristics on selecting input combinations. Other black box testing strategies, like state-based testing, define heuristics on selecting paths from a behavioral model of the system.

Category-partition testing partitions the input data space of a system into categories, or environment properties and input parameters. A tester describes the values, or choices for each category. A test case is a set of choices from all categories of a system (Ostrand and Balcer 1988). Boundary-value testing is a type of category-partition testing technique that selects input data values from the boundaries of a category.

Combinatorial testing techniques select tests that cover pair-wise, triple or n-way combinations of an application's parameters (Cohen, Dalal et al. 1997; Grindal 2007). Priorities and constraints can be added to combinatorial designs to favor or constrain a particular input combination (Bryce and Colbourn 2006).

Formal specification-based testing strategies create tests from formal system specifications, such as specifications written in Z notation, or specifications modeled by state machines. Stocks and Carrington describe a formal specification-based testing framework using Z notation and test templates, where a test template specifies the characteristics of the input data that can exercise a test requirement. Test templates can be used to instantiate test cases (Stocks and Carrington 1996). Amla and Amman transform Z specifications into category-partition test specifications by mapping the preconditions of a schema to input and environment variable partitions (Amla and Ammann 1992). Grieskamp et al generate scenario sequences from a use case specification with the Spec# executable specification language (Grieskamp, Tillmann et al. 2004).

State-based testing strategies model the behavior of the system with a state machine, and generate tests to cover a subset of the possible paths through the state machine. Chow defines a hierarchy for state-based testing coverage criteria such as branch (transition) coverage, and switch (transition pair) coverage, among others, based on their error-detecting power for a class of operation and transfer faults (Chow 1978). Offutt, Xiong and Lin formalize definitions for four types of test criteria for state-based testing strategies: transition coverage criterion, full predicate coverage criterion and the complete sequence coverage criterion (Offutt, Xiong et al. 1999). Offutt and Abdurazik use the above state machine specification test criteria to generate test cases from UML statecharts (Offutt and Abdurazik 1999).

3.6.3 Regression Testing

Regression testing is a maintenance task performed on a modified application to verify that the application has not been adversely affected by the modifications (Rothermel and Harrold 1994). Selective testing strategies select or regenerate a subset of the original test suite according to some criteria, such as tests that cover all modifications made to an application. The selective regression testing criteria used in the techniques described by Rothermel and Harrold select tests based on changes made to an application's implementation (Rothermel and Harrold 1994). Other regression testing criteria regenerate tests based on changes made to an application's functional requirements (Mayrhauser and Zhang 1999).

3.6.4 Model-Based Testing

Model-based testing creates test specifications and test cases from formal or semi-formal models of a software system. These models can be state machines, or other software models such as class, activity, and sequence diagrams. More than one software model can be used to create tests. Model-based testing methods that are based on a formal specification language can be mapped to an executable language to generate tests. Informal or semi-formal models need to be supplemented with additional information before they can be used to generate test specifications.

Mayrhauser et al developed a model-based automated testing environment for command-based systems. Object and command-language definitions useful for testing purposes were extracted from a class diagram in order to create test specifications for an application. Script rules, or constraints on command sequencing, were defined using state

transition diagrams. Test engineers used an automated testing tool called Sleuth to construct parameterized command sequences and command sequence templates for system testing and regression testing of several versions of an Automated Cartridge System (ACS) system and a Spacecraft Command and Data Handling system (Mayrhauser, Mraz et al. 1994; Mayrhauser 1996).

Poston developed a method to automatically derive test cases from test-ready object, dynamic, and functional OMT models (Rumbaugh 1991), where a test-ready model contains enough information to automatically generate test cases for one or more testing strategies. Testing annotations were added to the object, dynamic and functional models in order to make them test-ready. The object models described the classes and class relationships in the system, and were annotated with data domain definitions; the dynamic models described statecharts and sequence diagrams; and the functional models described data-flow diagrams, which were annotated with pre and post conditions. These models were verified for consistency with the help of tools. Then, test specifications were generated for a group of specification-based testing techniques: Functional testing, Boundary Value Analysis, Equivalence Class Analysis, Cause-effect graphing, Event-directed testing and State-directed testing. Manual projection and reference testing were used to derive the expected output values, and the test specifications were fed into a test execution tool. A case study showed that the cost of making the models test-ready was small and beneficial to developers, and that the reduction in testing effort was significant (Poston 1996).

R. Binder described a fault model and several testing techniques based on some characteristics of object oriented systems: encapsulation, inheritance, polymorphism and dynamic binding, and message sequence associated with object state. One of these testing techniques, the extended use case pattern, shows how decision tables can be created from use case descriptions (Binder 2002).

Briand and Labiche developed TOTEM, which is a model-based testing method that is used to create test cases from use cases. A test engineer uses TOTEM to create an activity diagram describing use case sequences. Then, the test engineer converts the use case descriptions to sequence diagrams and uses OCL to describe the conditions that drive a use case scenario in the sequence diagram. Next, the test engineer groups these conditions into decision tables, which can then be composed to derive system test sequences over the entire system (Briand and Labiche 2001). The decision table is based on the extended use case pattern which was first introduced by Binder (Binder 2002).

3.6.5 Test Management

Software testing techniques need to be incorporated within a test plan and test development method. The IEEE standard 829-1998 describes how to create a test plan using test design, test case and test procedure specification documents. A test plan organizes the test design, test case and test procedure specification documents. A test design specification describes a testing strategy, the functions that will be tested, and the relationship between the functions and tests. A test case specification describes a test case, which contains a test objective, inputs, outputs and test case dependencies. A test procedure specification describes the procedure for executing the tests (IEEE 1998).

3.7 Software Testing of Software Product Lines

The goal of a software testing strategy for a single application is to create tests that are effective and efficient at revealing faults in that application. SPL-based testing strategies are based on testing strategies for single applications, and are adapted to fit within an SPL development process. Besides being integrated within an SPL development process, SPL-based testing strategies differ from testing strategies for single applications in the methods used to generate, manage, and reuse test assets. These methods attempt to reduce the cost of creating, maintaining, and using the test assets over the set of applications derived from the SPL (Tevanlinna, Taina et al. 2004).

An orthogonal goal of most SPL-based testing strategies is to enable the systematic reuse of test assets over the set of applications derived from a SPL. Another orthogonal goal is to select representative applications to test from all possible applications that can be derived from a SPL. An application selection criterion is necessary in situations where the set of applications derived from the SPL is not pre-determined and is likely to change. The following describes the seminal work in this area.

3.7.1 A Testing Process for Software Product Lines

McGregor developed a testing process for the requirements-based testing of core assets and applications of a SPL. This process described how test management activities, requirements-based testing strategies, and test specifications could be adapted to fit within a SPL development process (McGregor 2001). Kolb highlighted some problems particular to testing SPLs, such as reusing generic tests and test results, and choosing which variants to test from a potentially large number of variants (Kolb 2003).

McGregor also defined some quality criteria for constructing software tests for SPLs, which include: traceability, modifiability, and configurability. *Traceability* indicates how easily the testing assets can be traced to the production software and the requirements. *Modifiability* is the ease with which the test assets can change when iterative changes are made to the SPL or to the applications derived from the SPL. *Configurability* is the ease with which the test assets can be configured to handle different combinations of variants for the different applications derived from a SPL (McGregor 2001).

To enhance configurability, McGregor recommended designing a test procedure that can be used during product line engineering and application engineering (McGregor 2001). A later paper recommended using the same variability mechanism for customizing the production software implementation and the test implementation in order to improve the traceability between tests and production software (McGregor, Sodhani et al. 2004).

3.7.2 Systematic Reuse of Use Case-Based Tests in a Software Product Line

Several use case-based testing methods for SPLs have expanded on McGregor's work. Use case-based testing strategies create tests from the use case descriptions, or from functional models developed from the use case descriptions. Most of these strategies address the problem of systematically reusing test specifications across a set of applications derived from an SPL. The following describes some of these use case-based testing techniques.

Bertolino et al. developed Product Line Use case Test Optimization (PLUTO), a method of creating tests from the use case descriptions of a SPL. Variation points in a use

case description are tagged to create a Product Line Use Case (PLUC) description. The tags in a PLUC identify alternative, optional and parametric variation points (Bertolino and Gnesi 2003). The traditional Category Partition (CP) method (Ostrand and Balcer 1988) is adapted for the PLUCs of a SPL by converting each alternative, optional or parametric tag in a PLUC into a test category, and then making each variation point value in a test category a possible test input choice. The PLUC's variations section describes constraints to limit the combination of input choices. Test instantiation is done in two steps: First, the tags corresponding to the features selected for an application derived from the SPL are enabled, selecting a set of test categories and test choices; next, a test specification is created for each valid combination of selected choices in all selected categories. The resulting test suite contains test specifications that cover all possible combinations of choices for the application (Bertolino and Gnesi 2003).

Nebut et al. developed a test design method based on the use case contracts of a SPL. Customizable use case contracts are created for a SPL, customized for an application derived from the SPL, and then used to derive use case sequences that satisfy a predicate coverage criterion. Use case contracts consist of use case precondition and postcondition predicates. Variant tags are inserted next to the variant parts of a contract, so that selecting the variant tags for an application extracts the contracts relevant to that application from the set of contracts of the SPL. Next, a Use Case Transition System (UCTS) is built for that application by linking the postcondition from one use case to the preconditions of other use cases. Nebut et al define the following predicate-based coverage criteria for the UCTS of an application: all instantiated use cases, all

precondition terms, and all false precondition terms. All instantiated use cases means that a set of test objectives exercises each instantiated use case at least once, where a test objective is a path in the UCTS. All precondition terms means that a set of test objectives exercises each use case in as many ways as there are predicate combinations to make its precondition true. All false precondition terms is a robustness testing criterion, which exercises each use case in as many ways as there are predicate combinations to make each precondition false. The robustness testing criterion tests the defensive code of the application (Nebut, Fleurey et al. 2003).

Other use case-based test design methods generate test specifications from the activity diagrams created from the use cases of a SPL. Kamsties et al. described a method of creating reusable test specifications from SPL activity diagrams, and suggested that some variability mechanisms are more suitable for describing certain types of variability. For example the parameterization mechanism, which associates variation points to parameters, is more appropriate for describing and configuring the variability in message parameters. On the other hand the fragmentation mechanism, which associates variation points to variability in the control flow of event sequences, is more appropriate for describing and configuring the variability in use case relationships (Kamsties, Pohl et al. 2003).

Hartmann et al. developed a tool that converts activity diagrams to category-partition tests for a system, and then extended that tool for a SPL. A test engineer maps product configurations to nodes and transitions in the activity diagrams. Selecting a

product configuration selects and enables the nodes and transitions in the activity diagrams that correspond to that product configuration (Hartmann, Vieira et al. 2004).

Reuys et al. further developed the creation and customization of the activity diagrams in (Kamsties, Pohl et al. 2003) with the Scenario-based Test case Derivation (ScenTED) technique and then applied it to an industrial case study at Siemens. A test engineer uses ScenTED to create hierarchical activity diagrams from the use cases of an SPL, and then uses the «variant» stereotype to identify decision nodes and activities in the activity diagrams that correspond to variation points. The test engineer uses ScenTED to trace paths from the activity diagrams during SPL engineering to satisfy the branch coverage testing criterion, and then converts these paths to test specifications, which can be manually customized for an application derived from the SPL (Reuys, Kamsties et al. 2005).

Other software models, such as a decision tree, have also been used to systematically reuse the test assets of an SPL. Instead of creating tests from use cases, Geppert et al. re-engineered a legacy system test suite and then configured these tests with a decision tree for an application of the SPL (Geppert 2004).

Unlike other approaches that investigate systematic reuse, Kishi et al. use a feature model to associate features to optional transitions in state machines, and then create a reusable verification model, which can be configured to verify the design of an application derived from a SPL. The verification model consists of a feature model, component diagram, and state machines. Optional features in the feature model are associated with optional components, optional transitions, and optional state machines. A

state machine representation is created for each actor and component, and a model checker can be used to verify properties expressed in Linear Temporal Logic (LTL) on the state machine representations (Kishi and Noda 2004; Kishi, Noda et al. 2005). However, the purpose of the reusable verification model is to verify the design of applications derived from a SPL, not to design reusable test specifications that can be customized for an application derived from the SPL.

3.7.3 Variability Management with Separation of Concerns

Pesonen et al. considered the problems and benefits of applying a separation of concerns technique together with a conventional object-oriented approach to configure the test procedures for a SPL. The paper suggested that aspects would be beneficial for implementing features that do not disturb conventional development in both the test procedures and the code, such as a logging feature. Further, the paper suggested that aspects would eliminate code tangling in test procedures without degrading performance. However, Pesonen et al stressed that without a systematic method and tool, the ad-hoc use of aspects can add to the problem of managing variability by making it difficult to trace a feature to the aspect implementation and program execution logic (Pesonen, Katara et al. 2005).

3.7.4 Selecting Representative Applications to Test

The methods in the previous sections described how to create reusable test specifications that can be selected and customized for an application derived from a SPL, but did not address the problem of selecting representative applications to test from the

configuration space of a SPL. In some situations, the set of applications derived from the SPL is not pre-determined and is likely to change. In this case, a set of representative configurations needs to be selected during early system testing in order to test for and detect faults caused by the selection of features and feature combinations. Since an SPL can contain many features which can be combined in different ways to create many application configurations, often it is not practicable to test every possible application that can be derived from the SPL.

A few methods describe how to select a set of representative applications to test from the configuration space of a SPL. McGregor used combinatorial designs, such as pair-wise testing (Cohen, Dalal et al. 1997), to select a representative set of applications to test (McGregor 2001). In contrast, Scheidemann selected an optimal set of application configurations, such that verifying the correctness of this set implied verifying the correctness of all possible application configurations. Scheidemann's method defines two types of relations: relationships between a configuration and the requirements relevant for that configuration, and relationships between a requirement and the architectural components relevant for that requirement (locality sets). This method uses a greedy algorithm to select the minimum set of configurations necessary to verify all requirements for all configurations, based on the assumption that the verification result for each requirement is independent of the behavior of the architecture elements that are not in the locality set (Scheidemann 2006).

3.8 Comparison and Analysis of Related Research on Software Testing SPLs

Most work on requirements-based testing of SPLs addressed the problem of systematically reusing the test specifications of the SPL (Bertolino and Gnesi 2003; Kamsties, Pohl et al. 2003; Nebut, Fleurey et al. 2003; Geppert 2004; Reuys, Kamsties et al. 2005). Bertolino and Gnesi's technique creates customizable use case descriptions, which can be adapted to generate category-partition tests for an application derived from the SPL (Bertolino and Gnesi 2003). Nebut et al.'s technique creates customizable use case contracts, which can be adapted to generate use case sequences for an application derived from the SPL (Nebut, Fleurey et al. 2003). Reuys et al.'s technique develops use case-level and system-level activity diagrams from use case descriptions, generates black-box test specifications that correspond to paths traced from the SPL activity diagrams, and then customizes the test specifications for an application derived from the SPL (Reuys, Kamsties et al. 2005). These test design methods are appropriate for the functional testing of a set of predetermined applications derived from the SPL.

Requirements-based SPL test design methods that address systematic reuse apply a variability mechanism to automate the configuration of test specifications for an application derived from the SPL. Kamsties et al. investigated the use of variability mechanisms in SPL-based test design methods, and suggested that the certain variability mechanisms are more suitable for representing and configuring certain types of functional variability (Kamsties, Pohl et al. 2003). However, requirements-based SPL test design methods that address systematic reuse (Bertolino and Gnesi 2003; Kamsties, Pohl et al. 2003; Nebut, Fleurey et al. 2003; Geppert 2004; Hartmann, Vieira et al. 2004;

Reuys, Kamsties et al. 2005) assume one type of functional variability in a SPL, such as a predicate in a use case contract, and arbitrarily select one type of variability mechanism, such as parameterization, to configure that functional variability. In reality, a variation point can represent varying degrees of a functional variability, from the coarse-grained functional granularity associated with an extension use case, to the fine-grained functional granularity associated with a parameter in a use case step. Existing requirements-based SPL test design methods do not provide a feature-oriented approach for representing and distinguishing between different granularities of functional variability, and for choosing a suitable variability mechanism configure this variability.

A few papers addressed the problem of selecting representative applications to test from the configuration space of a SPL, in situations where the applications derived from the SPL are not pre-determined and are likely to change (McGregor 2001; Scheidemann 2006). McGregor described how combinatorial testing strategies (Cohen, Dalal et al. 1997) can be used to select feature combinations for a set of application configurations (McGregor 2001), while Scheidemann used a feature model and greedy algorithm to select a minimal set of representative configurations, such that successful functional verification of this set implied the correctness of the entire SPL (Scheidemann 2006). These methods provided a feature-oriented approach to select representative applications from a SPL, but did not extend the feature-oriented approach to create reusable test specifications that could be customized to test these applications. Representing, analyzing and managing the relationships of features to test specifications can help a test engineer to identify possible feature interactions, decide on a set of

representative application configurations to cover these feature interactions, and then customize the test specifications for each application.

Related research described several types of functional test coverage criteria for black-box system testing of a single application. Some of these criteria have been extended for an SPL, such as the category-partition coverage criterion (Bertolino and Gnesi 2003) and the activity branch coverage criterion (Hartmann, Vieira et al. 2004; Reuys, Kamsties et al. 2005). However, these approaches lack traceability between the use cases, features and test specifications in that it is difficult to determine what use case scenarios and features have been tested in each application relative to the total number of use case scenarios and features in the SPL. A feature-oriented test design method can help a test engineer choose suitable use case scenario-based and feature-based test coverage criteria to apply during SPL engineering and application engineering, and to determine what use case scenarios and features have been tested in each application relative to the total number of use case scenarios and features in the SPL.

Previous research on systematic reuse focuses on the configurability and reusability of test specifications but does not provide a method of representing and managing the relationships of common and variable features to the test specifications of an SPL. On the other hand, previous research on selecting a set of representative applications to test does not extend the feature-oriented approach to create reusable test specifications that can be customized during feature-based application derivation for an application derived from the SPL. This research combines a feature-oriented approach with a use case-based test design method for SPLs to allow a test engineer to create

reusable test specifications to cover the use case scenarios in a SPL, select a set of representative application configurations to cover the features and feature combinations in a SPL, and choose and apply a variability mechanism to automate the selection and customization of these test specifications for each application.

Table 1 compares existing requirement-based SPL test design methods against CAdET, a feature-oriented model-based test design method for SPLs developed in this research. Some papers based on this research were published in (Gomaa and Olimpiew 2005; Olimpiew and Gomaa 2005; Olimpiew and Gomaa 2005; Olimpiew and Gomaa 2006; Gomaa and Olimpiew 2008). The methods are ordered by the date in which the method was introduced. If the method is given a name in related research, the method's name is shown as the first entry in each column; otherwise the author's name is shown as the first entry in each column. The criteria used to compare the methods are shown as the first entry in each row, such as whether the test design method is used to create reusable test specifications, is feature-oriented, or is used to select representative configurations to test. McGregor's SPL testing process framework (McGregor 2001) is not included in the comparison because it does not describe and evaluate a specific testing technique for SPLs, but rather addresses several issues that need to be considered when testing SPLs.

Table 1 Comparison of SPL testing methods

	PLUTO (Bertolino and Gnesi 2003)	Nebut (Nebut, Fleurey et al. 2003)	Geppert (Geppert 2004)	ScentTED (Reuys, E. Kamsties et al. 2005)	Scheidemann (Scheidemann 2006)	CADeT (This research)
Creates reusable test specifications	Yes	Yes	Yes	Yes	No	Yes
Feature-oriented	No	No	Yes	No	Yes	Yes
Selects representative applications	No	No	No	No	Yes	Yes
Functional variation point	Statement in use case description	Predicate in use case contract	Parameter in requirements	Decision in activity diagram	Informal Requirement	Use case scenario, variation point in use case scenario
Variability mechanisms	Parameterize use case statement	Parameterize use case predicates	Parameterize requirements	Parameterize conditions in decision nodes	Not applicable	Parameterization, Separation of concerns
Extent of functional coverage	All use cases and use case scenarios for an application	All instantiated precondition terms, etc... in contracts for an application	Selected parameters and parameter values for an application	All branches in all activity diagrams of SPL	Not applicable	All use case scenarios of SPL; all use case scenarios selected for an application
Extent of variability coverage	All combinations of variation point values for an application	Selected combinations of variation point values for an application	All features selected for one application	Selected combinations of variation point values for an application	Minimum set of configurations to verify all SPL requirements	All features and selected feature combinations of SPL
When to apply coverage criteria	Application engineering	Application engineering	Application engineering	SPL engineering	SPL Engineering	SPL and application engineering

4 Extending Model-Based Testing for Software Product Lines

This section describes how model-based testing for a single application is extended to create a feature-oriented model-based testing method for a SPL, and then explains how this method is incorporated within an evolutionary SPL development process. Customizable Activity Diagrams, Decision Tables and Test Specifications (CADET) and Customizable Activity Diagrams, Decision Tables and Test Specifications using Separation of Concerns (CADET-SoC) are the names of two feature-oriented model-based test design methods developed in this research to create functional test specifications for a SPL.

4.1 Incorporating CADeT and CADeT-SoC within a SPL Development Process

CADET and CADET-SoC can be incorporated within an evolutionary SPL development process which supports a feature-oriented use case-based requirements modeling method. A feature-oriented use case-based requirements modeling method uses feature models to distinguish between the commonality and variability in a SPL, and use case models to describe the functional requirements of the SPL.

The Product Line UML-Based Software Engineering (PLUS) (Gomaa 2005) method can be used within the SPL development process shown in Figure 2. PLUS is a feature-oriented UML-based design method that uses both feature modeling and use case

modeling to describe the SPL requirements. A test engineer uses CAdET or CAdET-SoC to create functional test specifications from SPL feature and use case requirement models created using the PLUS method.

Figure 2 shows how CAdET and CAdET-SoC impact the SPL development processes used with PLUS (shaded in gray). A SPL engineer develops the SPL requirement models using PLUS. Then, a test engineer uses CAdET to develop customizable activity diagrams, decision tables, and test specifications from the feature and use case SPL requirements models. During application engineering, an application engineer applies feature-based application derivation to derive one or more applications from the SPL. A test engineer uses CAdET to apply feature-based test derivation to select and customize the test specifications for each application, and then test each application.

Any unsatisfied requirements, errors and adaptations are sent back to the SPL engineers, who change the reusable assets and store them in the SPL repository. The process of updating the reusable assets, selecting features, deriving and testing each application is repeated until the applications are ready to be delivered to the customers.

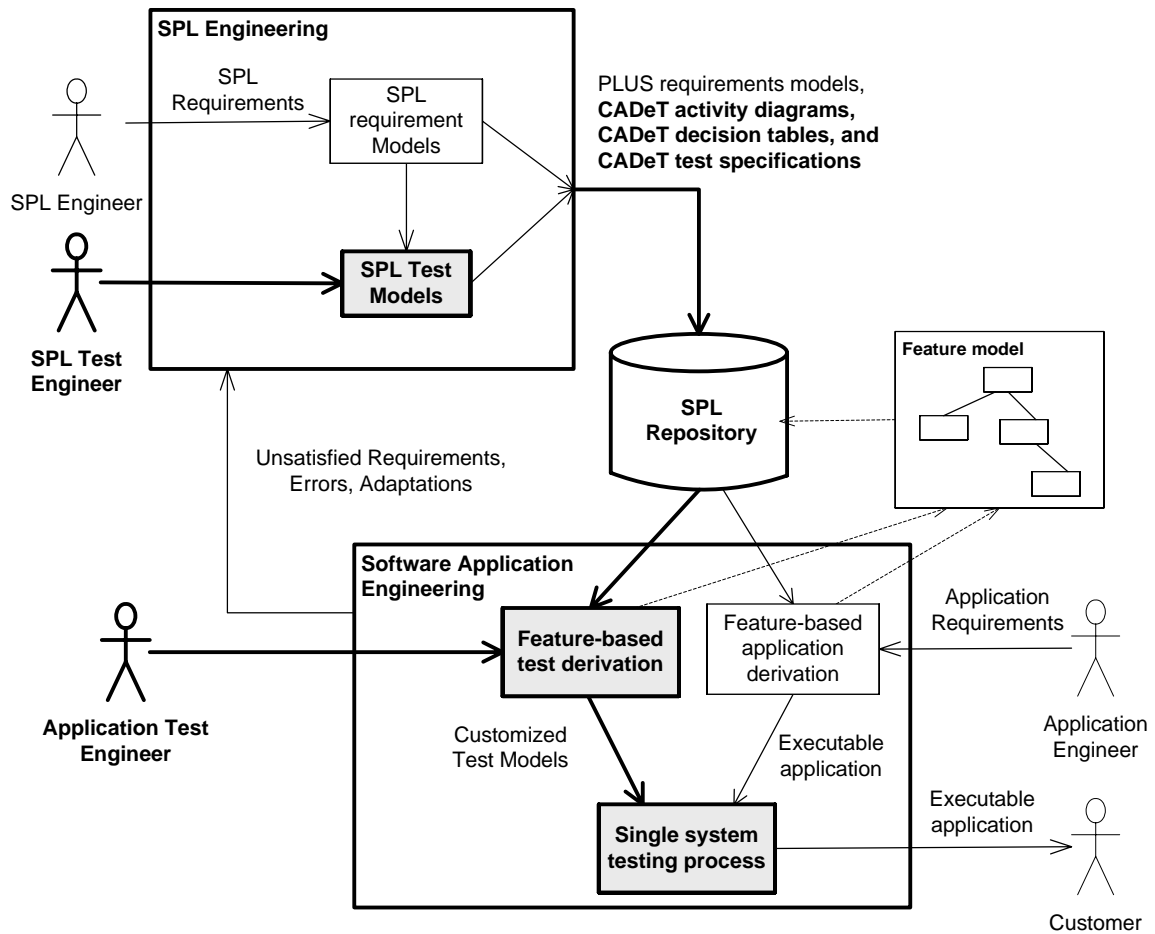


Figure 2 Incorporating CADeT within the SPL development process of PLUS

The test models created using CADeT and CADeT-SoC are organized around the test plan structure described in the IEEE Standard for Software Test Documentation (IEEE 1998). This standard describes the structure of the test design, test case, and test procedure specification documents. A test design specification describes a testing strategy, the functions that will be tested, and the relationship between the functions and tests. A test case specification describes a test case, which contains a test objective, inputs, outputs and test case dependencies. A test procedure specification describes the

procedure for executing the tests. These documents are supplemented with graphical test models, such as activity diagrams, to facilitate the identification, creation and derivation of functional tests for an application.

Then, these models are extended for a SPL using a feature-oriented approach to manage the variability in the test models, and reuse these models to generate test specifications for a set of applications derived from the SPL. The next section describes the model-based testing method for a single application used in this research, and the following section describes how this method is extended for a SPL.

4.2 Model-Based Testing for a Single Application

CADeT and CADeT-SoC build on model-based functional test design methods for single applications. Some model-based functional test design methods for single applications use UML activity diagrams to describe the processing flow and logic of a use case (Vieira, Johanne Leduc et al. 2006), while other methods use UML activity diagrams to describe use case sequencing, UML sequence diagrams to describe the flow of events in a use case, and decision tables to describe the logic of use case scenario sequences (Briand and Labiche 2001). In this research, the flow of events in a use case is described with activity diagrams as in (Hartmann, Vieira et al. 2004; Reuys, Kamsties et al. 2005; Vieira, Johanne Leduc et al. 2006) rather than sequence diagrams, because activity diagrams allow use case activities to be organized hierarchically, and are more precise at depicting how the main and alternative sequences of the use case fit together. The decision tables of (Briand and Labiche 2001) and (Binder 2002) were also included

and extended to allow a test engineer to relate a set of use case conditions and activities to paths traced from an activity diagram for each use case scenario.

Figure 3 shows the meta-model of a model-based functional test design method for a single system used in this research. A use case requirements model describes one or more use cases. Each use case is converted to an activity diagram, in order to formalize the activity flow and logic of the use case. A path is traced from an activity diagram for each use case scenario, and then converted into a column in a decision table, which identifies the conditions and activities covered by the path. The decision table is then used to generate test specifications.

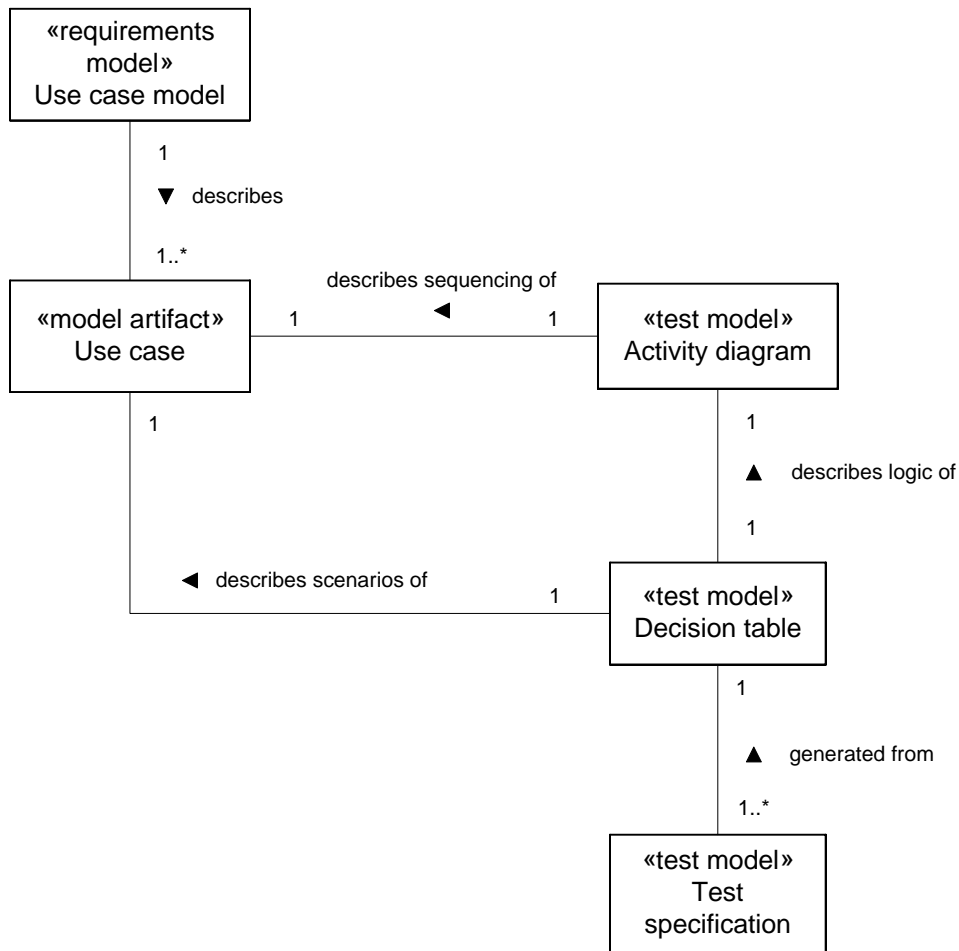


Figure 3 Meta-model describing model-based testing for a single application

4.3 Model-Based Testing for a SPL

The model-based testing method for a single application is extended in this research to create a feature-oriented model-based testing method for a SPL. Figure 4 is a meta-model that describes the relationships between the requirement models of PLUS and the functional models in CADeT. The feature to use case relationship table in PLUS associates the features from a SPL feature model with use cases from a SPL use case model (Gomaa 2005). One feature is associated with one or more use cases, and one use

case is associated with one or more features. An activity diagram and a decision table are created for each use case in the use case model. The feature to use case associations are then applied to the activity diagram and decision table created from each use case. Each feature in the feature model is associated with one or more activity diagrams, and each activity diagram is associated with one or more features. Each feature in the feature model is associated with one or more decision tables, and each decision table is associated with one or more features. Further, each feature in the feature model is associated with one or more test specifications, or one or more variation point points in a test specification. Each test specification, or variation point in a test specification, is associated with one or more features. Mapping features to the activity diagrams, decision tables, and test specifications allows these models to be selected and configured for an application derived from the SPL. The next chapter describes the CADeT method.

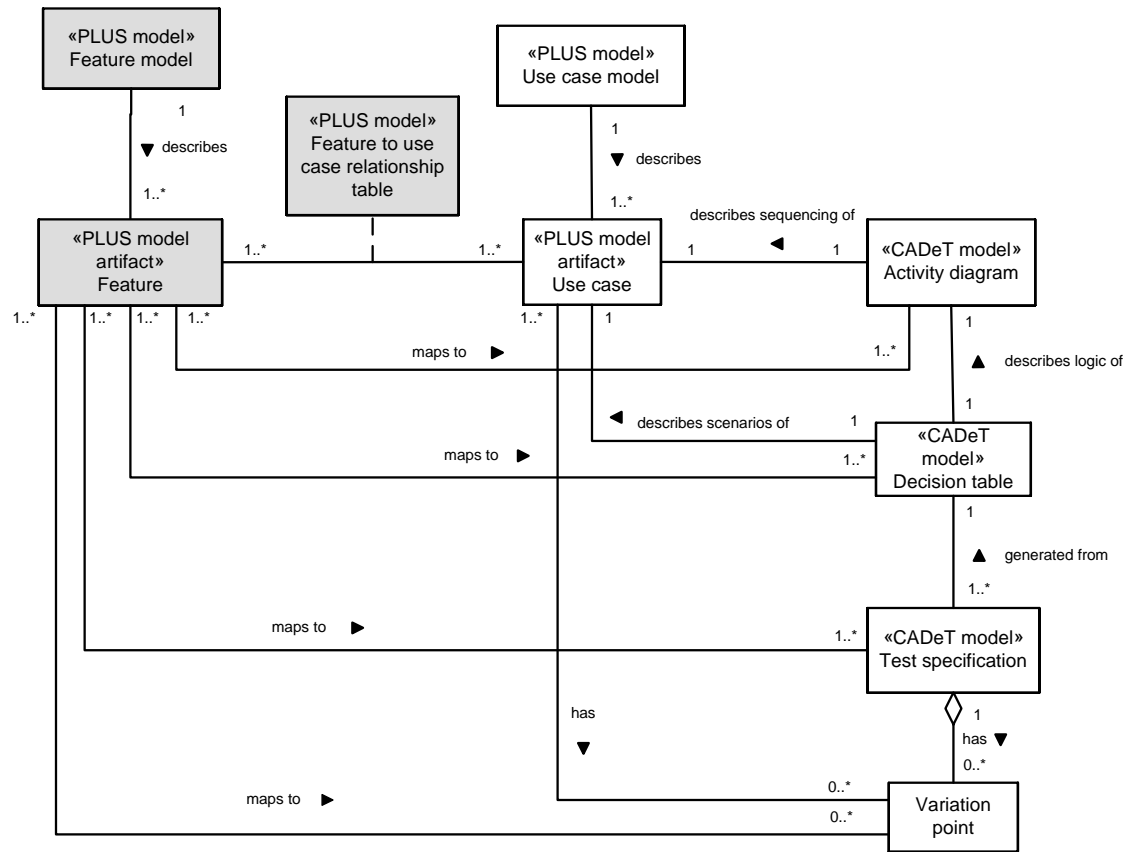


Figure 4 Feature-oriented model-based testing for a SPL

5 CADeT: A Model-Based Testing Method for SPLs

This chapter describes the Customizable Activity Diagrams, Decision Tables and Test Specifications (CADeT) method for Software Product Lines, and illustrates its application to excerpts of an Automated Highway Toll System (AHTS) SPL case study. CADeT is a feature-oriented test design method that can be used to create functional test specifications for SPLs. These test specifications can be reused and configured during feature-based test derivation to test a set of applications derived from the SPL. The following sections describe how CADeT is used to create these test specifications during SPL engineering, and how these test specifications are configured during application engineering to test a set of applications derived from the SPL.

5.1 Developing Customizable Test Specifications During SPL Engineering

A test engineer uses CADeT to develop test models in four phases during SPL engineering:

- Phase I: Create activity diagrams from use cases
- Phase II: Create decision tables and test specifications from activity diagrams
- Phase III: Define and apply feature-based coverage criteria
- Phase IV: Apply the parameterization variability mechanism to decision tables and test specifications

Figure 5 describes the activities that correspond to these phases, and the artifacts created by these activities during SPL engineering. In Phase I, CADeT is used to develop activity diagrams from the PLUS requirement models, and in Phase II a decision table is created from each activity diagram. In Phase III CADeT is applied to define a feature-based test coverage criterion for a SPL. In Phase IV, a variability mechanism is applied to automate feature-based test derivation of the test specifications during application engineering. Throughout this process, the features in the feature model are mapped to use cases in the use case model, activities and activity diagrams, decision tables, and test specifications.

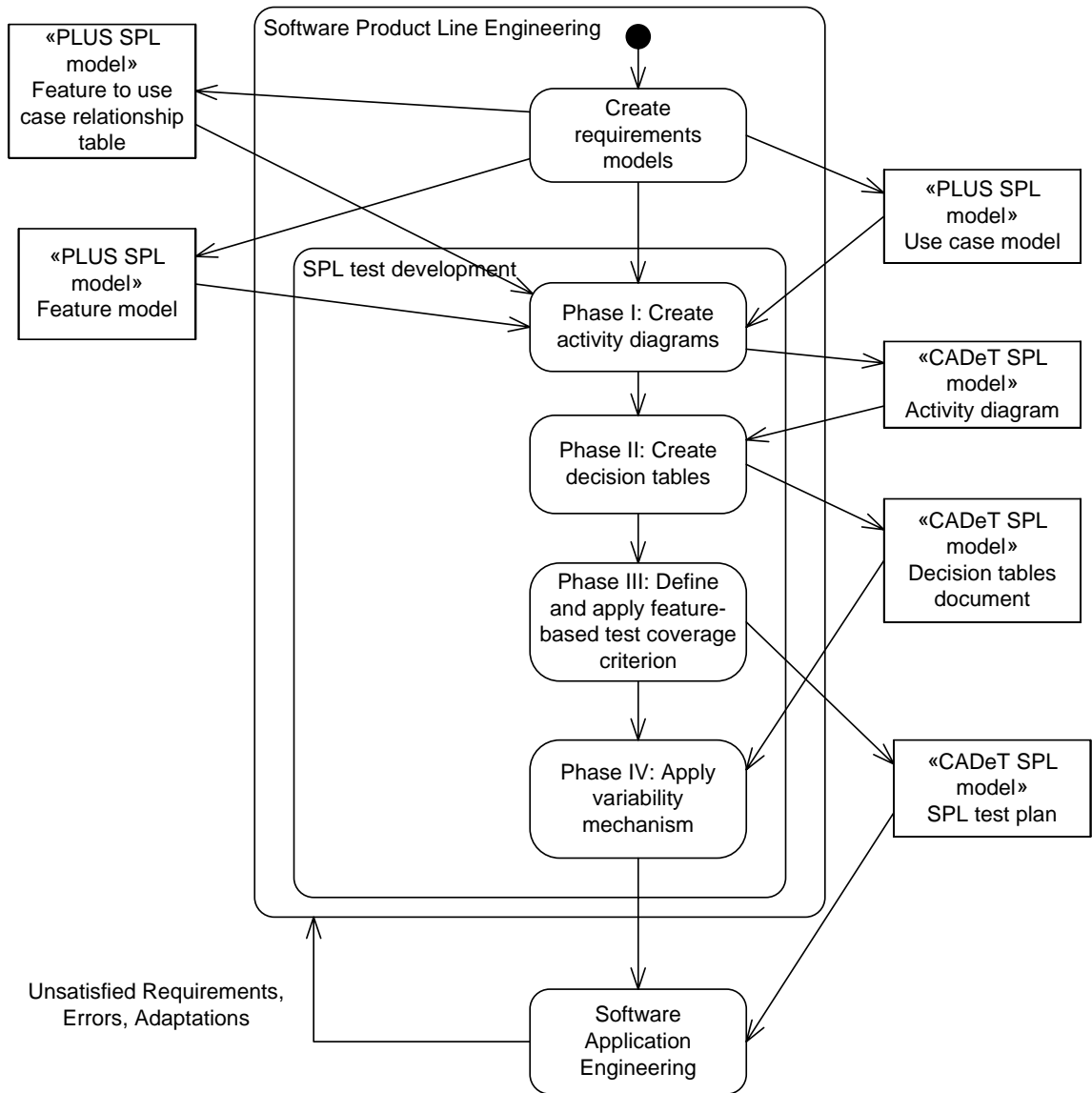


Figure 5 SPL test development activities

5.2 Phase I: Creating Activity Diagrams from Use Cases During SPL Engineering

A use case model provides informal, narrative descriptions of interactions between the actor and system in terms of a main sequence (scenario) and alternative sequences. A test case coverage criterion based on use case modeling would be to cover every scenario of the use case, namely the main scenario and all the alternative scenarios.

A more precise description of the use case scenario, in terms of sequencing and branching, can be given by an activity diagram, which depicts activity nodes and decision nodes. A test coverage criterion based on activity diagrams would be to cover all paths in the activity diagrams that correspond to the use case scenarios. In CADeT, the test coverage criterion is to cover all use case scenarios (which correspond to paths traced from activity diagrams) and all features and relevant feature combinations in the feature model of a SPL. The relevant feature combinations are described in more detail in Phase III.

In CADeT, an activity diagram is created from each use case description, similar to the ScenTED approach [5]. Unlike ScenTED, a feature model and feature to use case relationship table are used to analyze the relationships of features to activity diagrams. Analyzing these relationships helps an engineer to make informed decisions on how to best configure the variability in the functional models and test models of a SPL.

5.2.1 Role Stereotypes in CADeT

A *role stereotype* is a UML notation for classifying a modeling element by the role it plays in an application (Gomaa 2005). CADeT uses role stereotypes to distinguish between different granularities of functional variability in the activity diagrams of a SPL. Distinguishing between different granularities of functional variability makes the activity diagrams more precise for the purpose of testing. The following stereotypes distinguish between different granularities of functional abstraction in an activity node:

- A «use case» activity node, which describes a use case, as shown by the “Enter Toll Road” activity node in Figure 6.

- An «extension use case» activity node, which describes an extension use case.
- An «inclusion use case» activity node, which describes an inclusion use case.
- An «aggregate step» activity node, which groups a sequence of activities, or events, in a use case description, as shown by the “Process Ticket” activity node in Figure 6.
- An «input step», which describes an input event from the actor to the application in a use case description, as shown by the “Insert Ticket” activity node in Figure 6.
- An «output step», which describes an output event from the application to the actor in a use case description.
- An «internal step», which documents an internal (non-observable) activity in the application.



Figure 6 Example of role stereotypes

5.2.2 Reuse Stereotypes and Feature Conditions in CADeT

The feature to use case relationship table of PLUS (Gomaa 2005) is used together with reuse stereotypes and feature conditions of CADeT to analyze the impact of common, optional, and alternative features on the activity diagrams. A *feature to use case relationship table* associates a feature with one or more use cases or variation points, where a *variation point* is a location at which change can occur in a SPL (Gomaa 2005). A *reuse stereotype* is a UML notation that classifies a modeling element in a SPL by its reuse properties (Gomaa 2005). A *feature condition* is a variable that associates a model

element to features in a feature model, where the variable values represent possible feature selections.

In CAdET reuse stereotypes are applied to activity nodes rather than decision nodes as in (Reuys, Kamsties et al. 2005), since activity nodes can be abstracted or decomposed to represent different levels of functional granularity. CAdET contains the following reuse stereotypes to describe how an activity node is reused in the applications derived from the SPL:

- A «kernel» activity node, which corresponds to a «common» feature in the feature model, as shown by the “Enter Toll Road” activity node in Figure 7.
- An «optional» activity node, which corresponds to an «optional» feature in the feature model, as shown by the “Insert Ticket” activity node in Figure 7.
- A «variant» activity node, which corresponds to an «alternative» feature in the feature model.
- An «adaptable» activity node, which identifies an activity node that is associated with a use case variation point, as shown by the “Process Ticket” activity node in Figure 7.

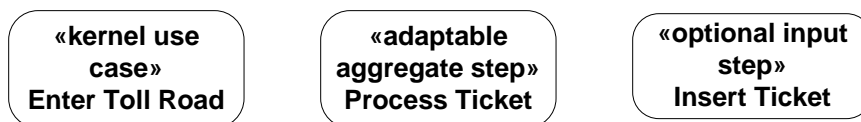


Figure 7 Example of reuse stereotypes

Reuse stereotypes are combined with the role stereotypes (Gomaa 2005) to describe how the activity node is reused in the applications derived from the SPL. The «kernel» activity node is “reused as is” in all applications derived from the SPL. The «optional» and «variant» activity nodes are “reused as is” in some applications derived

from the SPL, depending on whether the corresponding feature has been selected for the application. In contrast, the «adaptable» activity node describes a use case variation point, and needs to be customized for an application derived from the SPL. Selecting features for an application of the SPL configures the «adaptable» activity node for that application by replacing the adaptable activity node with the variants that correspond to the selected features.

Besides reuse stereotypes, feature conditions are added to associate the variability in the control flow of an activity diagram with a feature in a feature model. The values of a feature condition represent possible feature selections. Table 2 shows the feature conditions *fc* and feature selections associated with common, optional, alternative and parameterized features in the feature model of a SPL. A common feature is associated with a feature condition set to “T”, or True. An optional feature is associated with a Boolean feature condition that can be set to two possible selections “T” or “F”. A group of alternative features in a feature model is associated with a feature condition that has the alternatives as possible selections. A parameterized feature is associated with a feature condition that has discrete parameter values or a range of values as possible selections. Setting the value of a feature condition enables or disables the activities associated with the feature in the activity diagram of an application derived from the SPL.

Table 2 Feature condition selection values

Feature category	Feature condition selections
Common feature	$fc = T$
Optional feature	$fc = \{T, F\}$
Alternative feature	$fc = \{alternative1, alternative2, \dots\}$
Parameterized feature	$fc = \{parameter1, parameter2, \dots\}$

5.2.3 Creating Activity Diagrams from Use Cases

The steps of Phase I are described in more detail below:

- 1 Create an activity diagram for each use case.
 - 1.1 Stereotype the use case activity diagram as «use case»
 - 1.2 Map the events in the use case description to activity nodes in the activity diagram. Stereotype these activity nodes as «input step», «output step», «internal step» or «aggregate step»
 - 1.3 Map the use case conditions that drive the alternative use case scenarios to decision nodes and execution conditions in the use case activity diagram. An execution condition is a tester-controlled variable that affects the control flow of a path in an activity diagram.
 - 1.4 If a base use case extends or includes another use case, define the activities of the extension or included use case, and then group these activities in an activity group using a structured activity node. A structured activity node (Rumbaugh, Jacobson et al. 2005) references an activity diagram of an

extension or included use case. Stereotype the structured activity node as «extension use case» or «included use case».

1.5 Add preconditions and postconditions to each use case activity diagram as described by the UML language reference manual (Rumbaugh, Jacobson et al. 2005). A precondition describes the state of an application before a use case activity node is executed, and a postcondition describes the state of an application after a use case activity node has executed. A system state variable can be used to encode these system states.

1.6 If a use case contains variation points, analyze the impact of the variation point on the use case activity diagram. This is described in more detail in the next section.

2 Create a system level activity diagram. A *system level activity diagram* is an activity diagram that describes the sequencing between the activity diagrams associated with the use cases of an application.

2.1 Map each use case activity diagram to an activity node in the system level diagram.

2.2 New decision nodes, guard conditions, and activity nodes may be added to the diagram to show control flow dependencies between use cases.

5.2.4 Analyzing the Impact of Features on the Activity Diagrams

The feature to use case relationship table of the PLUS method (Gomaa 2005) associates a feature with one or more use cases or use case variation points, where a variation point identifies one or more locations of change in the use cases. This table is

used to analyze the impact of a feature on the system level and use case level activity diagrams. Feature conditions are created to represent features in the feature model, and used to control the execution of activities in the activity diagram depending on feature selection. The following steps describe how feature conditions are added to the activity diagrams:

3a. If a feature corresponds to one or more use cases:

- Add a feature condition to the decision nodes that control the execution of this use case in the use case and system level activity diagrams of the SPL.
- If the use case does not contain variation points, stereotype the use case activity node as «kernel», «optional», or «variant»

3b. If a feature corresponds to one or more variation points, identify which activity nodes and decision nodes in the use case activity diagram are impacted by the variation point from the use case description.

3b.1 If the variation point impacts an existing activity node:

3b.1.1 Stereotype the impacted activity node as «adaptable». Show the variation point in the parameter list of the «adaptable» activity node.

3b.1.2 If the variants of the variation point are known, create a sub-activity diagram for each «adaptable» activity node. In the sub-activity diagram:

- Describe the sequencing logic of the variation point variants.
- Associate each variant with one or more features in the feature model using feature conditions and reuse stereotypes.

3b.2 If the variation point impacts the control flow of an existing decision node

- Add control flows and activity nodes to the use case activity diagram for the variation point variant that corresponds to the feature.
- Stereotype the added activity nodes as «kernel», «optional» or «variant» depending on whether the feature is common, optional or alternative in the feature model
- Add a feature condition to the decision node to control the execution of the «optional» or «variant» activity nodes
- If possible, create an «adaptable» activity node to group the newly added activity nodes. Show the variation points in the parameter list of the «adaptable» activity node.

5.2.5 Example of Creating Activity Diagrams from Use Cases

The following example illustrates how the CADeT method is used to create the activity diagrams from the requirements models of an Automated Highway Toll System (AHTS) SPL. An AHTS consists of a series of toll roads. Each toll road has a series of fixed toll plazas consisting of one or more toll booths, which serve as collection points for the toll fees. Customers who use the toll road may pay the tolls by using a transponder placed in their vehicle or by paying with cash or credit cards at selected toll booths. An AHTS SPL is a family of Automated Highway Toll Systems that share many similarities but have some variations, such as the speed limit with which vehicles may pass through, the types of transponder devices supported, and how the toll amount is calculated.

A feature model, a use case model, and a feature to use case relationship table have been developed for an AHTS SPL using PLUS (Gomaa 2005). Figure 8 describes a

feature model created for the AHTS SPL, which contains optional Transponder Entry/Exit Booths, Full Service Entry/Exit Booths and Ticket Entry/Exit Booths features. There are mutually includes feature dependencies between the toll booth and toll booth devices, such as Transponder Entry/Exit Booths mutually includes “Transponder Account”. This dependency indicates the “Transponder Entry/Exit Booths” must be selected together with the “Transponder Account” feature.

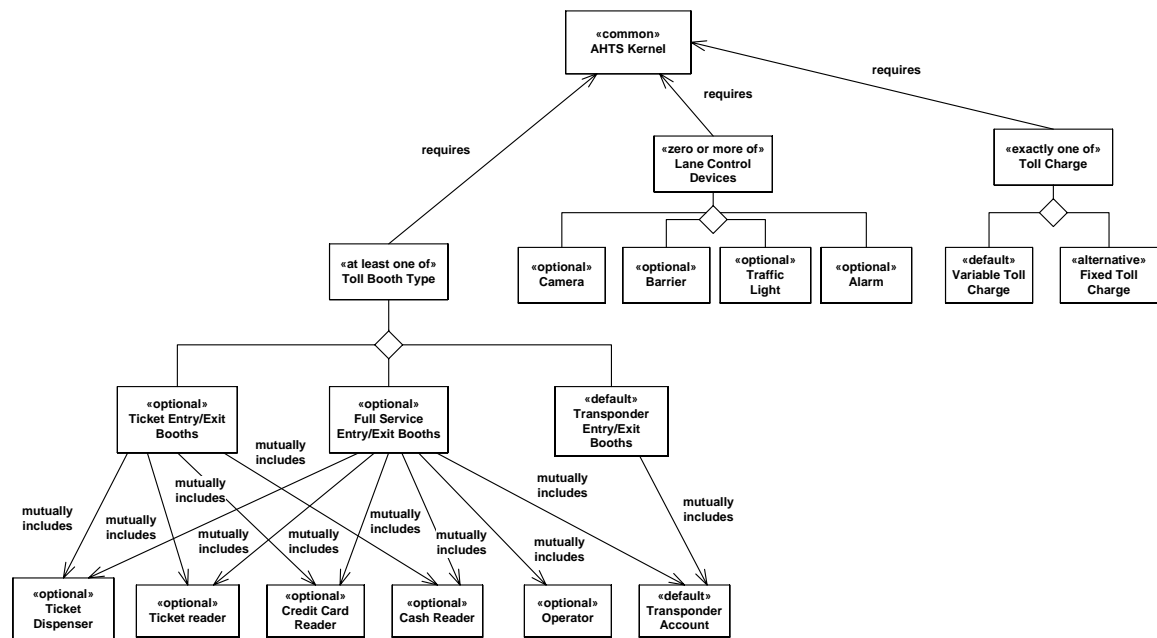


Figure 8 Feature model for AHTS SPL

Figure 9 describes part of a use case model for the AHTS SPL, which contains the “Enter Toll Road” use case, the “Enter through Transponder-enabled Booth” and “Enter through Ticket-issuing Booth” extension use cases. These use cases have been stereotyped as «kernel» and «optional» using the PLUS notation. The «kernel»

stereotype indicates that the use case is included in all applications of the AHTS SPL, and the «optional» stereotype indicates that the use case is optionally included in some applications of the AHTS SPL.

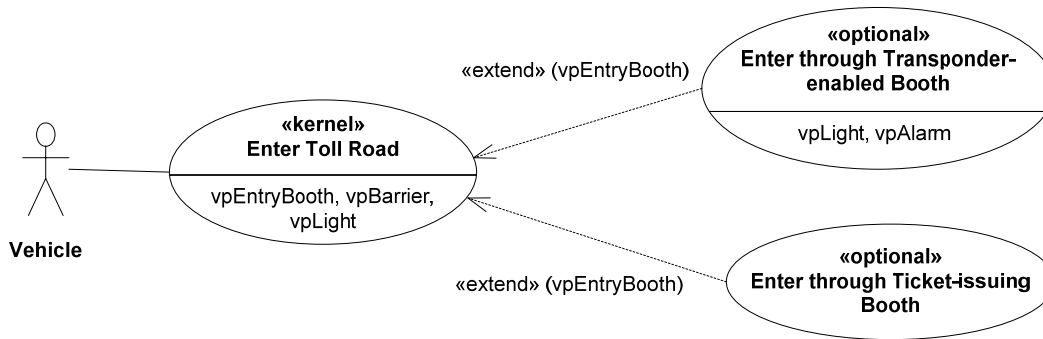


Figure 9 “Enter toll road” use case and extension use cases

The use case descriptions for the “Enter Toll Road”, “Enter through transponder enabled booth”, and “Enter through ticket issuing booth” use cases are in Figure 10, Figure 11, and Figure 12, respectively. The “Enter Toll Road” use case has three variation points: vpBarrier, vpLight, and vpEntryBooth. The variation points vpBarrier and vpLight refer to barrier and light devices which can be added to the toll booth to control access to the toll road. The variation point vpEntryBooth extends the “Enter Toll Road” use case to describe the “Enter through transponder enabled booth” extension use case in Figure 11 and “Enter through ticket issuing booth” extension use case in Figure 12.

Use case Name: Enter toll road
Reuse category: Kernel
Summary: Vehicle enters toll road through an entry booth
Actor: Vehicle
Precondition: Tollbooth is operational
Description:

1. Vehicle approaches entry booth, and system detects vehicle's presence.
2. **Extend** the "Enter through Transponder-enabled Booth" use case.
3. System authorizes vehicle entry
4. System detects vehicle departure
5. System resets toll booth

Alternatives:
None
Postcondition: The vehicle has entered the toll road

Variation Points:

Name: vpBarrier
Type of functionality: Optional
Line numbers: 3, 5
Description of functionality: An optional barrier device can be added to a toll booth. The barrier can be lowered to prevent a vehicle from leaving a toll booth (line 5) and raised to allow a vehicle to leave the toll booth (line 3).

Name: vpLight
Type of functionality: Optional
Line numbers: 3, 5
Description of functionality: An optional light device can be added to a toll booth. The light can turn red, yellow or green. A red light tells the vehicle to stop (line 5), a yellow light is a warning to the vehicle, and a green light tells the vehicle to go (line 3).

Name: vpEntryBooth
Type of functionality: Optional
Line numbers: 2
Description of functionality: If the ticket entry booth feature is selected for an application of the SPL, include a ticket printer at an entry booth, and extend the "Enter through ticket booth" use case in addition to extending the "Enter through transponder booth" use case.

Figure 10 "Enter toll road" use case description

Use case name: Enter through transponder-enabled booth

Reuse category: Kernel

Summary: Vehicle enters highway through transponder enabled booth.

Actor: Vehicle

Dependency: Extends “Enter Toll Road” use case

Precondition: Booth has detected vehicle presence

Description:

1. System scans transponder
2. If transponder is detected, system stores time and account id of trip transaction.
3. System checks transponder account
4. If transponder account is valid, system stores authorization code

Alternatives:

Line 2: If the transponder is not detected or is invalid, system stores time and id of transaction for an invalid vehicle entry and warns the vehicle.

Line 4: If the account is invalid, system stores invalid vehicle entry and warns the vehicle.

Postcondition: The transponder account has been checked.

Variation Points:

Name: vpLight

Type of functionality: Optional

Line numbers: Both alternatives

Description of functionality: The light device can be added to a toll booth. The light can turn red, yellow or green. A red light tells the vehicle to stop, a yellow light is a warning to the vehicle, and a green light tells the vehicle to go.

Name: vpAlarm

Type of functionality: Optional

Line numbers: Both alternatives

Description of functionality: The alarm device can be added to a toll booth. The alarm can sound a warning to the vehicle.

Figure 11 “Enter through transponder-enabled booth” use case description

<p>Use case name: Enter through ticket-issuing booth</p> <p>Reuse category: Optional</p> <p>Summary: Vehicle enters highway through ticket booth.</p> <p>Actor: Vehicle</p> <p>Dependency: Extends “Enter Toll Road” use case</p> <p>Precondition: Booth has detected vehicle presence</p> <p>Description:</p> <ol style="list-style-type: none"> 1. System checks ticket supply 2. System issues ticket with the time, day and booth id 3. Driver takes ticket <p>Alternatives:</p> <p>Line 1: If the machine is low on tickets it sends a low ticket warning to an operator.</p> <p>Postcondition: The driver has taken a ticket</p> <p>Variation Points:</p>
--

Figure 12 “Enter through ticket-issuing booth” use case description

Activity diagrams were created for each use case of the AHTS SPL. First, the use case steps in each use case description were mapped to activity nodes. Second, the use case conditions were identified from the description of the use case alternatives, and then mapped to decision nodes and execution conditions in each activity diagram. Third, the system state variable “Vehicle Trip” was created to encode the system states associated with the pre and post conditions of each use case. Then, an initial system level activity diagram for the AHTS SPL was created by grouping and referencing use case activity diagrams using structured activity nodes, as shown in Figure 13. This diagram shows that the “Exit Toll Road” use case activity diagram is executed after the “Enter Toll Road” use case activity diagram. The system state variable “VehicleTrip” is defined for each trip through the toll road. This variable is initially set to “NotInTollRoad” for a vehicle trip, is updated to “EnteredTollRoad” after the “Enter Toll Road” use case activity diagram is executed, and then updated to “ExitedTollRoad” after the “Exit Toll Road” use case activity diagram is executed.

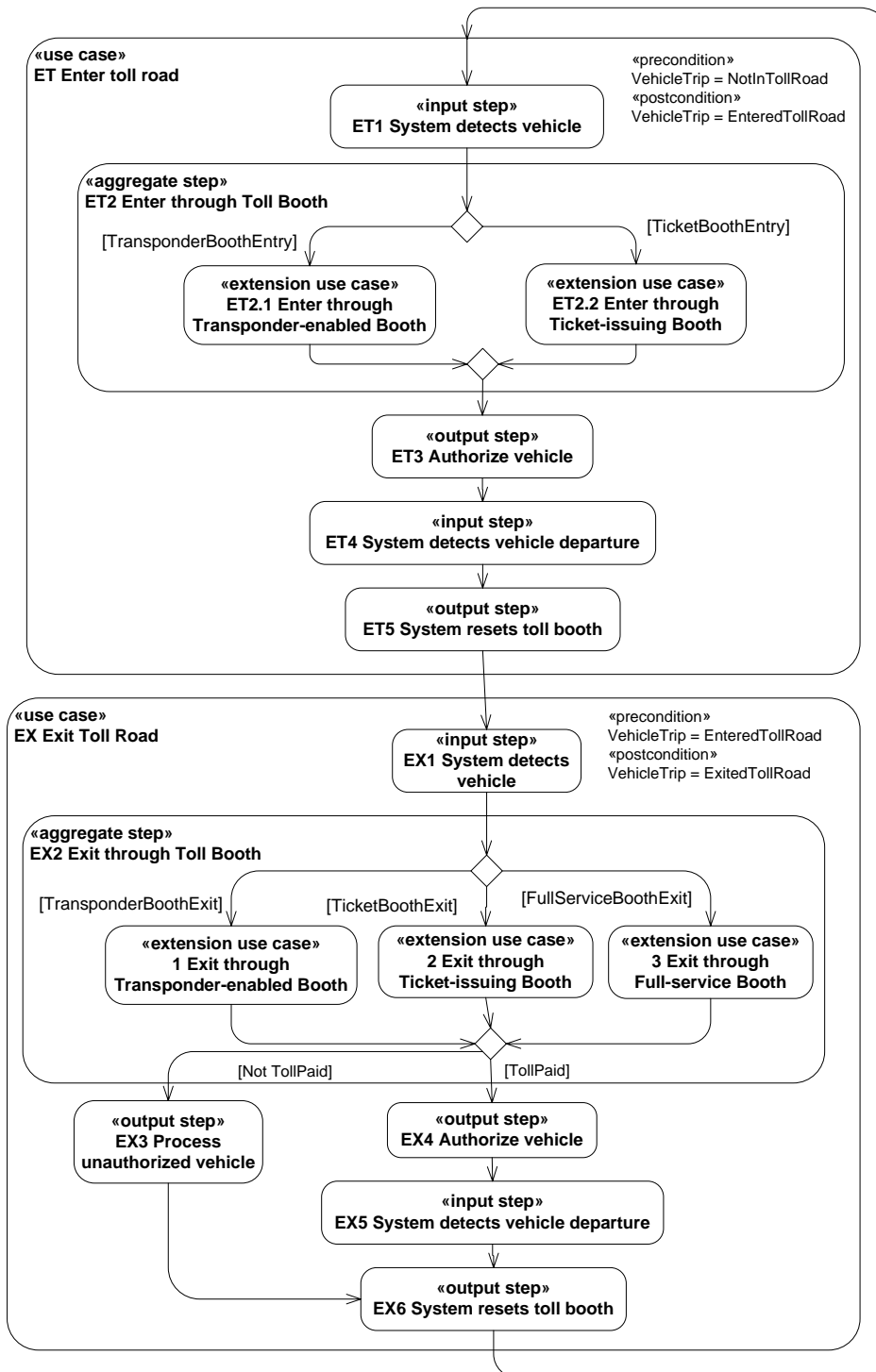


Figure 13 Initial system level diagram for AHTS SPL

Table 3 shows a list of feature conditions created to represent features in the feature model of the AHTS SPL, and to control the execution of activities in the activity diagram depending on feature selections. The AHTS SPL contains 16 features, but only 9 of these features (shaded in gray) can be explicitly selected to derive an application configuration from the AHTS SPL. The Automated Toll System Kernel feature is represented by the AHTSKernel feature condition with a feature selection of T, since this feature is always selected for an application derived from the SPL. The toll booth types and the lane control device features are optional features which map to feature conditions with $\{T, F\}$ feature selections. Each of these feature conditions will be set to 'T' if the optional feature is selected for an application of the SPL, else it will be set to 'F'. The alternative variable and fixed toll charge features are represented by the tollCharge feature condition.

Selecting a toll booth will select devices associated with that toll booth:

- IF (ticketBooth = T) THEN ((ticketDispenser AND ticketReader AND creditCardReader AND cashReader) = T)
- IF (fullServiceBooth = T) THEN ((ticketDispenser AND ticketReader AND creditCardReader AND cashReader AND operator AND transponderAccount) = T)
- IF (transponderBooth = T) THEN (transponderAccount = T)

Table 3 Feature list for AHTS SPL

Feature condition	Feature Selections
AHTSKernel	T
ticketBooth	$\{T, F\}$
fullServiceBooth	$\{T, F\}$
transponderBooth	$\{T, F\}$
camera	$\{T, F\}$
barrier	$\{T, F\}$
trafficLight	$\{T, F\}$
alarm	$\{T, F\}$
tollCharge	$\{variable, fixed\}$
ticketDispenser	$\{T, F\}$
ticketReader	$\{T, F\}$
creditCardReader	$\{T, F\}$
cashReader	$\{T, F\}$
operator	$\{T, F\}$
transponderAccount	$\{T, F\}$

Next, the impact of features on the activity diagrams of the AHTS SPL was analyzed using the feature to use case relationship table together with the use case descriptions. Table 4 describes an excerpt from a feature to use case relationship table created for the AHTS SPL.

Table 4 Excerpt of feature to use case relationship table for the AHTS SPL

Feature Name	Feature Category	Use Case Name	Use Case Category / Variation Point (vp)	Variation Point Name
Automated Toll System Kernel	common	Enter Toll Road	kernel	
		Exit Toll Road	kernel	
Transponder Entry/Exit Booth	optional, default	Enter through Transponder-enabled booth	optional	
		Exit through Transponder-enabled booth	optional	
Ticket Entry/Exit Booth	optional	Enter through Ticket-issuing Booth	optional	
		Exit through Ticket-issuing Booth	optional	
Barrier	optional	Enter Toll Road	vp	vpBarrier
		Exit Toll Road	vp	vpBarrier
Traffic Light	optional	Enter Toll Road	vp	vpLight
		Enter through Transponder-enabled booth	vp	vpLight
		Exit Toll Road	vp	vpLight
Alarm	optional	Enter through Transponder-enabled booth	vp	vpAlarm

The activity diagrams of the AHTS SPL were modified to contain feature conditions and reuse stereotypes, in order explicitly associate features in the feature model with activities in the activity diagrams. Figure 14, Figure 15, and Figure 16 show the modified activity diagrams for the “Enter Toll Road” use case and its two extension use cases. The feature conditions are underlined in the diagrams to distinguish them from the execution conditions.

The process of analyzing the impact of features on the activity diagram for the “Enter Toll Road” use case is described next. The “Enter Toll Road” use case is

associated with the “Automated Toll System Kernel”, “Barrier” and “Traffic Light” features in Table 4. The “Enter Toll Road” use case contains a variation point in the “«adaptable aggregate step» ET2: Enter through toll booth” activity node. This variation point is associated with two extension use cases: “Enter through transponder enabled booth” and “Enter through ticket issuing booth”. Since “Enter through ticket issuing booth” is optional, a “ticketBooth” feature condition is added to control the execution of the activity associated with this use case in the “Enter Toll Road” activity diagram in Figure 14.

The optional Barrier and Traffic Light features in Table 4 are associated with vpBarrier and vpLight variation points in the “Enter Toll Road” use case description in Figure 10. These variation points impact use case steps “3. System authorizes vehicle entry” and “5. System resets toll booth”, which correspond to the adaptable output steps “ET3 Authorize vehicle” and “ET5 System resets toll booth” in the activity diagrams of Figure 14 and Figure 16. The feature conditions and sequencing logic of the variation point variants associated with the vpBarrier and vpLight variation points are described in the sub-activity diagrams of Figure 16.

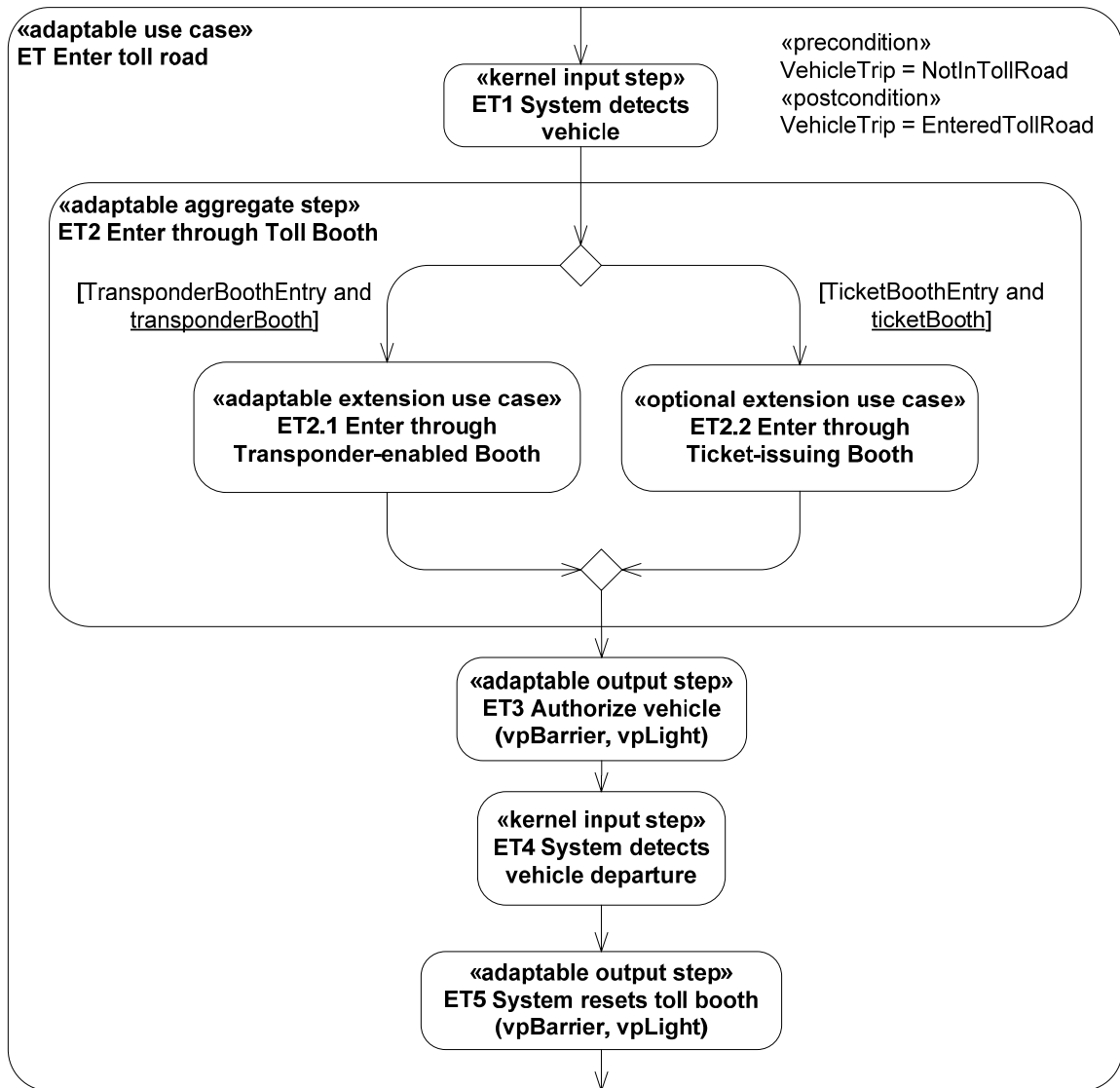


Figure 14 Modified "Enter toll road" use case activity diagram

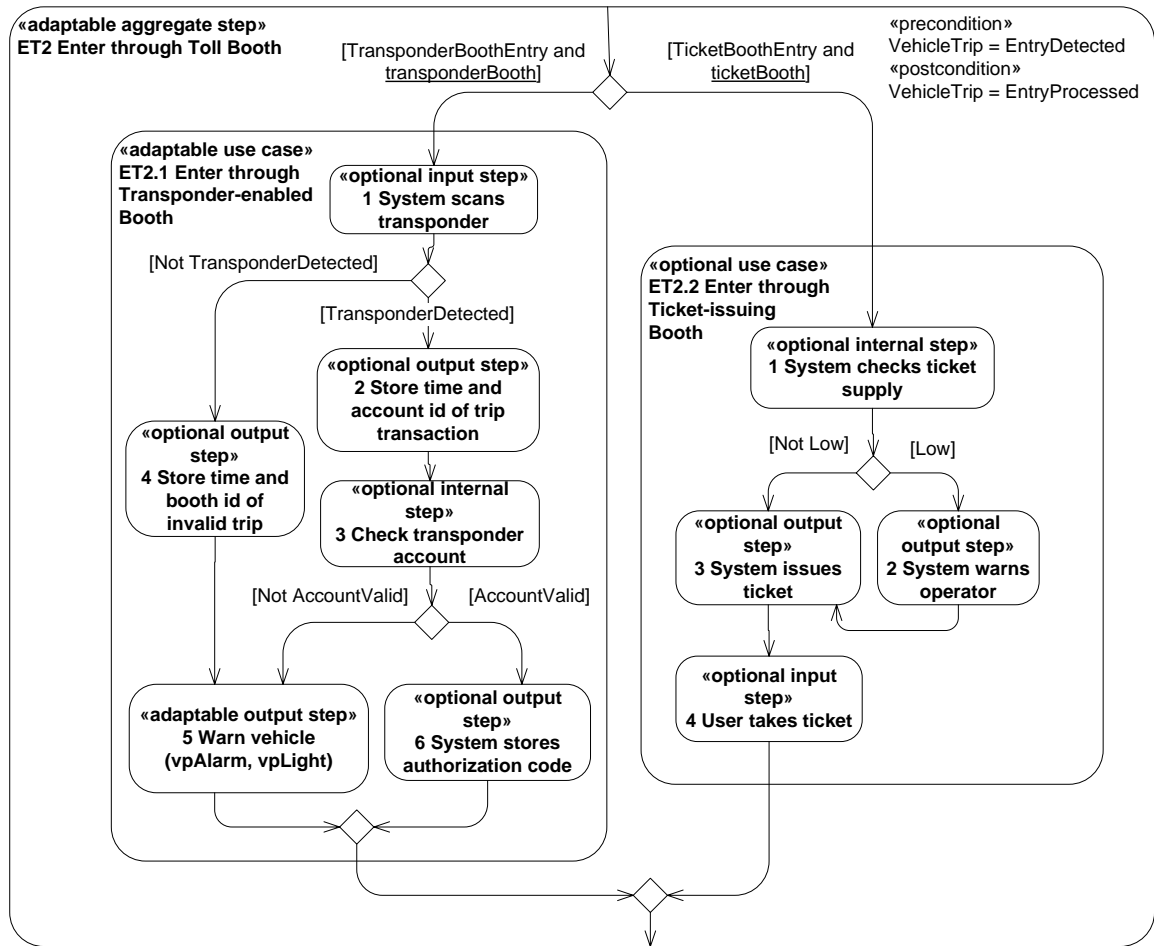


Figure 15 Activity diagram referenced by “Enter through toll booth” activity node

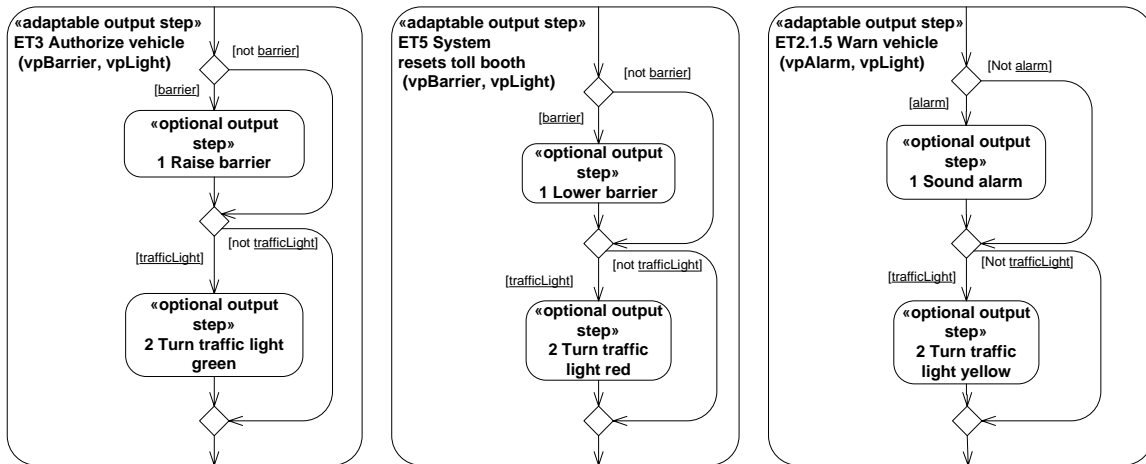


Figure 16 Sub-activity diagrams for adaptable activity nodes

5.3 Phase II: Creating Decision Tables and Test Specifications from Activity

Diagrams During SPL Engineering

In CADeT, a *decision table* is a chart that represents and organizes the relationships of conditions and activities in a use case activity diagram to the test specifications of a SPL. The decision tables in CADeT are similar to the extended use case pattern decision tables of (Binder 2002) for single applications, but differ in that the tables in CADeT can be customized for a set of applications derived from an SPL.

Decision tables are used to describe test specifications created from the activity diagrams of a SPL. Decision tables describe the conditions and sequence of activities traced from a use case activity diagram for each use case scenario. Each column in a decision table represents a simple path associated with a use case scenario. A *simple path* is a sequence of unique activities traced from an activity diagram. A simple path starts at a precondition and ends at a postcondition in the activity diagram, does not contain

repeated activity nodes, and can be concatenated with simple paths to represent sequences of use case scenarios.

Simple paths are converted to test specifications, and then added as columns to the decision tables. The precondition, feature conditions, execution conditions, postconditions, and activity nodes traversed by a simple path in the activity diagram are mapped one to one to the precondition, feature conditions, execution conditions, postconditions and test steps of a test specification in a column of the decision table. These test specifications can be concatenated with other test specifications to test sequences of use case scenarios, and can be selected, deselected, or customized depending on what features are selected for an application of the SPL.

In CADeT, decision tables also help to represent and manage the relationships of features in a feature model to the test specifications of a SPL. A feature can be associated with a test specification, which represents a unit of coarse-grained functionality, or a feature can be associated with a variation point in a test specification, which represents a unit of fine-grained functionality. Feature conditions are used to associate a feature with a test specification, and the «adaptable» stereotype is used to identify a variation point in a test specification.

The structure of the decision tables is summarized in Table 5. The test specifications are described in each numbered columns in the table, and the conditions and actions (test steps) are described in the rows. The value of a condition is entered in the intersection of the condition with the test specification. If a test step is relevant to a test specification, then an X is entered in the intersection of that test step with the test

specification, else nothing (null) is entered. The numbers 1 and 2 in the first row represent labels used to uniquely identify test specifications.

Table 5 Structure of decision table

Id	Name of use case activity diagram	1	2
	Test specification	Main scenario	First alternative scenario
Feature conditions	First feature condition	a feature condition value	a feature condition value
Preconditions	Precondition label	a precondition value	a precondition value
Execution conditions	First execution condition label	an execution condition value	an execution condition value
Actions			
1	First test step	X or null	X or null
2	Second test step	X or null	X or null
Post conditions	Postcondition label	a postcondition value	a postcondition value

The following describes the method of mapping the use case activity diagrams to decision tables in more detail:

1. Create a decision table for each use case activity diagram.
2. For each precondition in the use case activity diagram
 - Add the precondition to a row in the preconditions section of the table.
 - Label the row with the name of the precondition.
3. If the use case activity diagram is associated with a common, optional, or alternative feature in the feature to use case relationship table
 - Add a feature condition that corresponds to the common, optional, or alternative feature to the feature conditions section of the decision table
4. For each execution condition in the activity diagram

- Add the execution condition to a row in the execution conditions section of the table. Label the row with the name of the execution condition.
5. For each post condition in the use case activity diagram
 - Add the post condition to a row in the post conditions section of the table. Label the row with the name of the post condition.
 6. For each use case scenario
 - Trace one or more simple paths from the use case activity diagram for each use case scenario, beginning at the precondition of the use case activity diagram, and ending at the next precondition or postcondition reached by the path.
 7. For each simple path traced from the use case activity diagram
 - Add a column in the test specifications section of the table. Label the test specification with a unique name.
 - If the path is guarded by a feature condition, enter the value of the condition in the intersection of the feature condition row with the test specification column.
 - If the path traverses an execution condition, enter the value of the condition in the table as T (True) or F (False), else leave it blank.
 - Enter the value of the precondition and postcondition of the path in the table.

- List the activity nodes traversed by the path in the actions section of the table. Each activity node becomes a test step with the same stereotype in the actions section of the table.
 - Mark an X in the row, column intersection of the test step with the test specification.
8. Distinguish between test specifications that will be reused as is, or adapted for an application derived from the SPL. Stereotype a test specification as «adaptable» if it is impacted by a variation point, else stereotype it as «reuse as is». An «adaptable» test specification contains «adaptable» test steps which correspond to the «adaptable» activity nodes in the use case activity diagram.

5.3.1 Example of Creating Decision Tables from Activity Diagrams

Phase II of CADeT was applied to create one decision table for each use case activity diagram of the AHTS SPL. Table 6 is a decision table created from the “Enter through transponder-enabled booth” use case activity diagram of the AHTS SPL in Figure 15.

Table 6 shows a decision table created for the “Enter through transponder enabled booth use case”. The preconditions, feature conditions, preconditions, execution conditions, postconditions and actions from the activity diagram from Figure 15 were added to the decision table in Table 6. Simple paths were traced for each use case scenario of the “Enter through transponder-enabled booth” use case, and then converted into a test specification column in the decision table. The use case description of “Enter through transponder-enabled booth” in Figure 11 contains three use case scenarios: a

main scenario, an invalid transponder scenario, and an invalid account scenario. Figure 17 shows an example of a simple path (in bold) traced for the main scenario of the “Enter through transponder-enabled booth” use case, that corresponds to the “ «reuse as is» Main scenario” test specification in Table 6.

Table 6 Decision table for "Enter through transponder enabled booth" use case

ET2.1	«adaptable extension use case» Enter through transponder enabled booth	3	4	5
	Test Specifications	«reuse as is» Main scenario	«adaptable» Invalid transponder	«adaptable» Invalid account
Feature conditions	transponderBooth	T	T	T
Preconditions	VehicleTrip	Entry Detected	EntryDetected	EntryDetected
Execution conditions	TransponderBoothEntry	T	T	T
	TransponderDetected	T	F	T
	AccountValid	T		F
Actions				
ET2.1.1	«optional input step» System scans transponder (in trnspld)	X	X	X
ET2.1.2	«optional output step» System stores trip transaction (out accountId, out location)	X		X
ET2.1.3	«optional internal step» System checks transponder account	X		X
ET2.1.4	«optional output step» System times out and stores invalid trip transaction (out boothId, out location)		X	
ET2.1.5	«adaptable output step» Warn vehicle (vpAlarm, vpLight)		X	X
ET2.1.6	«optional output step» System stores authorization code (out code)	X		
Post conditions	VehicleTrip	Entry Processed	Entry Processed	Entry Processed

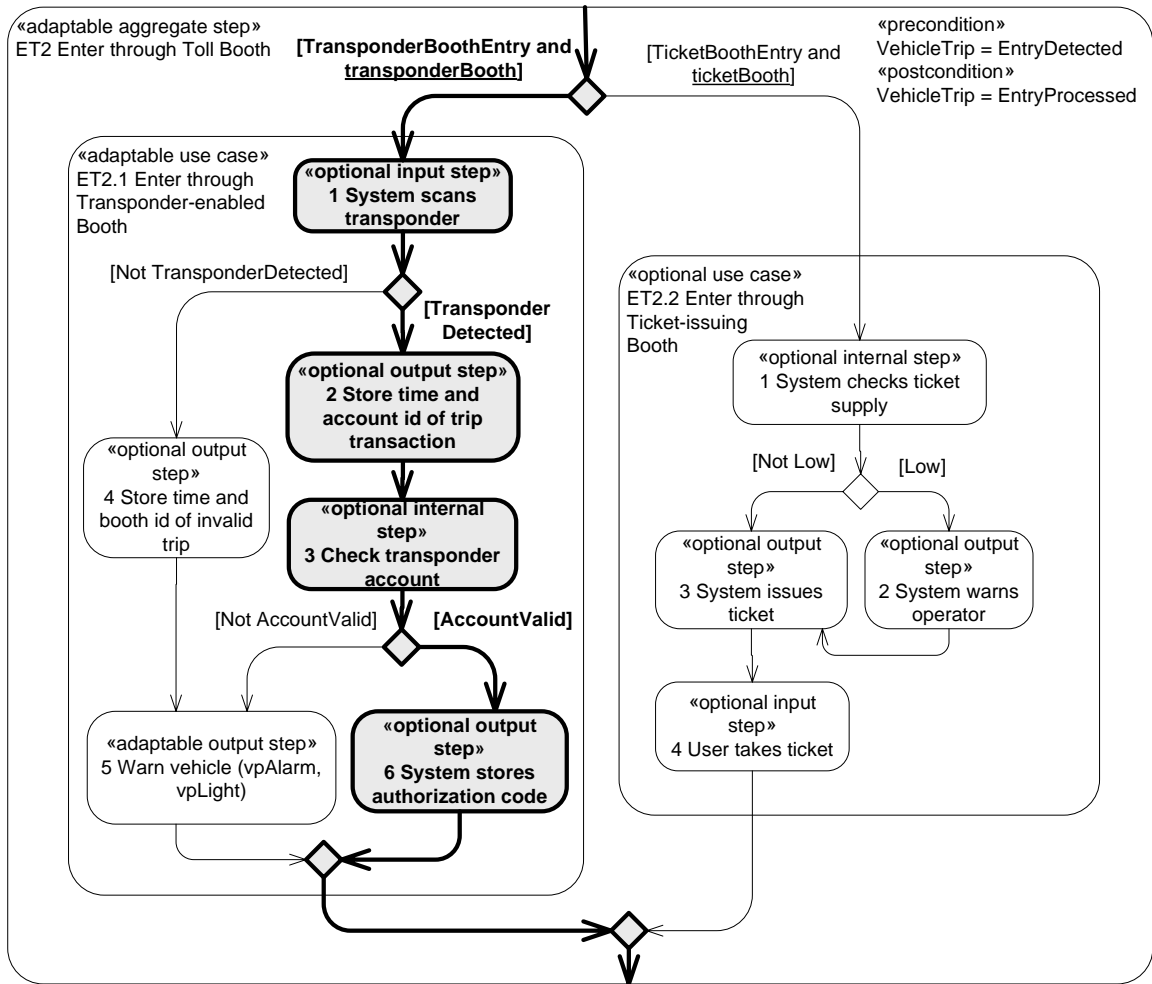


Figure 17 Example of a simple path trace

5.4 Phase III: Defining Feature-Based Test Plan

In some SPLs it may not be possible to test all possible application configurations during early system testing, because the total number of feature combinations can be large or unbounded. In these SPLs, it is not clear what features and feature combinations should be covered.

CADeT can be used to create a feature-based test plan that describes a set of application configurations to test that will cover all features, relevant feature combinations and all use case scenarios of a SPL. A *feature-based* test plan describes the features, feature combinations and use case scenarios that will be covered during testing, and the associations between features, use case scenarios, and test specifications. A *feature combination* is a selection of two or more features from the feature model for an application of the SPL. A *relevant feature combination* is a combination of features participating in a feature dependency or in a feature interaction. The feature model and the relationship of features in the feature model to the reusable test specifications are analyzed in Phase III to determine the relevant feature combinations to test.

5.4.1 Analysis of Feature Model

First, the feature model is analyzed to limit the number of application configurations to test. A *feature dependency* is a configuration constraint where the selection of one feature requires or excludes the selection of another feature. Feature dependencies, such as one feature (A) requires another feature (B), must be tested together (test combinations $\neg A \neg B$, $\neg AB$ and AB). If one feature (A) mutually includes another feature (B), then only combination AB must be tested. Feature grouping constraints, such as mutually exclusive group, also limit the number of possible feature combinations in a SPL. Parameterized features describe a range of values, which must be defined during application derivation. The boundary-value test selection criterion can be applied to select discrete values for the parameterized features of a SPL.

Table 7 describes the number of possible feature selections for feature conditions associated with the common, optional, alternative and parameterized features of a SPL. The total number of applications that can be derived from an SPL can be calculated by multiplying the number of possible feature selections for each feature condition in the SPL. For example a SPL with three feature conditions $f1, f2, f3$, where each feature condition has two possible feature selections, describes $2 \times 2 \times 2 = 8$ possible application configurations.

Table 7 Number of possible feature selections for feature conditions

Feature conditions	Feature selection	Number of feature selections
A feature condition of a common feature	T	One
A feature condition of an optional feature	$\{T, F\}$	2 (True or False)
A feature condition of an exactly-one-of feature group	$\{altern1, altern2, \dots\}$	$ altern1, altern2, \dots $, or the number of alternative features in the feature group
A feature condition of a zero-or-one-of feature group	$\{\emptyset, altern1, altern2, \dots\}$	$ \emptyset, altern1, altern2, \dots $, or the number of alternative features in the feature group plus the empty set
A feature condition of a zero-or-more-of feature group	$\{opt1, opt2, \dots\}$	$2^{ opt1, opt2, \dots }$, or two raised to the power of the number of optional features in the feature group
A feature condition of an at-least-one-of feature group	$\{opt1, opt2, \dots\}$	$2^{ opt1, opt2, \dots } - 1$, or two raised to the power of the number of optional features in the feature group minus the empty set
A feature condition of a parameterized feature	$\{param1, param2, \dots\}$	$ param1, param2, \dots $ or the number of parameter values in the set

5.4.2 Analysis of Relationships between Features and Test Specifications

Next, the relationships of features to test specifications are analyzed to reduce the number of application configurations to test, without omitting any relevant feature combinations. For each test specification (which corresponds to a use case scenario), the features that impact the scenario and its variation points must be considered. A test specification that is associated with two or more features may describe an implicit feature dependency, or a feature interaction. An *implicit feature dependency* is a feature dependency that is not described in the feature model, but is discovered late in the functional requirements of a SPL. A feature dependency in the functional requirements occurs when the selection of one feature enables or excludes functional behavior associated with the selection of another feature.

A *feature interaction* is a functional behavior that is enabled for a feature combination selected for an application derived from the SPL, but that is not enabled when any feature of the combination is selected separately. A feature interaction can cause desirable (expected) or undesirable system behavior (Zave 2004). In an activity diagram, a feature interaction is represented as an activity or data value that is enabled by a combination of two or more features, but is not enabled when any feature of the combination is selected separately.

The relationship between features and test specifications is summarized in a test specification / feature table. For each test specification, the features that affect the test specification, the adaptable test steps in a test specification, and the nature of the feature interaction (or dependency), are depicted in a feature combination function. The feature

combinations for a given test specification are described with the * or + operator, where the * operator denotes a relevant feature combination, while the + operator denotes a pair of independent features. An «adaptable» test specification with k feature conditions, where the first feature condition has n_1 possible values, the second has n_2 possible values, and so forth, can represent up to $n_1 * n_2 * \dots * n_k$ variant test specifications. If the feature conditions associated with a test specification are denoted to be independent, the number of variant test specifications can be fewer, as in $n_1 + n_2 + \dots + n_k$. The feature combination function can be simplified by grouping the relevant feature combinations and then removing duplicate terms.

If this analysis reveals implicit feature dependencies or feature interactions, then the feature model is revised to be consistent with this analysis. This procedure is limited to the analysis of feature interactions in a test specification associated with a use case scenario. Analyzing feature interactions that are associated with the execution of a sequence of test specifications (inter-use case scenario interactions) is left as an area of further research.

5.4.3 Applying a Feature-Based Coverage Criterion

Combinatorial testing techniques (Cohen, Dalal et al. 1997; Grindal 2007) can be used to reduce the number of application configurations to test. CAdET applies a combinatorial testing technique to select a set of application configurations that cover all features and relevant feature combinations of a SPL.

CAdET extends combinatorial testing techniques for single applications for SPLs by applying the notion of a configuration parameter with possible parameter values to a

feature condition with possible feature selections. The notion of constraints in (Cohen, Dalal et al. 1997) is applied to feature selection constraints in the feature model, such as a requires dependency between two features. A combinatorial test generation tool, such as Jenny (Jenkins 2005) is used to describe a representative set of application configurations that covers all features and relevant feature combinations of the SPL.

The largest number of relevant feature combinations in the feature to test specification relationship table of a SPL is used to determine a minimum n -way feature-based combinatorial coverage criterion for that SPL. An n -way combinatorial coverage criterion covers combinations of at most n features. Alternatively, the feature model can be used to determine a minimum n -way feature-based combinatorial coverage criterion for a SPL. Analyzing the feature model to determine a minimum n -way feature-based combinatorial coverage criterion is left as an area of further research.

5.4.4 Example of Defining a Feature-Based Test Plan

The feature model of the AHTS SPL in Figure 8 has a total of 16 features. Only nine of these features can be explicitly selected by an application engineer during application derivation. An application engineer can select or omit the optional Camera, Barrier, Traffic Light and Alarm features (2^4); must choose between the alternative Variable Toll Charge and Fixed Toll Charge features (2^1); and must choose at least one toll booth type from the three types of toll booths in the at-least-one-of Toll Booth Type feature group (2^3-1). Selecting a toll booth implicitly selects the devices and payment

options that correspond to the toll booth type. With these restrictions an application engineer can configure a total of $2^4 \times 2^1 \times (2^3 - 1)$, or 224 applications.

A feature combination function was defined for each test specification of the AHTS SPL. Table 8 shows an excerpt of a feature / test specification relationship table for the AHTS SPL. This table shows the feature combination functions associated with three test specifications from the “Enter through transponder enabled booth” decision table of the AHTS SPL. All three test specifications are associated with the transponderBooth feature condition, and two of these test specifications have an adaptable test step which is impacted by the alarm and trafficLight feature conditions.

Table 8 Excerpt of feature / test specification relationship table for AHTS SPL

Test specification	Feature to test specification	Adaptable test steps	Feature to adaptable test step	Feature combination function
3. «reuse as is» Enter through transponder enabled booth: Main scenario	transponder Booth = T	N/A	N/A	
4. «adaptable» Enter through transponder enabled booth: Invalid transponder	transponder Booth = T	«adaptable output step» Warn vehicle (vpAlarm, vpLight)	trafficLight={T,F} + alarm={T,F}	(transponderBooth*trafficLight) + (transponderBooth*alarm)
5. «adaptable» Enter through transponder enabled booth: Invalid account	transponder Booth = T	«adaptable output step» Warn vehicle (vpAlarm, vpLight)	trafficLight={T,F} + alarm={T,F}	(transponderBooth*trafficLight) + (transponderBooth*alarm)

The remaining test specifications in the feature / test specification relationship table of the AHTS SPL were analyzed, and the largest number of relevant feature combinations in this table is also 2 (see Table 35 in Chapter 7). The feature combination function (transponderBooth *trafficLight) + (transponderBooth* alarm) in Table 8

describes an implicit feature dependency between the traffic light and alarm lane control devices, and the transponder toll booth. The selection of a lane control device requires the selection of a toll booth. Figure 18 shows an excerpt of the feature model of the AHTS SPL, which has been updated to describe this implicit dependency.

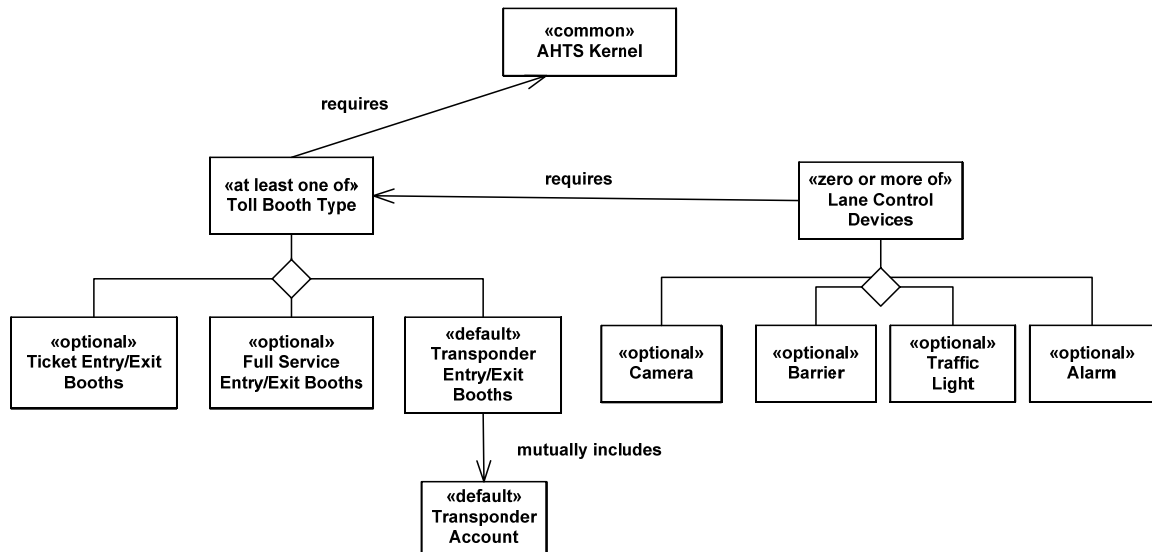


Figure 18 Excerpt of AHTS model with implicit feature dependency

The largest number of feature conditions in a relevant feature combination function in the feature / test specification relationship table of the AHTS SPL is 2. Thus, at least a 2-way, or pair-wise combinatorial testing strategy was needed to check these relevant feature combinations. Table 9 shows a feature-based combinatorial test plan that was generated to cover all valid pair-wise feature combinations in the AHTS SPL using the Jenny tool (Jenkins 2005). Eight application configurations were generated to cover all valid pair-wise feature combinations of features in the AHTS SPL.

Table 9 A feature-based combinatorial test plan for the AHTS SPL

TEST PLAN for AHTS SPL									
Features:		TS1	TS2	TS3	TS4	TS5	TS6	TS7	TS8
ticketBooth									
a.	TRUE	x		x		x			
b.	FALSE		x		x		x	x	x
fullServiceBooth									
a.	TRUE	x		x	x				
b.	FALSE		x			x	x	x	x
transponderBooth									
a.	TRUE	x	x				x	x	x
b.	FALSE			x	x	x			
camera									
a.	TRUE		x	x			x		x
b.	FALSE	x			x	x		x	
barrier									
a.	TRUE		x	x		x		x	
b.	FALSE	x			x		x		x
trafficLight									
a.	TRUE	x				x	x	x	x
b.	FALSE		x	x	x				
alarm									
a.	TRUE		x		x	x		x	x
b.	FALSE	x		x			x		
tollCharge									
a.	Variable		x	x		x			x
b.	Fixed	x			x		x	x	

5.5 Phase IV: Applying the Parameterization Variability Mechanism to Decision

Tables and Test Specifications During SPL Engineering

In CADeT, a feature can be associated with a test specification created for a use case scenario, which represents a unit of coarse-grained functionality, or a feature can be associated with a variation point value in that test specification, which represents a unit of fine-grained functionality. As in an activity diagram, a variation point in a test specification in a test specification is represented using the «adaptable» stereotype. A

variation point value corresponds to an optional or variant test step that can be inserted at the variation point location. A *variability mechanism* is a technique that enables the representation and automatic configuration of the variability in an application's requirements, models, implementation and tests. In CAdET, a *parameterization variability mechanism* is a technique that uses feature conditions to enable the automatic configuration of the variability in an application's test specifications during feature-based test derivation. Feature conditions are associated with the features of a SPL, and the values of a feature condition represent possible feature selections.

5.5.1 Tool Support for Parameterization Variability Mechanism

CAdET contains a tool suite based on Excel spreadsheets that automates the selection and configuration of test specifications for an application derived from the SPL. This tool suite uses a variability mechanism based on parameterization to select and configure the test specifications of an SPL. CAdET's tool suite contains a test generator tool and a test procedure tool. The test generator tool reads the feature selections in the feature list for an application configuration and then selects and configures the test specifications associated with these feature selections. The test procedure tool also reads the same feature selections, and then creates a test execution graph that describes the order in which the test specifications can be executed for an application derived from the SPL. In this graph, a vertex represents a test specification, and an edge represents an execution dependency between two test specifications.

5.5.2 Binding Times Supported by CADeT Tools

Associations between features and test specifications are represented and bound in a test specification during SPL engineering, while associations between features and variation point values are represented in a test specification during SPL engineering but are bound by the tools during feature-based test derivation. The meta-model in Figure 19 describes the associations between features and test specifications. A feature can be associated with either a test specification or a variation point value in the same test specification.

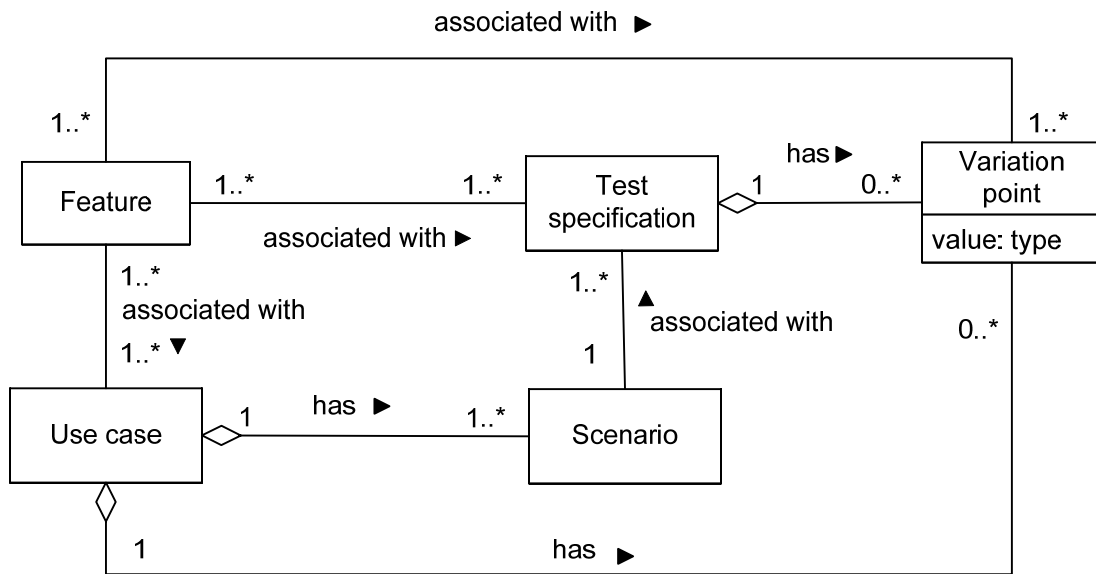


Figure 19 Association between features and test specifications

Associations between features and test specifications are bound during SPL engineering, while associations between features and variation point values are represented during SPL engineering but are bound during feature-based test derivation. Binding associations between features and test specifications during SPL engineering

allows these test specifications to be selected during feature-based test derivation for an application of the SPL. Further, binding associations between features and variation point values during feature-based test derivation allows the selected test specifications to be customized for that application.

Another option which was considered, but not used in CADeT, was to bind the variation point values during SPL engineering. However, this required a test engineer to create test specifications to cover all possible combinations of (possibly unspecified) variation point values in the SPL, and needlessly increased test development effort.

5.5.3 Description of Approach

Before the tools can be applied to automate the configuration of the test specifications, the decision tables need to be modified to describe the features and variation point values associated with each variation point in the adaptable test specifications. Discrete variation point values need to be defined and then associated with feature selections from a feature list. This section describes how a parameterization variability mechanism is applied to the decision tables and test specifications created in Phase II of CADeT.

The parameterization variability mechanism in this research uses feature conditions to associate a feature with a test specification of the SPL. The values of a feature condition variable represent possible feature selections, and selecting a value for the feature condition automatically configures the decision tables to enable the test specifications and test steps associated with that feature.

First, a feature list is created to show all feature condition variables associated with the features of a SPL, as described in Table 2 in Phase I of CADeT. This feature list is used to customize the decision tables during feature-based test derivation for an application of the SPL.

Next, the decision tables created in Phase II are modified to describe the feature conditions associated with adaptable test steps in the test specifications. Furthermore, these tables are modified to describe the optional or variant test steps associated with the adaptable test steps in the test specification. Spreadsheet functions are then added to these tables to enable the selection of optional or variant test steps for a test specification depending on the selection of a feature or a combination of features.

5.5.4 Example of Applying Parameterization Mechanism

The parameterization variability mechanism was applied to the decision tables of the AHTS SPL. The feature list in Table 3 was created to show all feature condition variables associated with the features of the AHTS SPL.

Next, the parameterization variability mechanism is applied to the decision tables of the AHTS SPL. The following describes how the parameterization mechanism is applied to the decision table of the “Enter through transponder-enabled booth” use case in Table 6, to create the modified decision table in Table 10.

The decision table of the “Enter through transponder-enabled booth” use case in Table 6 has one adaptable test step “«adaptable output step» Warn vehicle (vpAlarm, vpLight)”. The feature conditions *alarm* and *trafficLight*, which impact the variation

points in this adaptable test step (as shown in Table 8 in Phase III), were added to the feature conditions section of the decision table.

This decision table contains two adaptable test specifications called “Invalid transponder” and “Invalid account” which include this adaptable test step. The spreadsheet functions $Fc(alarm)=\{T,F\}$ and $Fc(trafficLight)=\{T,F\}$ were entered in the intersection of the *alarm* and *trafficLight* feature conditions with each test specification, to display the feature selections associated with an adaptable test step.

Next, the adaptable test step was replaced with optional test steps associated with the feature selections $alarm=T$ and $trafficLight=T$. The “«optional output step» Sound alarm” is associated with feature selection $alarm=T$ and the “«optional output step» Turn traffic light yellow” is associated with the feature selection $trafficLight=T$. The spreadsheet function $Fs(alarm=T)=\{X, Null\}$ enables “«optional output step» Sound alarm” when the *alarm* feature is selected, and disables this step when the *alarm* feature is not selected for an application. Likewise, the spreadsheet function $Fs(trafficLight=T)=\{X, Null\}$ enables or disables “«optional output step» Turn traffic light yellow” depending on whether the corresponding feature is selected for an application derived from the AHTS SPL.

Table 10 Example of parameterization mechanism applied to decision table

ET2.1	Enter through transponder enabled booth	3	4	5
	Test Specifications	«reuse as is» Main scenario	«adaptable» Invalid transponder	«adaptable» Invalid account
Feature conditions	transponderBooth	T	T	T
	alarm		$Fc(alarm) = \{T, F\}$	$Fc(alarm) = \{T, F\}$
	trafficLight		$Fc(trafficLight) = \{T, F\}$	$Fc(trafficLight) = \{T, F\}$
Pre-conditions	VehicleTrip	Entry detected	Entry detected	Entry detected
Execution conditions	TransponderBoothEntry	T	T	T
	TransponderDetected	T	F	T
	AccountValid	T		F
Actions				
1	«optional input step» System scans transponder (in trnspId)	X	X	X
2	«optional output step» System stores trip transaction (out accountId, out location)	X		X
3	«optional internal step» System checks transponder account	X		X
4	«optional output step» System times out and stores invalid trip transaction (out boothId, out location)		X	
5.1	«optional output step» Sound alarm		$Fs(alarm=T) = \{X, Null\}$	$Fs(alarm=T) = \{X, Null\}$
5.2	«optional output step» Turn traffic light yellow		$Fs(trafficLight=T) = \{X, Null\}$	$Fs(trafficLight=T) = \{X, Null\}$

6	«optional output step» System stores authorization code (out code)	X		
Post conditions	VehicleTrip	Entry Processed	Entry Processed	EntryProcessed

5.6 Customizing Test Specifications During Application Engineering

The remaining phases of CADeT are applied during application engineering to customize the test specifications for an application derived from the SPL:

- Phase V: Customize the decision tables and test specifications using the parameterization variability mechanism
- Phase VI: Select test data for an application
- Phase VII: Test application

The activities that correspond to these phases, and the artifacts created by these activities are shown in Figure 20. The CADeT tools (described in Phase IV) are used to automate the generation of the test specification, test procedure and system test documents during feature-based test derivation for application of the SPL

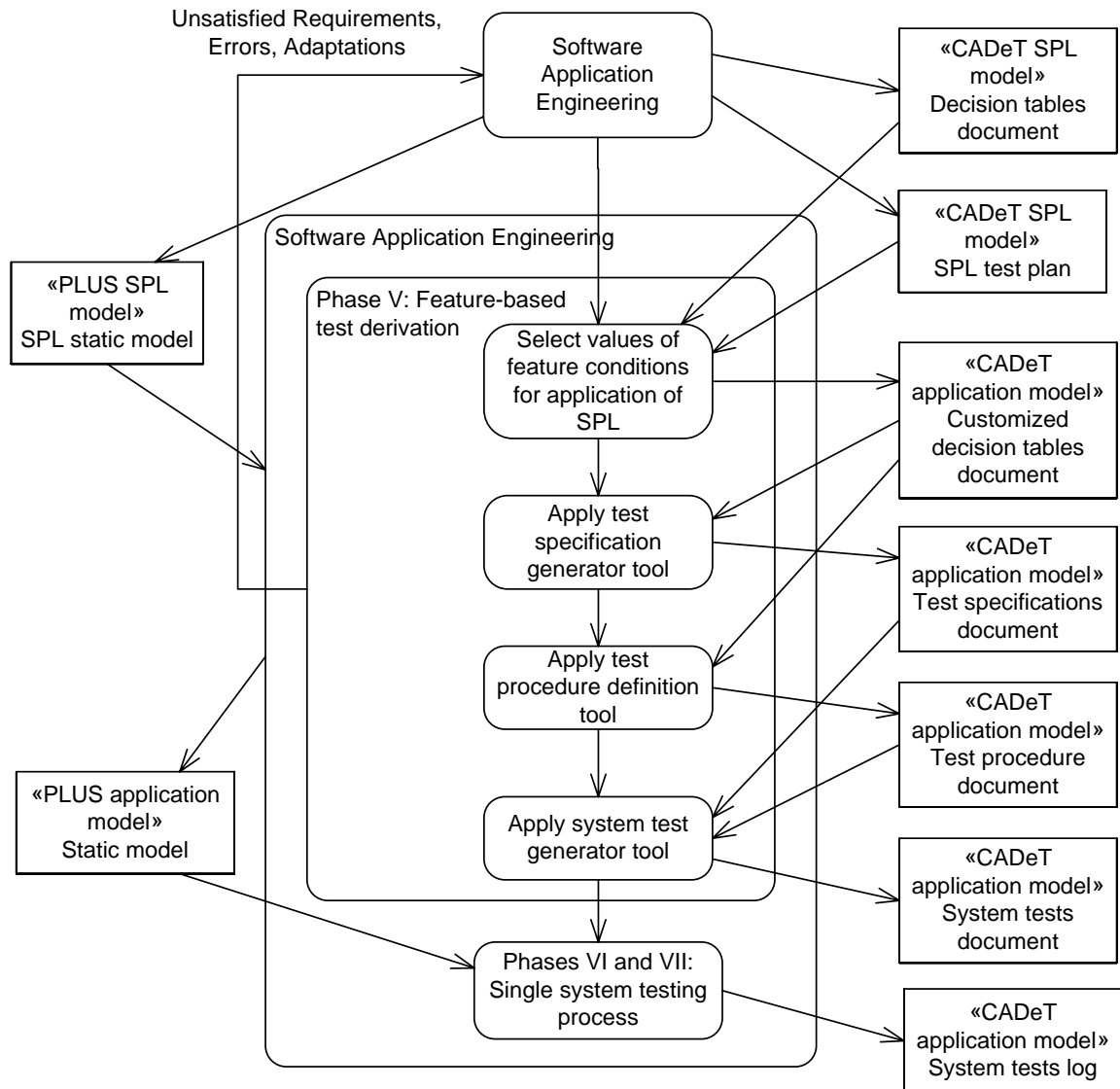


Figure 20 Incorporating CAdET within an application engineering process

5.7 Phase V: Customizing the Decision Tables and Test Specifications Using the Parameterization Variability Mechanism

The parameterization variability mechanism uses feature conditions to associate features with the test specifications of a SPL. Selecting a value for a feature condition automatically configures the decision tables to enable the test specifications and test steps

associated with that feature. Tools are used during feature-based test derivation in Phase V to automate the configuration of the decision tables, and to generate test specification, test procedure and system tests documents. This process is described in more detail in the following sections, and then illustrated with examples from the AHTS SPL.

5.7.1 Selecting Values of Feature Conditions

First, the feature selection values in the feature list of the SPL are set to correspond to the feature selections of an application derived from the SPL. The feature list, decision tables and test specifications of a SPL are stored in a spreadsheet document, which is created in Phase II, and then updated in Phase IV with the parameterization variability mechanism. This document contains functions that automatically configure the decision tables based on the features selected for the application.

5.7.2 Example of Selecting Values of Feature Conditions

The following example describes the customization process for TS1, an application from the test plan of the AHTS SPL in Table 9. First, the feature list from Table 3 is customized for application TS1, as shown in Table 11. The “Feature selections for TS1” column in Table 11 shows the feature selections associated with TS1 for each feature condition in the AHTS SPL.

Table 11 Feature selections for TS1

Feature condition	Feature Selections
AHTSKernel	T
ticketBooth	T
fullServiceBooth	T
transponderBooth	T
camera	F
barrier	F
trafficLight	T
alarm	F
tollCharge	fixed
ticketDispenser	T
ticketReader	T
creditCardReader	T
cashReader	T
operator	T
transponderAccount	T

Setting the feature conditions in the feature list automatically customizes the SPL decision tables for TS1. Table 12 is an example of a customized decision table for the “Enter through transponder enabled booth” use case. The Transponder Booth and Traffic Light features have been selected for TS1, so the values of the feature conditions associated with these features has been set to ‘T’ in the decision table. The Alarm feature has not been selected for TS1, so the value of the Alarm feature condition has been set to ‘F’ in the decision table. Setting the Traffic Light feature condition to ‘T’ enables test step “5.2 Turn traffic light yellow” in the “Invalid transponder” and “Invalid account” test specifications. Setting the Alarm feature condition to ‘F’ disables test step “5.1 Sound alarm” in these test specifications.

Table 12 Example of a customized decision table for TS1

ET2.1	Enter through transponder enabled booth	1	2	3
	Test Specifications	«reuse as is» Main scenario	«adaptable» Invalid transponder	«adaptable» Invalid account
Feature conditions	transponderBooth	T	T	T
	alarm		F	F
	trafficLight		T	T
Preconditions	VehicleTrip	Entry Detected	EntryDetected	Entry Detected
Execution conditions	TransponderBoothEntry	T	T	T
	TransponderDetected	T	F	T
	AccountValid	T		F
Actions				
1	«optional input step» System scans transponder (in trnspld)	X	X	X
2	«optional output step» System stores trip transaction (out accountId, out location)	X		X
3	«optional internal step» System checks transponder account	X		X
4	«optional output step» System times out and stores invalid trip transaction (out boothId, out location)		X	
5.1	«optional output step» Sound alarm			
5.2	«optional output step» Turn traffic light yellow		X	X
6	«optional output step» System stores authorization code (out code)	X		
Postconditions	VehicleTrip	Entry Processed	Entry Processed	Entry Processed

5.7.3 Applying Test Specification Generator Tool

Next, the test specification generator tool is used to generate the test specifications document from the customized decision tables. This tool uses feature-based test derivation to select test specifications in the decision tables that correspond to features selected for an application of the SPL, and then write the selected test specifications to a test specifications document for that application.

5.7.4 Example of Applying Test Specification Generator Tool

Only test specifications that are relevant to an application are included in the test specifications document for that application. Since application TS1 contains the Transponder Booth feature (see Table 11), the test specifications document for TS1 includes all test specifications from the customized “Enter through transponder enabled booth” decision table in Table 12.

Table 13 shows an example of a test specification generated for application TS1. The “Invalid transponder” column from the customized “Enter through transponder enabled booth” decision table in Table 12 has been reformatted to show only the conditions and actions that are relevant to that test specification.

Table 13 Example of a test specification generated for TS1

Use case name	Enter through transponder enabled booth	
Test specification name	«adaptable» Invalid transponder	
Feature conditions		
	transponderBooth	T
	alarm	F
	trafficLight	T
Preconditions		

	VehicleTrip	Entry detected
Execution conditions		
	TransponderBoothEntry	T
	TransponderDetected	F
Actions		
	«optional input step» System scans transponder (in trnspld)	
	«optional output step» System times out and stores invalid trip transaction (out boothId, out location)	
	«optional output step» Turn traffic light yellow	
Post conditions		
	VehicleTrip	Entry processed

5.7.5 Applying Test Procedure Definition Tool

Next, the test procedure definition tool is used to create a test procedure document for the application. The test procedure document describes a collection of system tests, where a *system test* describes the order in which a sequence of test cases will be executed for an application derived from the SPL. A *test case* is an instance of a reusable test specification that describes the input and output data values selected to satisfy the predicates in the test specification.

This tool applies a graph building algorithm to construct a test order graph from the customized decision tables during feature-based test derivation. The test order graph sorts the test specifications by pre and post conditions, thus constraining the order in which these specifications can be executed for the application.

The pseudo code for the graph building algorithm is shown in Figure 21. A test specification is included in the graph if the values of its feature conditions match the values of the feature selections for the application derived from the SPL. A test specification in a decision table is mapped to a vertex in the test order graph, and an execution dependency between test specifications is mapped to an edge in the graph. An execution dependency is a relationship between two test specifications, where the precondition of one test specification matches the postcondition of another test specification.

```

var IncludeTest As Boolean
type TestSpecType As {
    TestId As String
    Pre As String
    Post As String
    hasEdge As Integer 'True if the pre of this spec matches the post of another spec, else false'
}

var testSpecification As TestSpecType
var max As Integer 'The maximum number of test specifications'

var AM [0.. max, 0..max] of testSpecification 'AM is the adjacency matrix that encodes the graph'

BuildGraph() {
    Initialize AM
    FOR each decision table in the SPL
        FOR each test specification in the decision table
            IncludeTest = True
            FOR each feature condition in test specification
                Look up the feature selection for this feature condition
                in the Feature list
                IF the feature selection in the feature list does not
                match the feature condition value set for this test
                specification THEN
                    IncludeTest = False
                END IF
            END FOR
            IF IncludeTest = True THEN
                Add testSpecification to a row and column in AM
            END IF
        END FOR
    END FOR
    FOR each testSpec in row of AM
        FOR each adjTest in column of AM
            IF testSpec.Post matches adjTest.Pre THEN
                AM [row, column].testSpecification.hasEdge = True
            END IF
        END FOR
    END FOR
}

```

Figure 21 Graph building algorithm

After the graph is created, a test engineer uses an interface provided by the tool to create system tests, save these tests to a test procedure document, and view the percentage of test specifications covered by the test procedure.

The user interface of the test procedure definition tool is shown in Figure 22. A test engineer initializes the tool with the name and initial value of a system state variable defined for the application. The tool displays all test specifications in the test suite that have a precondition that matches this initial state. Then, the test engineer uses this tool to create system tests for the application. The test engineer creates a system test by clicking the “Begin Test” button, and then double-clicks one of the available test specifications to add it as a test case to the system test. The tool updates the current system state to match the post condition of the selected test specification, and then displays only those test specifications that have a precondition that matches the current system state. The test engineer continues adding test cases to the system test until he or she decides to end the system test. Then, the test engineer saves the system test to the test procedure document, which updates the percentage of test specifications covered by the test procedure. The test engineer continues creating system tests until all test specifications in the application’s test suite have been covered at least once (percent coverage = 100%).

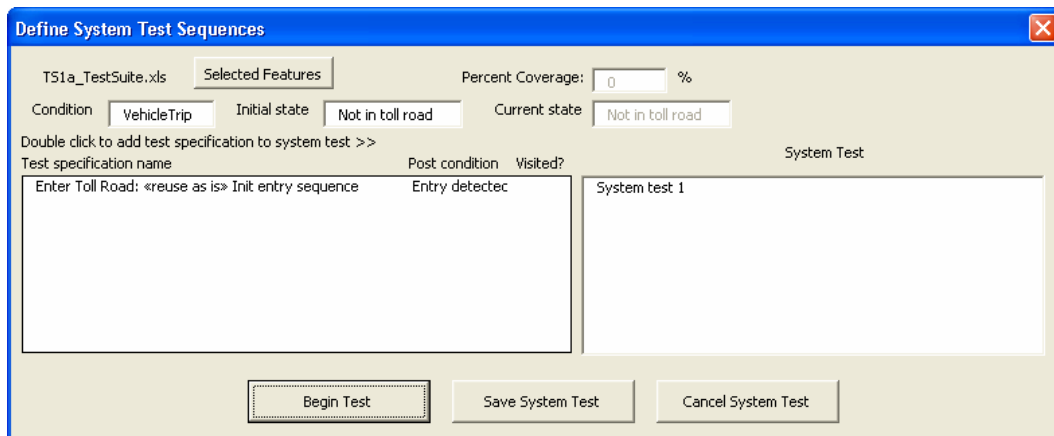


Figure 22 System test definition tool

However, it is possible that errors in the description of pre and post conditions of can prevent a test specification from being added to a system test. In this case, the use case requirements and activity diagrams need to be checked for errors and then updated.

Covering all test specifications in an application's test suite does not imply that all possible execution orderings of the test specifications have been covered. Stronger coverage criteria can be applied to the customized graph, such as covering all edges, or covering all preconditions of the test specifications in the graph. Applying stronger coverage criteria improves the effectiveness of a test procedure, but also increases the effort needed to execute the procedure. Even with a small number of test specifications, the number of possible paths in a test execution sequence graph can be large.

5.7.6 Example of Applying Test Procedure Definition Tool

The test procedure definition tool shown in Figure 22 was used to create a test procedure document for application TS1. A test order graph was generated for TS1, sorting the test specifications of the AHTS SPL according to the pre and post conditions defined in the use case and system level activity diagrams of the AHTS SPL in Phase I (see Figure 13) and then mapped to test specifications in Phase II. Then paths were traced from this graph to define system tests for the test procedure document. Figure 23 is an excerpt of the test order graph generated for TS1, which shows the execution dependencies between some of the test specifications selected for TS1. The "Enter toll Road: Init entry" test specification must be executed first, followed by one of the test specifications from the "Enter through transponder enabled booth" or "Enter through

ticket issuing booth” use cases. The feature conditions are underlined in the graph to distinguish them from the execution conditions.

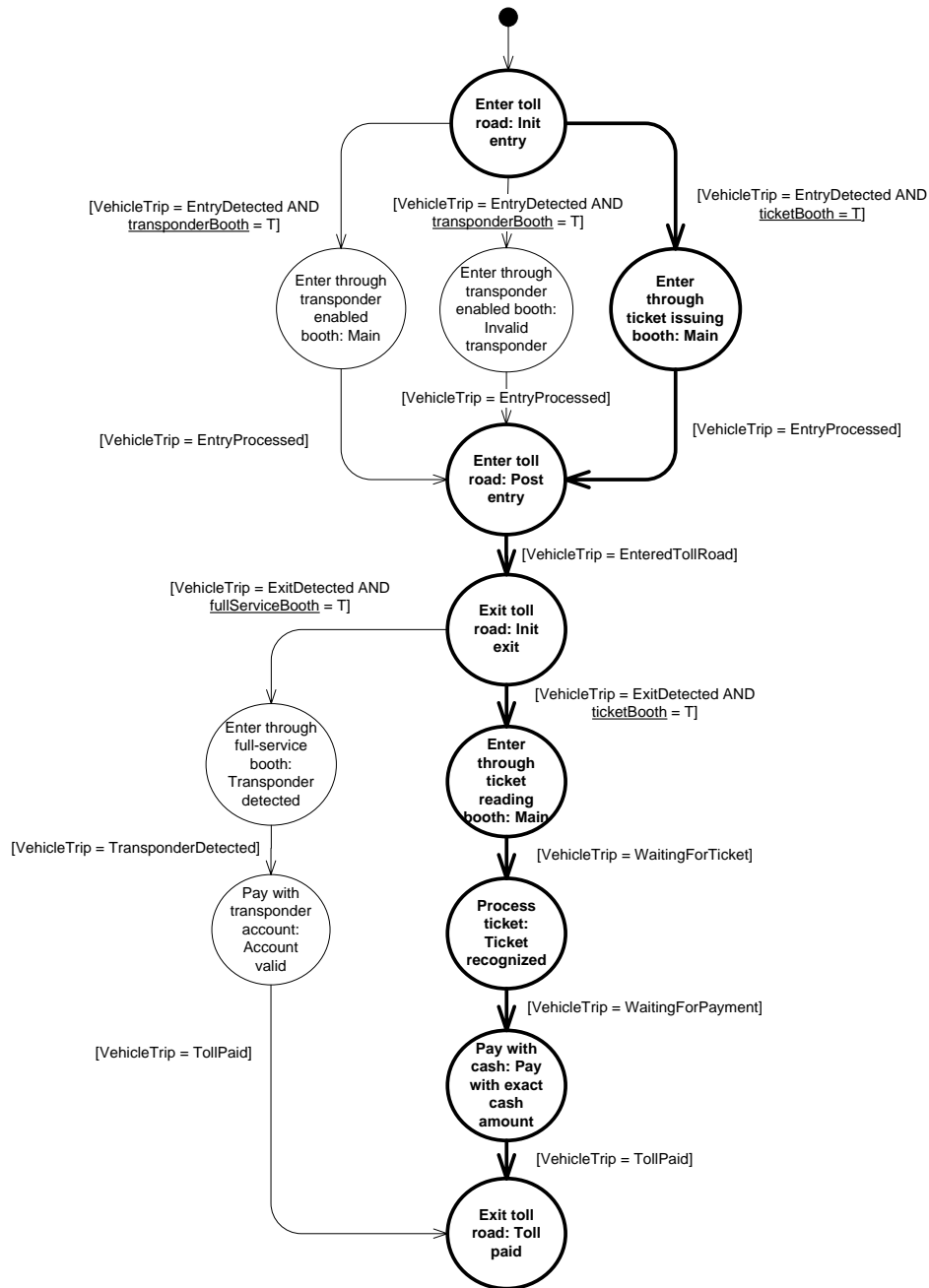


Figure 23 Excerpt of test order graph for TS1

Figure 23 also shows a path traced from this graph to define a system test for TS1 (shown in bold). This path describes a system test for a vehicle trip where a driver enters the toll road through a ticket issuing entry booth, exits the toll road through a ticket reading exit booth, and then pays the toll using an exact cash amount. This path corresponds to “System test 1” in an excerpt from the test procedure document of TS1 in Table 14.

Table 14 Example of system tests from test procedure document of TS1

System test 1
Enter Toll Road: «reuse as is» Init entry sequence
Enter through ticket-issuing booth: «reuse as is» Main scenario
Enter Toll Road: «adaptable» Post entry sequence
Exit Toll Road: «reuse as is» Init exit sequence
Exit through ticket reading booth: «reuse as is» Main
Process ticket: «adaptable» Ticket recognized
Pay with Cash: «reuse as is» Pay with exact cash amount
Exit Toll Road: «adaptable» Toll paid
System test 2
Enter Toll Road: «reuse as is» Init entry sequence
Enter through transponder enabled booth: «reuse as is» Main scenario
Enter Toll Road: «adaptable» Post entry sequence
Exit Toll Road: «reuse as is» Init exit sequence
Exit through full-service booth: «reuse as is» Transponder detected
Pay with transponder account: «adaptable» Account valid
Exit Toll Road: «adaptable» Toll paid

5.7.7 Applying System Test Generator Tool

Next, the system test generator tool is used to generate a system tests document from the test specifications and test procedure documents of an application. The system tests document describes the details of the test cases referenced by a test procedure, such as the inputs and outputs selected to satisfy test predicates, and the test results. Before generating a system tests document, the test specifications of a SPL can be refined to

describe the actual input and environment variables used by an SPL implementation. The system tests document is used in Phase VI to select test data for an application, and in Phase VII to test the application.

5.7.8 Example of Applying System Test Generator Tool

The system test generator tool was applied to generate a system tests document for application TS1. Table 15 shows an excerpt of the system tests document for TS1. Besides describing the conditions and actions of each test case in a system test, the system test document has columns to enter selected inputs, expected outputs and test results.

Table 15 Excerpt of system tests document for TS1

System test 2		Inputs / Outputs	Pass / Fail
Test specification name	Enter Toll Road: «reuse as is» Init entry sequence		
Feature conditions:			
AHTSKernel	T		
Preconditions:			
VehicleTrip	Not in Toll Road		
Execution conditions:			
Actions:			
	«kernel input step» System detects vehicle		
	«adaptable aggregate step» Invoke “Enter through toll booth”		
Postconditions:			
VehicleTrip	Entry detected		
Test specification name	Enter through transponder enabled booth: «reuse as is» Main scenario		
Feature conditions:			
transponderBooth	T		
Preconditions:			
VehicleTrip	Entry detected		
Execution conditions:			
TransponderBoothEntry	T		
TransponderDetected	T		
AccountValid	T		
Actions:			
	«optional input step» System scans transponder (in trnspld)		
	«optional output step» System stores trip transaction (out accountId, out location)		
	«optional internal step» System checks transponder account		
	«optional output step» System stores authorization code (out code)		
Postconditions:			
VehicleTrip	Entry processed		

5.8 Phase VI: Selecting Input Data

In Phase VI the test engineer selects input data for the database and system tests of the application. This process is similar to the process of selecting input data for the database and system tests of a single application, with also the same limitations. The problem of determining whether a particular input data exists to satisfy a test requirement is undecidable in general (DeMillo and Offutt 1991; Allen, Wang et al. 1994). The instructions for Phase VI are described in the following sections.

5.8.1 Creating Database Structure from Static Entity Model of Application

Transaction-based software applications usually interface with a database, which needs to be created and initialized prior to testing the application. A database structure can be created from the static entity class model of an application. A static entity class model for an application is derived from the static entity class model of a SPL created using the PLUS method (Gomaa 2005) during SPL engineering. The feature to class dependency table in PLUS (Gomaa 2005) describes the relationship between the features and classes of a SPL. The SPL class model is customized for an application by removing optional and alternative classes that are not associated with the features selected for the application.

To create the database structure, classes, constraints and associations in the static class model are mapped to tables and constraints in the database, as described in (Hoffer, George et al. 2005). Table 16 summarizes the relationship between the elements in the static model and the database structure. One table is created for each class in the static model. The class attributes map to fields in the table, and constraint on class attributes

map to constraints on field values. A class instance, or object, is represented by rows in a database table. A constraint on an association maps to a constraint on the database table rows and keys of the related tables.

Table 16 Relationship between static model notation and database structure

Static model	Database structure
Class	Table
Class attribute	Field in table
Constraint on class attribute (type, range of values)	Constraint on field value
Class instance (object)	Row in table
Constraint on association (cardinality)	Constraint on table rows and table keys

5.8.2 Selecting Input Data to Satisfy Database Constraints

Next, the test engineer initializes the database tables by selecting input data for the database tables that satisfy database constraints. These database constraints may be comprised of three types of constraints: The maximum number of table rows; the relationship between a table key and another table's foreign key; and the type and range of table field values. The constraint on maximum number of table rows corresponds to the constraint in the cardinality of a related class association; the relationship between a table key and another table's foreign key corresponds to the type of association between two classes (e.g. unary, binary) and the cardinality of the association (e.g. 1 to 1..*, 1 to 1); and the type and range of table field values corresponds to a constraint on a class attribute.

5.8.3 Example of Selecting Input Data for Database

A database structure was created from the static entity class model of TS1, an application derived from the AHTS SPL test plan in Table 9. Then, input data was selected to satisfy the constraints in the database and in the system tests of TS1.

An excerpt of the static model for TS1 is shown in Figure 24, and an excerpt of the database created from some classes in this static model is shown in Table 17. The TollStation, TollBooth, Transponder and TransponderAccount tables in the database correspond to the classes with the same name in the static model of Figure 24. The primary keys in each table are underlined and the foreign keys are italicized. Some of the associations in the static model of Figure 24 have been mapped to database constraint rules in Table 17. For example, the one-to-many association between the TollStation and TollBooth classes in Figure 24 corresponds to a constraint on the relationship between the TollStation and TollBooth objects: “A toll station object has one or more toll booth objects.” This constraint is described in Rule 1 in Table 17 as “For each unique TollStation.stationid there exists one or more rows in TollBooth where TollStation.stationId = TollBooth.*stationId*”.

After mapping the class model to a database structure, the database tables were initialized to satisfy the constraints in the database tables, as shown in Table 17. For example, the TollBooth table has two entries with ids B1 and B2. Each of these entries references entry T0001 in the TollStation table.

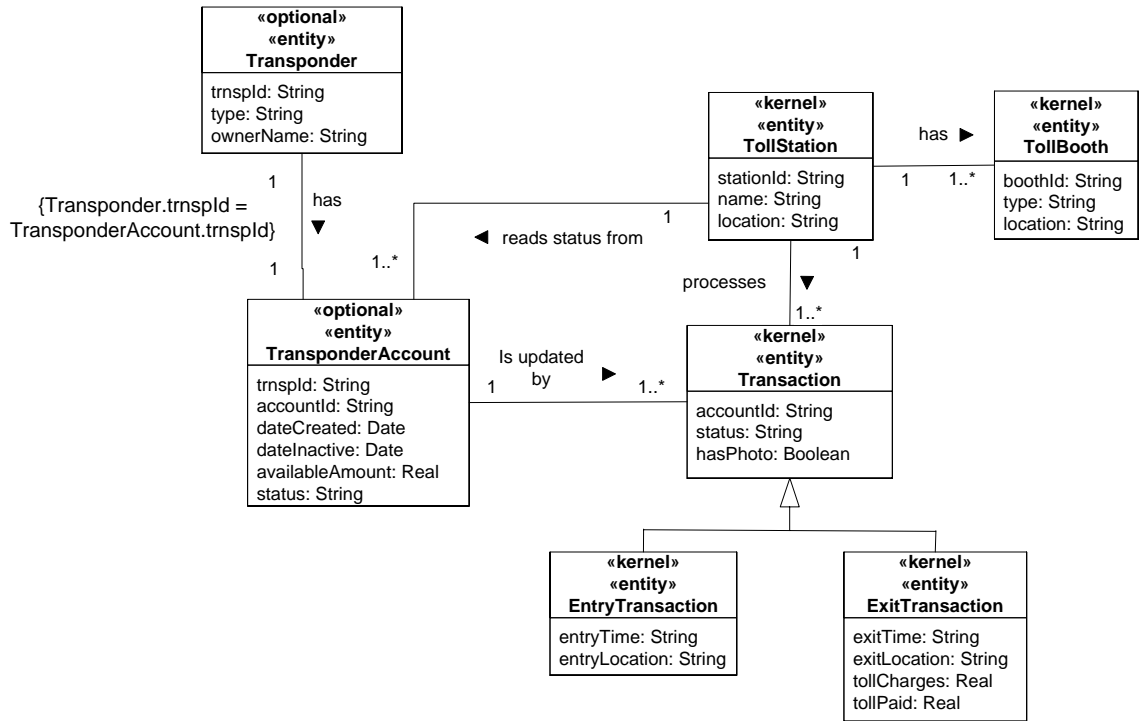


Figure 24 Excerpt of static model for AHTS SPL

Table 17 Example of input data selected for database of TS1

Database tables

TollStation		
<u>stationId</u>: String	name: String	location: String
T0001	TollStation1	Dulles, VA

Rule1: For each unique TollStation.stationId there exists one or more rows in TollBooth where TollStation.stationId = TollBooth.stationId

TollBooth		
<u>boothId</u>: String	type: String	<i>stationId: String</i>
B1	Ticket Entry Booth	T0001
B2	Transponder Entry Booth	T0001

Transponder		
<u>trnspId</u>: String	type: String	ownerName: String
TR1	Interior	Joe Shmoe

Rule2: For each unique Transponder.trnspId there exists exactly one row in TransponderAccount where Transponder.trnspId = TransponderAccount.trnspId

TransponderAccount			
<i>trnspId: String</i>	<u>accountId: String</u>	availableAmount: Real	status: String
TR1	112233	\$30.00	Active

5.8.4 Selecting Input Data to Satisfy Execution Conditions in System Tests

Most transaction-based software applications also provide a user interface, which allows a human actor to provide inputs to an application, and also to observe the outputs of the application. The system tests document provided by CADeT can be used by a test engineer to select and enter input data values for user interface commands, and derive expected output values for user interface display actions.

A system test in a system tests document describes a sequence of test cases. Each test case has one or more execution conditions that constrain the values of input parameters in one or more «input test step»s, or commands invoked in the user interface

of the application. Further, each test case has one or more «output test step»s, or user interface display actions, that may contain output parameter values. A test engineer selects inputs for the input parameters in the «input test step»s that satisfy the execution conditions in the test specification. Then, the test engineer derives the expected values of the output parameters and database state of subsequent output test steps.

Table 18 describes the conventions used in CADeT to represent the selections of input parameter values, output parameter values, and database attribute values. Some of these conventions, such as using a question mark to indicate the selection of a new input, are adapted from the Z notation.(Diller 1994).

Table 18 Conventions for representing the selection of variable values

Variable	Description	Convention
input parameter	Select new actor input	variableName?
output parameter	Derive expected output	variableName!
database attribute	Match any database attribute value	*databaseAttribute
input parameter, output parameter or database attribute	Reference a previously selected variable value	variableName or databaseAttribute

It is possible that no input data exists that can satisfy that particular combination of execution conditions. This may occur if the input data selected for the database is incomplete, or if no input data exists that can satisfy the combination of execution conditions in the system test. In the first case, test data can be added to, changed, or removed from the database. In the second case, the conditions in the test specifications need to be compared with the original requirements to detect possible inconsistencies or errors.

5.8.5 Example of Selecting Input Data for System Tests

After initializing the database, input data was selected to satisfy the execution conditions of the test cases in the system tests of TS1. Table 19 shows an example of input data selected for the test case “Enter through transponder enabled booth: «reuse as is» Main scenario” in the system test of TS1. The execution condition “AccountValid” constrains the values of the input parameter trnspId in “«optional input step» System scans transponder (in trnspId)”, the “trnspId” field in the Transponder table, and the related “status” field in the TransponderAccount table. The input value “trnspId=TR1” was selected to satisfy the execution condition “AccountValid: (trnspId = Transponder.trnspId) and (TransponderAccount.status = Valid)” in this test case.

Table 19 Example of input data selected for a system test

System test 2		Inputs / Outputs	Pass / Fail
Test specification name	Enter Toll Road: «reuse as is» Init entry sequence		
Feature conditions:			
AHTSKernel	T		
Preconditions:			
VehicleTrip	Not in Toll Road		
Execution conditions:			
Actions:			
	«kernel input step» System detects vehicle (in vehicleId)	vehicleId = V1	
	«adaptable aggregate step» Invoke “Enter through toll booth”		
Postconditions:			
VehicleTrip	EntryDetected		
Test specification name	Enter through transponder enabled booth: «reuse as is» Main scenario		
Feature conditions:			
transponderBooth	T		
Preconditions:			
VehicleTrip	EntryDetected		
Execution conditions:			
TransponderBoothEntry	T		
TransponderDetected	T		
AccountValid: (trnspId = Transponder.trnspId) and (TransponderAccount.status = Valid)	T		
Actions:			
	«optional input step» System scans transponder (in trnspId)	trnspId = TR1	
	«optional output step» System stores trip transaction (out accountId, out location)	location = B2, accountId = 112233	
	«optional internal step» System checks transponder account		
	«optional output step» System stores authorization code (out code)	code = ValidEntry	
Postconditions:			
VehicleTrip	EntryProcessed		

5.9 Phase VII: Testing Application

An executable of the application is derived from the SPL during feature-based application derivation. The test engineer deploys this executable, and then follows the system tests document to run the executable against the selected input data. This process is similar to the manual testing process for a single application. The test engineer enters the input data in the user interface of the executable, observes the actual output, compares the actual output with the expected output, and then logs the test results. A “Pass”, “Fail”, or “Inconclusive” test result is logged for each input and output test step in each test case in the system tests document.

A “Pass” result means that the observed outputs matched the expected outputs; a “Fail” result means that the observed outputs did not match the expected outputs, which may be the result of a fault being executed in the implementation; and an “Inconclusive” result means that the test engineer could not determine with certainty whether the test step passed or failed. The test results of each test step are aggregated to describe a test result for a test case, and the results of each test case are aggregated to describe a test result for a system test. A test case passes if all of its test steps have a “Pass” result, and a system test passes if all of its test cases have a “Pass” result. Else, if any test step in a test case of a system test has a “Fail” result, then the system test has a Fail result.

The example in Table 20 shows how the Pass or Fail status might be entered during a test run of “System test 2” for a fictional AHTS simulator configured for application TS1. The simulator is initialized to model a vehicle V1 entering a toll road

through a transponder enabled booth with a transponder id of TR1. Then, System test 2 is executed against the simulator. If the actual outputs match the expected outputs a Pass status is entered in the intersection of each output step with the Pass / Fail column, as shown in Table 20.

5.10 Summary

This chapter has described the CADeT approach and illustrated it on examples from an AHTS SPL. CADeT defines and applies a feature-based test coverage criterion together with a use case-based coverage criterion to a SPL: Cover all use case scenarios, all features and all relevant feature combinations of a SPL.

With the CADeT approach, use case descriptions are converted to activity diagrams, and reusable test specifications are created from these activity diagrams for each use case scenario. Then, the feature model, and the relationships of features to test specifications are analyzed to determine feature combinations relevant for a SPL. A test coverage criterion is applied to select a set of applications to cover these feature combinations. Next, the SPL test specifications are customized during feature-based test derivation for each of these applications.

Chapter 6 describes CADeT-SoC, an extension of the CADeT approach that uses a separation of concerns variability mechanism to customize the SPL test specifications. Chapter 7 describes the validation of CADeT on an AHTS SPL and a Banking System SPL. The application of CADeT and CADeT-SoC to the Banking System SPL is described in more detail in Appendix A.

Table 20 Example of pass / fail status in a test specification

System test 2		Inputs / Outputs	Pass / Fail
Test specification name	Enter Toll Road: «reuse as is» Init entry sequence		
Feature conditions:			
AHTSKernel	T		
Preconditions:			
VehicleTrip	Not in Toll Road		
Execution conditions:			
Actions:			
	«kernel input step» System detects vehicle (in vehicleId)	vehicleId = V1	Pass
	«adaptable aggregate step» Invoke “Enter through toll booth”		Pass
Postconditions:			
VehicleTrip	EntryDetected		
Test specification name	Enter through transponder enabled booth: «reuse as is» Main scenario		
Feature conditions:			
transponderBooth	T		
Preconditions:			
VehicleTrip	EntryDetected		
Execution conditions:			
TransponderBoothEntry	T		
TransponderDetected	T		
AccountValid: (trnspId = Transponder.trnspId) and (TransponderAccount.status = Valid)	T		
Actions:			
	«optional input step» System scans transponder (in trnspId)	trnspId = TR1	Pass
	«optional output step» System stores trip transaction (out accountId, out location)	location = B2, accountId = 112233	Pass
	«optional internal step» System checks transponder account		
	«optional output step» System stores authorization code (out code)	code = Valid Entry	Pass
Postconditions:			
Vehicle Trip	Entry processed		

6 CAdET-SoC: Extending CAdET with Separation of Concerns

This chapter describes how CAdET is extended to use a separation of concerns variability mechanism to form CAdET-SoC. In CAdET, feature conditions are used to associate features with test specifications in a decision table, and to associate features with variation point values in a test specification. CAdET distinguishes between the binding times of coarse-grained functional variability (feature to test specification) and fine-grained variability (feature to variation point). The values of feature conditions associated with test specifications are bound during SPL engineering, while the values of feature conditions associated with variation points are bound during feature-based test derivation. Delaying the binding of the fine-grained variability improves the reusability of the test specifications by reducing the number of test specifications that need to be created and maintained for a SPL.

However, applying a parameterization mechanism in CAdET incurs additional overheads, such as the effort needed to implement this mechanism to configure variation points during SPL engineering. If a use case variation point contains a larger number of values, a decision table (created from a use case activity diagram) will need to be modified during SPL engineering to describe all the variable test steps associated with these values. Also, if the same use case variation point is repeated across several decision tables, then the same variable test step needs to be described repeatedly in each of these

decision tables. For example, suppose a variation point with three values impacts four decision tables. Each of these four decision tables needs to be modified to describe each of these three values, resulting in a total of $3+3+3+3$, or $3*4=12$ modifications.

Separation of concerns alleviates some problems with configuring the fine-grained variability in the test specifications of a SPL that has many variation points repeated across several use cases (and hence across several decision tables). Separation of concerns is the principle that a given problem involves different kinds of concerns, or aspects, which should be identified and separated in order to achieve the required software engineering quality factors such as robustness, adaptability, maintainability, and reusability (Aksit, Tekinerdogan et al. 1996).

CADeT-SoC is an extension of CADeT that uses a separation of concerns variability mechanism to configure the fine-grained functional variability in the decision tables of a SPL. Separation of concerns is not used to configure the large-grained variability in these tables, since they are designed in Phase II in Chapter 5 to be separate documents that can be grouped and associated with features from a SPL.

Separation of concerns is used in CADeT-SoC to physically separate the variable test steps that correspond to a variation point from a decision table. Then, these variable test steps are explicitly grouped and associated with the corresponding feature during SPL engineering. Thus, each unique variable test step is defined at one time in one location, regardless of the number of times a variation point is repeated in the decision tables. This facilitates the maintenance and reuse of the fine-grained variability in the test specifications of a SPL.

6.1 Separation of Concerns Variability Mechanism in CAdET-SoC

Four different separation of concerns variability mechanisms were considered for integration with CAdET-SoC: AOP (Kiczales, Lamping et al. 1997), Framed Aspects (Loughran and Rashid 2004), XVCL Frames (Zhang and Jarzabek 2004), and the Static Client Application Customization (SCAC) technique (Saleh 2005; Saleh and Gomaa 2005). AOP (Kiczales, Lamping et al. 1997) and Framed Aspects (Loughran and Rashid 2004) were considered, but not selected, because these mechanisms required the test specifications to be written in the Java programming language, and do not include built-in constructs to associate features with variation point in a test specification. XVCL Frames (Zhang and Jarzabek 2004) is language independent, but was not selected, because like the former variability mechanisms, XVCL does not include built-in constructs to associate features with variation points in a test specification. The SCAC pattern and tool (Saleh 2005; Saleh and Gomaa 2005) was selected because it is language independent, easy to learn, and contains built-in constructs to associate features with variable test steps. Furthermore, SCAC enables the application code and test specifications to be customized together during feature-based application and test derivation for an application configuration.

CAdET-SoC adapts the SCAC technique (Saleh 2005; Saleh and Gomaa 2005) to configure the variability in the test specifications of a SPL. CAdET-SoC replaces phases IV and V of CAdET with the following phases:

- Phase IV_{SoC}: Apply separation of concerns to test specifications during SPL engineering

The following phase is done during application engineering for each application:

- Phase V_{SoC} : Apply feature-based test derivation using separation of concerns

The following sections describe Phases IV_{SoC} and V_{SoC} of CADeT-SoC.

6.2 Phase IV_{SoC} : Applying Separation of Concerns to Test Specifications During SPL Engineering

The SCAC technique (Saleh 2005; Saleh and Gomaa 2005) was selected and extended to implement separation of concerns in CADeT-SoC. Instead of using the SCAC technique to represent and bind the variability in program code, CADeT-SoC extends SCAC to represent and bind the variability in the test specifications of a SPL. The Static Customization of Test Specifications (SCT) technique is an extension of SCAC that can be used to separate variable test steps from the test specifications of a SPL, and then associate these test steps with an alternative or optional feature from the SPL.

6.2.1 Extending SCAC for SCT

An *insertion point* is a notation used by the SCAC method to uniquely identify and name a location of variation in the code (Saleh and Gomaa 2005). In SCT, a *test insertion point* is a notation used to uniquely identify and name a location of variation in the decision tables.

SCAC provides a feature language that is used to relate features to variable code, and then relate the variable code to insertion points in the common code of a SPL (Saleh and Gomaa 2005). In SCT, this language is used to relate features to variable test steps,

and then relate the variable test steps to test insertion points in the SPL decision tables, and test specifications generated from these tables.

The keywords of this feature description language are described in Table 21. A feature scope is identified using the `$FEATURE[featureName]` and `$ENDFEATURE` keywords, where *featureName* identifies an optional or alternative feature in the SPL. Enclosed within a feature scope are one or more test insertion point names preceded by `$START-$END` keywords. The variable test steps that correspond with the feature are identified within the scope of a test insertion point name (as shown in Figure 25).

```
$FEATURE[Alarm]

$START insD
«optional output step»  Sound warning alarm
$END insD

$ENDFEATURE[Alarm]
```

Figure 25 Association of the alarm feature with a variable test step

A combination of features can also be associated with a variable test step. The `$FEATUREINTERACTION[C, D]` and `$ENDFEATUREINTERACTION` keywords are used to associate a combination of features to one or more test insertion points. The parameters *C, D* identify two optional feature names. The `$IF-$ELSEIF` conditional statement within a feature interaction describes the combinations in which both, one, or the other feature are selected. A combination of more than two features can be described by inserting additional parameters after the `$FEATUREINTERACTION` keyword, as in `[C, D, E, ..., Z]`.

Table 21 Feature description language

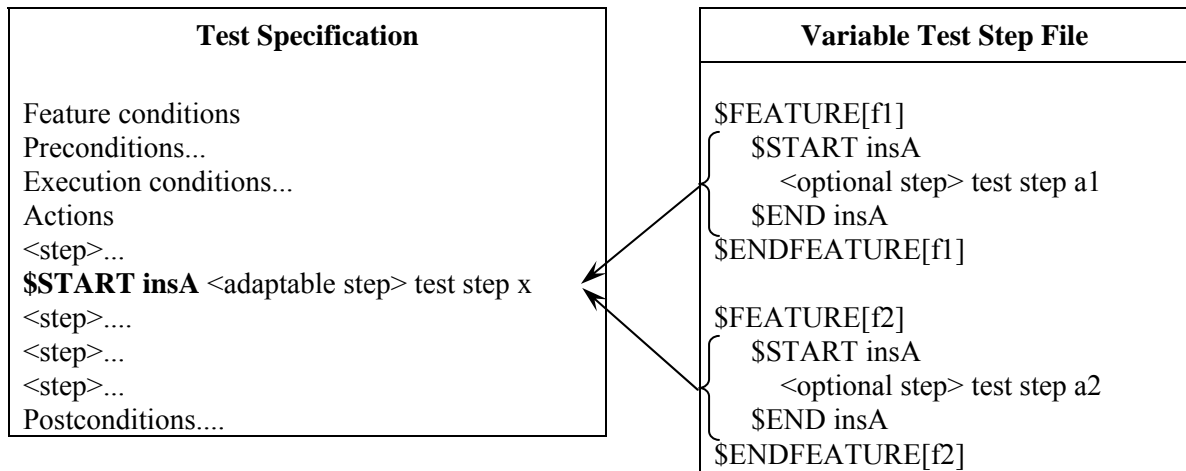
Keyword	Description
\$FEATURE [featureName] \$START testInsertionPointName Variable test step \$END testInsertionPointName \$ENDFEATURE [featureName]	Explicitly groups and associates a feature with one or more variable test steps.
\$FEATUREINTERACTION [C,D] \$START testInsertionPointName \$IF FEATURE [C,D] //Both Variable test step 1 \$ELSEIF FEATURE [C] //C Only Variable test step 2 \$ELSEIF FEATURE [D] //D Only Variable test step 3 \$ENDIF \$END testInsertionPointName \$ENDFEATUREINTERACTION [C,D]	Explicitly associates a combination of features with one or more variable test steps.

A *variable file* is a document used by the SCAC method to represent the relationships between the features, insertion points, and variable code to the common code of a SPL (Saleh 2005; Saleh and Gomaa 2005). In SCT, a *variable test step file* is a document that represents the relationships between the features, test insertion points, and variable test steps to the decision tables and test specifications of a SPL.

Table 22 shows the relationships between test insertion points in a test specification, and between features, test insertion points and variable test steps in a variable test step file. The test insertion point *insA* identifies a location of variation in the test specification text file. This test insertion point is impacted by two features, *f1* and *f2*, which are associated with different optional test steps in the variable test step file. Selecting feature *f1* will insert “<optional step> test step a1” at all *insA* test insertion points in the SPL test specifications during feature-based test derivation. Likewise, selecting feature *f2* will

insert “<optional step> test step a2” at all *insA* test insertion points in the SPL test specifications during feature-based test derivation.

Table 22 Relationship between insertion points, test specifications and variable test step file



Sometimes the features associated with an adaptable test step interact, as shown by the relationship of the *f1* and *f2* parameters to test insertion point *insB* in Table 23. The \$FEATUREINTERACTION keyword is used in the variable test step file to describe a feature interaction between *f1* and *f2*, followed the insertion point name *insB*, and a conditional statement describing the combinations in which both, one, or the other feature are selected. If both *f1* and *f2* are selected, then “<optional step> test step b1” will be inserted at all *insB* test insertion points in the SPL test specifications during feature-based test derivation. Else, if either *f1* or *f2* is selected, then either “<optional step> test step b2” or “<optional step> test step b3” will be inserted at all *insB* test insertion points in the SPL test specifications during feature-based test derivation.

Table 23 Representing interacting features in variable file

Test specification text file	Variable File
Feature conditions...	\$FEATUREINTERACTION[f1,f2]
Preconditions...	<pre> \$START insB \$IF FEATURE(f1, f2) <optional step> test step b1 \$ELSEIF FEATURE(f1) <optional step> test step b2 \$ELSEIF FEATURE(f2) <optional step> test step b3 \$ENDIF \$END insB \$ENDFEATURE[f1, f2]</pre>
Execution conditions...	
Actions	
<step>...	
\$START insA <adaptable step> test step x	
<step>....	
\$START insB <adaptable step> test step y	
<step>...	
Postconditions....	

6.2.2 Applying SCT to the Test Specifications of a SPL

An overview of how the SCT technique is applied to the test specifications of a SPL during SPL engineering is shown in Figure 26. First, test insertion points are manually added to tag the variation points in the decision tables created using CADeT. A test specifications document is generated from these modified decision tables using CADeT's test specification generator tool. Next, the decision tables, and test specifications in the test specifications document are exported as separate text files, in order to be compatible with the tools in the Software Product Line Environment Tool

(SPLET) (Saleh 2005; Saleh and Gomaa 2005).

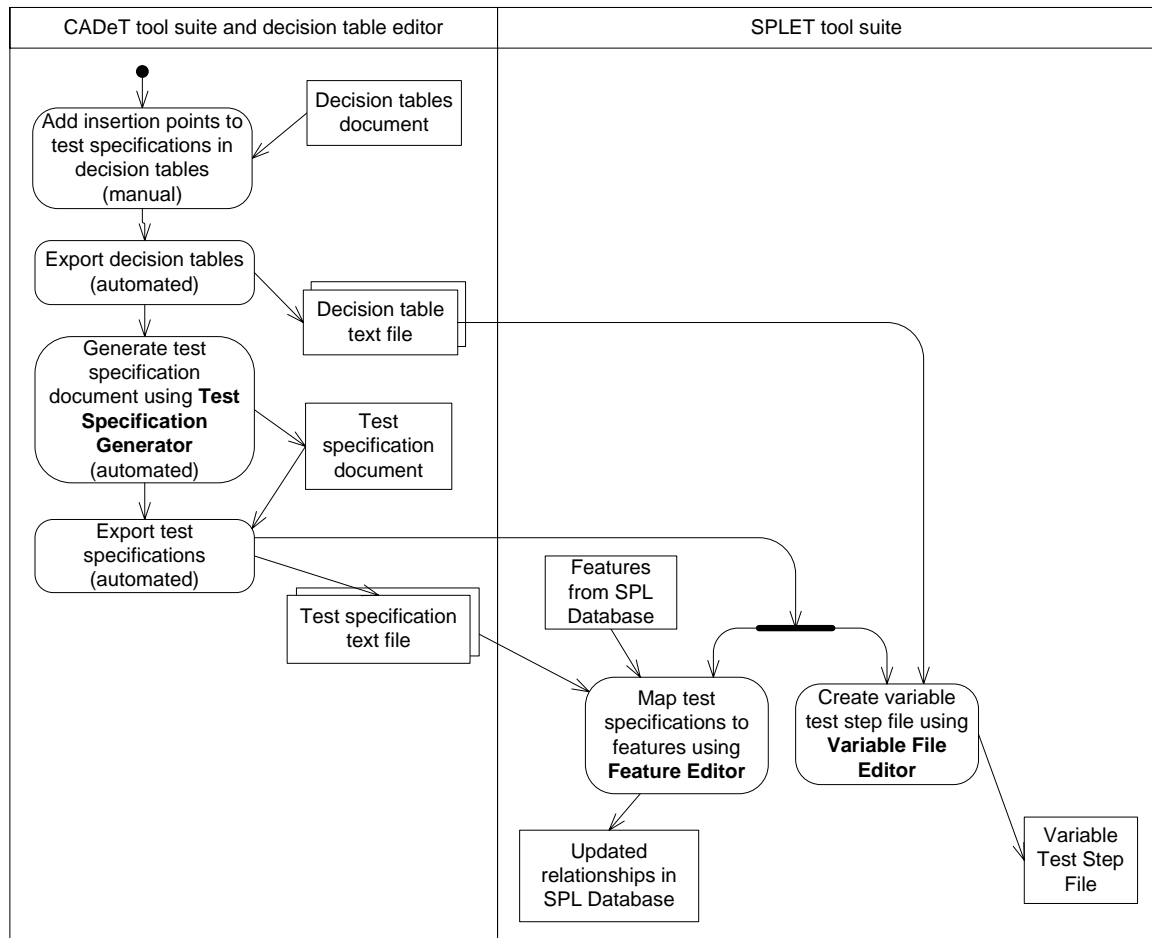


Figure 26 Application of SCT during SPL engineering

SPLET (Saleh 2005; Saleh and Gomaa 2005) is a tool that automates part of the SCT technique. SPLET is used to associate the features in the SPL with the exported test specification and decision table text files. The Feature Editor component of SPLET is used to map the test specification text files to the features of the SPL. The Variable File Editor component is used to map the test insertion points in the decision table text files to the features of the SPL. Then, the Variable Editor component is used to generate a

variable test step file, which contains a list of features in the SPL, the test insertion points that are associated with each feature, and the variable test steps.

6.2.3 Example of Applying SCT to the Test Specifications of a SPL

The following example illustrates how SCT is applied to the “Enter through transponder-enabled booth” decision table and related test specifications of the AHTS SPL. First, test insertion points are added to identify the adaptable test steps in the decision table. Figure 27 shows the test insertion points and excerpt from variable test step file created for the “Enter through transponder-enabled booth” decision table of the AHTS SPL. The test insertion point “\$START insD” was added to identify a location of variation in the “«adaptable output step» Warn vehicle” in the decision table in Figure 27. A variable test step file was created to show the features and variable test steps associated with the *insD* test insertion point. The variable test step “«optional output test step» Sound warning alarm” is associated with the selection of the Alarm feature, and the variable test step “«optional output test step» System turns light yellow” is associated with the selection of the Traffic Light feature at *insD*.

This process is repeated for each feature at each insertion point, until each insertion point is associated with at least one feature in the SPL. An excerpt of the variable test step file generated for the AHTS SPL is shown in Figure 28. This file is read during feature-based test derivation to customize the test specifications for an application of the AHTS SPL.

ET2.1	Enter through transponder enabled booth use case
	Test Specifications
Feature conditions	TransponderBooth
Preconditions	VehicleTrip
Execution conditions	TransponderBoothEntry
	TransponderDetected
	AccountValid
Actions	
1	«optional input step» System scans transponder (in trnspld)
2	«optional output step» System stores trip transaction (out accountId, out time)
3	«optional internal step» System checks transponder account
4	«optional output step» System times out and stores invalid trip transaction (out boothId, out time)
5	\$START insD «adaptable output step» Warn vehicle (vpAlarm, vpLight)
6	«optional output step» System stores authorization code (out code)
Post conditions	VehicleTrip

```

$FEATURE[Alarm]
$START insD
«optional output step»
Sound warning alarm
$END insD

$ENDFEATURE[Alarm]

-----
$FEATURE[Traffic
Light]

$START insD
«optional output step»
System turns light
yellow
$END insD

$ENDFEATURE[Traffic
Light]

```

Figure 27 Example of test insertion points and variable test step file

```

$FEATURE[Alarm]
$START insD
    «optional output step» Sound warning alarm
$END insD
$START insE
    «optional output step» Sound unauthorized entry alarm
$END insE
$ENDFEATURE[Alarm]
////////////////////////////////////
$FEATURE[Barrier]
$START insB
    «optional output step» System raises barrier
$END insB
$START insC
    «optional output step» System lowers barrier
$END insC
$ENDFEATURE[Barrier]
////////////////////////////////////
$FEATURE[Traffic Light]
$START insB
    «optional output step» System turns traffic light green
$END insB
$START insC
    «optional output step» System turns traffic light red
$END insC
$START insD
    «optional output step» System turns light yellow
$END insD
$ENDFEATURE[Traffic Light]

```

Figure 28 Excerpt of variable feature file for AHTS SPL

Text specification files were generated from the “Enter through transponder-enabled booth” decision table of the AHTS SPL. These files were then mapped to the TransponderBooth feature of the AHTS SPL using the Feature Editor component of SPLET.

6.2.4 Phase V_{SoC} : Applying Feature-Based Test Derivation using Separation of Concerns

In the SCAC method, the SPLET tool is used to select features for an application of the SPL (feature-based application configuration), and then apply feature-based application derivation to generate code for an application of that SPL (Saleh 2005).

During feature-based application configuration, features are selected for an application of the SPL. The selected features are checked for consistency, and then a customization file is generated describing the selected features. During feature-based application derivation the SPLET tool uses separation of concerns to select variable code according to the selected features, and then combine the variable code with the common code of the SPL.

In Phase V_{SoC} of CADeT-SoC, the SCAC method and tool are adapted in the SCT technique to apply feature-based test derivation to the test specifications of an SPL. Figure 29 shows an overview of how the SCT technique (shaded in gray) is applied to the test specification of a SPL. The File Extractor component of SPLET reads the customization file (which was generated during application configuration) to automatically select a set of test specifications for the application. The Static Code Weaver component reads the feature selections from the customization file, the variable test steps from the variable test step file, and the references to test insertion points from the adaptable test specifications. Then, the Static Code Weaver replaces the test insertion points in the adaptable test specifications with the variable test steps associated with feature selections of an application.

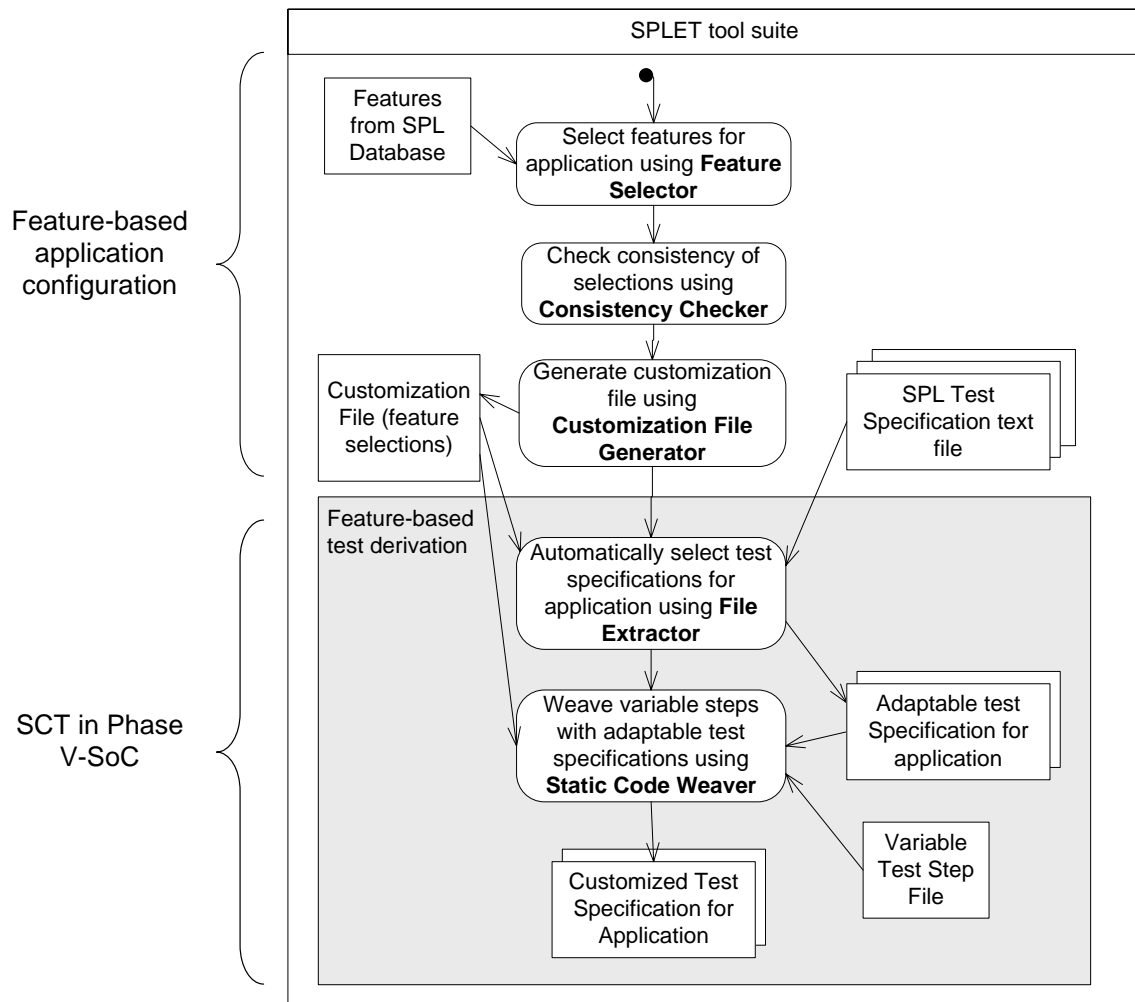


Figure 29 Application of SCT during feature-based test derivation

Figure 30 shows the Code Weaver tab in the SPLET tool. The names of the variable file, customization file, directory of the adaptable test specifications, and output directory are provided by the application test engineer. The “Static” button has been selected to apply feature-based test derivation using SCAC.

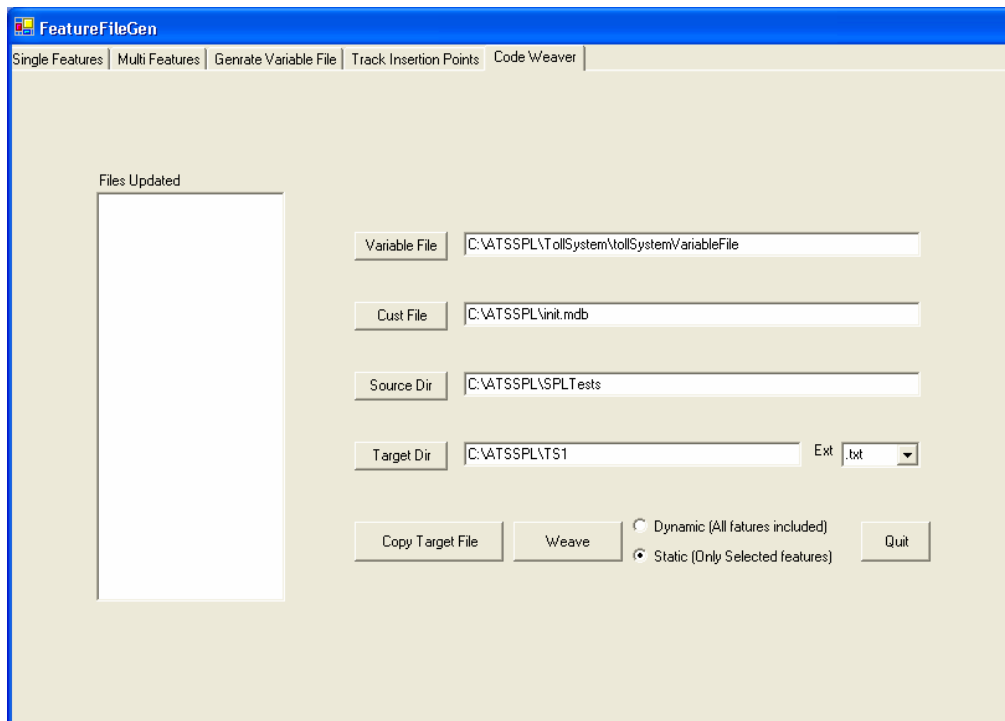


Figure 30 Code weaver tab in SPLET tool

6.2.5 Example of Applying Feature-Based Test Derivation using SCT

Application TS1 from the AHTS SPL test plan in Table 9 has been configured to include all toll booth types and the Traffic Light feature. The following example describes how SCT is used to apply feature-based test derivation to the test specifications of the “Enter through transponder-enabled booth” use case decision table (see Figure 27) for TS1.

First, the File Extractor component of SPLET is used to select a set of test specifications for TS1, which includes the “«adaptable» Invalid transponder” test specification of the “Enter through transponder-enabled booth” use case decision table shown in Figure 31. Then, the Static Code Weaver component of SPLET is used to configure the test insertion points in the test specifications of TS1.

Use case name:	Enter through transponder enabled booth
Test specification name:	«adaptable» Invalid transponder
Feature conditions:	TransponderBooth = T
Preconditions:	VehicleTrip = EntryDetected
Execution conditions:	TransponderBoothEntry = T TransponderDetected = F
Actions:	«optional input step» System scans transponder (in trnspId) «optional output step» System times out and stores invalid trip transaction \$START insD «adaptable output step» Warn vehicle (vpAlarm, vpLight)
Post conditions:	VehicleTrip = EntryProcessed

Figure 31 Test specification for Invalid Transponder

Figure 32 shows an example of how the “«adaptable» Invalid Transponder” test specification in Figure 31 is customized for TS1. Both the Alarm and Traffic Light features of the AHTS SPL impact *insD* , but only the Traffic Light feature has been selected for TS1. The test insertion point *insD* has been removed and replaced with “«optional output step» System turns light yellow”, which corresponds to the Traffic Light feature.

Use case name:	Enter through transponder enabled booth
Test specification name:	«adaptable» Invalid transponder
Feature conditions:	TransponderBooth = T
Preconditions:	VehicleTrip = EntryDetected
Execution conditions:	TransponderBoothEntry = T TransponderDetected = F
Actions:	«optional input step» System scans transponder (in trnspld) «optional output step» System times out and stores invalid trip transaction // \$START insD «adaptable output step» Warn vehicle (vpAlarm, vpLight) «optional output step» System turns light yellow
Post conditions:	VehicleTrip = EntryProcessed

Figure 32 Example of customized test specification

6.3 Comparison of CAdET and CAdET-SoC

Table 24 compares the number of variable test steps defined using parameterization for each adaptable test step in the decision tables of the AHTS SPL, against the number of variable test steps defined using separation of concerns for each adaptable test step in the same decision tables. Only 11 variable test steps needed to be defined using separation of concerns in CAdET-SoC, instead of the 18 variable test steps defined using parameterization in CAdET.

Table 24 Number of variable test steps defined for variation points in AHTS SPL decision tables

Decision table	Adaptable test steps	Parameterization	Separation of Concerns
Enter Toll Road	«adaptable output step» Authorize Vehicle (vpBarrier, vpLight)	2	2
	«adaptable output step» System resets toll booth (vpBarrier, vpLight)	2	2

Enter through transponder enabled booth	«adaptable output step» Warn vehicle (vpAlarm, vpLight)	2	2
Enter through ticket-issuing booth	None	0	0
Exit Toll Road	«adaptable output step» Process unauthorized vehicle (vpCamera, vpAlarm)	2	1
	«adaptable output step» Authorize vehicle (vpBarrier, vpLight)	2	0
	«adaptable output step» System resets toll booth (vpBarrier, vpLight)	2	0
Exit through transponder-enabled booth	None	0	0
Pay with transponder account	«adaptable output step» System calculates toll from transponder (vpTollCharge)	2	2
	«adaptable output step» Warn vehicle (vpAlarm, vpLight)	2	0
Exit through ticket-issuing booth	None	0	0
Process ticket	«adaptable output step» System calculates toll from ticket (vpTollCharge)	2	2
Exit through full-service booth	None	0	0
Pay with Cash	None	0	0
Pay with Credit Card	None	0	0
Pay Operator	None	0	0
Total number variable test steps		18	11

6.4 Summary

This chapter described an alternative variability mechanism that uses separation of concerns to customize the fine-grained functional variability in the test specifications of an SPL. Like the parameterization variability mechanism in Chapter 5, this technique delays the binding time of the fine-grained functional variability in the test specifications, in order to increase reusability by reducing the number of test specifications that need to

be created and maintained for an SPL. However, this technique is more suitable than parameterization for configuring the test specifications of a SPL that has many variation points repeated across several use cases. With this separation of concerns technique, each unique variable test step is defined one time in one location, regardless of the number of times a variation point is repeated in the test specification suite. This facilitates the maintenance and reuse of the fine-grained variability in the test specification suite of a SPL.

7 Evaluation of CAdET and CAdET-SoC

This research applied the case study method (see rationale in section 7.1) to evaluate the following hypothesis on two SPLs:

A test design method can be developed to create reusable and functional test specifications to satisfy use case-based and feature-based coverage criteria for a SPL, where these test specifications can be configured during feature-based test derivation to test a set of applications derived from the SPL.

In this research, the selected use case-based and feature-based coverage criteria required the test specifications to *cover all use case scenarios, features and relevant feature combinations of a SPL.*

CAdET and CAdET-SoC were applied to two SPLs in three separate studies: an Automated Highway Toll System (AHTS) SPL and a Banking System SPL. Reusable test specifications were created to cover all use scenarios in each SPL. Then, a set of representative application configurations was selected to cover all features and relevant feature combinations in each SPL. The reusable test specifications were customized for each application configuration in each SPL using CAdET and CAdET-SoC. A set of

applications was derived from a Banking System SPL implementation and then tested using the customized test specifications.

7.1 Rationale for Selecting Case Study Research Method

The *case study research method* is an empirical investigation of the effect of a contemporary phenomenon (method, tool, etc...) within its real life context, when the boundaries between the phenomenon and context are not clearly distinguishable, and in which multiple sources of evidence are used. A case study research method is relevant in situations where the research question is explanatory (asks how, why?), a researcher has little to no control over behavioral events, and the research focuses on contemporary events rather than historical events (Yin 2003). The goal of an *exploratory study* is to formulate a hypothesis, while the goal of an *explanatory case study* is to test a hypothesis to evaluate the cause and effect relationships of a contemporary phenomenon (method, tool, etc...) on one or more cases. Analytic, rather than statistical generalization is used to relate the results to hypothesis (Yin 2003).

An explanatory case study research method was selected to investigate whether the CAdET and CAdET-SoC test design methods could be used to create reusable and functional test specifications to cover all use case scenarios, features and relevant feature combinations of a SPL, and then configure these test specifications during feature-based test derivation to test a set of applications derived from the SPL. The case study method was relevant because the research question was explanatory; there was little control over external factors, such as the context in which the test methods might be used; and the research evaluated a contemporary event.

7.2 Description of Evaluation

Table 25 describes the three studies (labeled 1, 2, 3) undertaken to assess the hypothesis against an Automated Highway Toll System (AHTS) SPL (described in Chapter 5), and a Banking System SPL (described in Appendix A). The purpose of these studies was to evaluate whether CAdET and CAdET-SoC could be used to create functional test specifications to cover the use case scenarios of each SPL (Phases I-II); determine the relevant feature combinations, and apply a feature-based coverage criterion to select a set of representative applications for each SPL (Phase III); configure these test specifications during feature-based test derivation for each application (Phases IV, IV-SoC, V, V-SoC); and then test these applications (Phases VI-VII).

Different phases of CAdET and CAdET-SoC were evaluated over a span of several semesters on one or both SPLs. This was because some phases of the method needed to be defined and tested before the remaining phases could be applied. The subject matter experts (persons with expertise in a particular area) in the first and third studies were graduate students from an advanced software design class. The subject matter expert in the second study was the researcher.

Table 25 Studies used to evaluate CAdET and CAdET-SoC

Type of study	Subject matter experts	Purpose of study	AHTS SPL	Banking System SPL
1. Single case	Five graduate students (A, B, C, D, E)	Evaluate an initial version of CAdET (Phase I-II).	√	
2. Multiple case	Researcher	Evaluate Phases I-V of CAdET	√	√
3. Single case	Five graduate students (F, G, H, I, J)	Evaluate phases IV-VII of CAdET and phases IV-SoC and V-SoC of CAdET-SoC		√

7.3 Preliminary Study to Evaluate Feasibility of Initial Version of CAdET

The first study was a preliminary study to assess whether a graduate student could follow an initial version of CAdET to create activity diagrams, decision tables and test specifications from the feature model, use case model, and feature to use case relationship table of a SPL.

7.3.1 Description of Study

In this study, five graduate students (participants) from an advanced software design class learned and applied four sets of instructions from an initial version of CAdET to the requirement models of an AHTS SPL. Each participant in the study had created a feature model, use case model, and feature to use case relationship table for an AHTS SPL as part of a group project. Table 26 shows the requirements models assigned to each participant in the study. Three different sets of models, created earlier by three groups, are labeled as g1, g2, and g3. The number of use cases in each use case model

consisted of between 8 and 13 use cases, and the number of features in each feature model consisted of between 21 and 22 features.

Table 26 Requirements models of AHTS SPL

Participant	A	B	C	D	E
Group	g1	g2	g2	g3	g3
# Use cases	8	13	13	11	11
# Features	21	22	22	22	22

Each participant followed four sets of instructions to apply Phases I-II of an initial version of CADeT to a set of use case and feature models. With the first set of instructions, “Create activity diagrams for a single system”, participants were directed to ignore the variability in the use case descriptions of the AHTS SPL and to manually create activity diagrams from these use case descriptions in the same way as for a single system. With the second set of instructions, “Create activity diagrams for a SPL”, participants were directed to map features from the feature model to use case activity diagrams and activity nodes. With the third set of instructions, “Create decision tables for a SPL”, participants were directed to create decision tables for some of the use cases in the SPL, manually trace paths from the use case activity diagrams for each use case scenario, and then map these paths to columns in the decision tables. With the fourth set of instructions, “Create Test Templates for a SPL”, participants were asked to run a macro to generate partial test specifications from the decision tables.

Each phase was scheduled to take about two weeks. During that time, participants were encouraged to ask questions to get help correcting the models, and to keep track of the time spent creating the models. As soon as a participant completed the

instructions, the models were evaluated by the researcher and modifications were suggested via email. Sometimes the researcher or the participant would schedule a meeting to help clarify the instructions. After the study was completed, each participant answered a questionnaire that asked about the participant's background, time spent completing each phase, perceptions on the difficulty of each phase, and suggestions for improvement.

7.3.2 Results

The evaluation of the models consisted of checking whether the participant was able to follow and understand the rules in each set of instructions. Table 27 summarizes the ability of participants to follow each set of instructions. A check mark means that the participant was able to follow the instructions without additional assistance from the researcher to create the models correctly; a check minus mark means that the participant had difficulty understanding and applying the instructions, and needed help from the researcher to correct the models.

Table 27 Assessment of initial version of CADeT

Instruction set	A	B	C	D	E
Create Activity Diagrams for a Single System	√	√	√	√-	√-
Create Activity Diagrams for a SPL	√-	√	√-	√-	√-
Create Decision Tables for a SPL	√-	√-	√	√-	√-
Create Test Templates for a SPL	√-	√-	√-	√-	√-

Three out of five participants were able to follow the first set of instructions to create activity diagrams for a single system, but almost all participants had difficulty

adapting these activity diagrams for a SPL. Further, almost all participants had difficulty representing variability in the decision tables and test specifications.

The results of the questionnaire are shown in Table 28. Only one of the participants had some experience with software testing, and all had some background in UML modeling methods. Although each participant was asked to keep track of the time spent creating the models, only participant C actually recorded the time spent in each phase. The other times are approximations based on what the participant remembered.

Table 28 Results of questionnaire for first applied project

	A	B	C	D	E
Experience w/ UML modeling methods	4 classes at GMU	2 years	8 years	2 classes at GMU	4 classes at GMU
Experience w/ software testing	None	None	1 class at GMU, some industry experience	None	None
Time to create activity diagrams from use cases	9 hours	6 hours	18-20 hours	24 hours	8 hours
Time to create decision tables from activity diagrams	9 hours	6 hours	8-10 hours + 4 hours revising models	24 hours	5 hours
Time to create test specifications from decision tables	9 hours	6 hours	6 hours	24 hours	2 hours
What is the easiest phase?	Create activity diagrams from use cases	Create activity diagrams from use cases	Create test specifications from decision tables	Create test specifications from decision tables	Create test specifications from decision tables
What is the most difficult phase?	Create decision tables from activity diagrams	Create decision tables from activity diagrams	Create decision tables from activity diagrams	Create activity diagrams from use cases	Create activity diagrams from use cases
How can the method be improved?	Define terms	Improve instructions and add examples	Add automation; give tutorials; define terms	Use color-coding to group related variation points	Use examples to illustrate method

7.3.3 Interpretation of Results

This study showed it was possible, but difficult for a graduate student with some background in SPL modeling methods to follow an initial version of CADeT to represent variability in the activity diagrams and decision tables of a SPL. One reason for this may

be attributed to the way the method was taught to the participants, as several participants indicated a need for more tutorials and examples (see Table 28).

The results of this study prompted revisions to CAdET, and revisions to the approach used to teach CAdET to other participants in a later study. Initially, CAdET required each participant to manually map the test specifications in decision tables to test specifications of the SPL. This mapping was automated in a later version of CAdET. Also, a mechanism of automatically selecting the test specifications during feature-based test derivation was implemented in the later version of CAdET. Some of the participant's suggestions for improving the teaching of the method, such as adding examples, were incorporated in the third study.

7.4 Evaluate Feasibility of Creating and Customizing Test Specifications Using CAdET (Phases I-V)

The second study was applied by the researcher, to evaluate whether CAdET could be used to create functional test specifications to cover the use case scenarios of each SPL (Phases I-II); analyze the relationships of features to test specifications to determine the relevant feature combinations, and then apply a feature-based coverage criterion to select a set of representative applications for each SPL (Phase III); and configure these test specifications during feature-based test derivation for each application (Phases IV, V). Furthermore, the researcher compared the number of test specifications created using CAdET against the number of test specifications created using two alternative test design methods to cover all use case scenarios, all features and all relevant feature combinations in each SPL.

7.4.1 Description of Study

In this study, the researcher took the role of a subject matter expert, and applied PLUS (Gomaa 2005) to create a feature model, use case model, and feature to use case relationship table for the AHTS SPL and Banking System SPLs. The following sections describe the characteristics of the requirement models created in this study for each SPL, summarize the activities applied in each phase, and then describe the results of applying these activities to each SPL.

7.4.2 Characteristics of Requirement Models

Table 29 summarizes the characteristics of the requirements models created in this study using PLUS (Gomaa 2005) for each SPL. The researcher created an AHTS with 11 use cases, 28 use case scenarios and 16 features. Ten out of sixteen features of the AHTS SPL were associated with one or more use cases. Seven out the ten features were associated with use cases and variation points (e.g. the TranponderBooth feature maps to vpEntryBooth in “Enter Toll Road” use case and the “Enter through Transponder-Enabled Booth” extension use case).

The Banking System SPL of (Webber 2001) was adapted to use the PLUS method in this study (see Appendix A). The Banking System SPL had 7 use cases, 21 use case scenarios, and 12 features. The ATM kernel feature was associated with every use case in the Banking System SPL, and the remaining features were associated with variation points in the Banking System SPL. The following sections describe the application of phases I-V of CADeT to each SPL.

Table 29 Characteristics of requirement models created for each SPL

	ATHS SPL	Banking System SPL
# Use cases	11	7
# Use case scenarios	28	21
# Features	16	12
# Features associated with use cases	3	1
# Features associated with both use cases and variation points	7	0
# Features associated with variation points	6	11

7.5 Application of Phases I-V: Creating and Customizing Test Specifications for the AHTS SPL

The researcher applied phase I of CADeT to create activity diagrams for each use case of the AHTS SPL in Table 29. Excerpts from these activity diagrams were used to illustrate the CADeT method in Chapter 5.

7.5.1 Coverage of All Use Case Scenarios and All Features in AHTS SPL

Then, the researcher followed phase II of CADeT to trace paths from the activity diagrams for each use case scenario, and then map these paths to test specifications in decision tables. Excerpts from these decision tables are shown in Chapter 5. A total of 30 test specifications were created to cover the 28 use case scenarios of the AHTS SPL.

Figure 33 shows the associations between the 28 use case scenarios and 30 test specifications of the AHTS SPL. Each scenario in Figure 33 was covered by at least one test specification. Some scenarios, such as “Enter toll road: Main scenario”, were covered by more than one test specification because the flow of the scenario was diverted by an extension or included use case scenario. Other scenarios, such as “Exit through Ticket-

issuing Booth: Ticket not recognized” and “Exit through Full-Service Booth: Ticket not recognized” were covered by the same test specification, because these scenarios described common behavior that was factored out into a “Process Ticket” aggregate activity decision table.

The set of test specifications created for the AHTS SPL also covered all features of this SPL. Table 30 shows the relationship of each feature to the test specifications of the AHTS SPL. The “Feature” column lists the features of the AHTS SPL. Each feature in the AHTS SPL is associated with at least one test specification, and each test specification is associated with at least one feature. Six out of the sixteen features (Camera, Barrier, Traffic Light, Alarm, Variable Toll Charge, Fixed Toll Charge) are associated with a variation point in one or more adaptable test specifications while the remaining ten features are associated with an entire reuse as is or adaptable test specification. Thus, most of the variable features of the AHTS SPL represented a coarse granularity of functional variation in these test specifications (refer to explanation of fine-grained and coarse-grained variability in phase IV of Chapter 5).

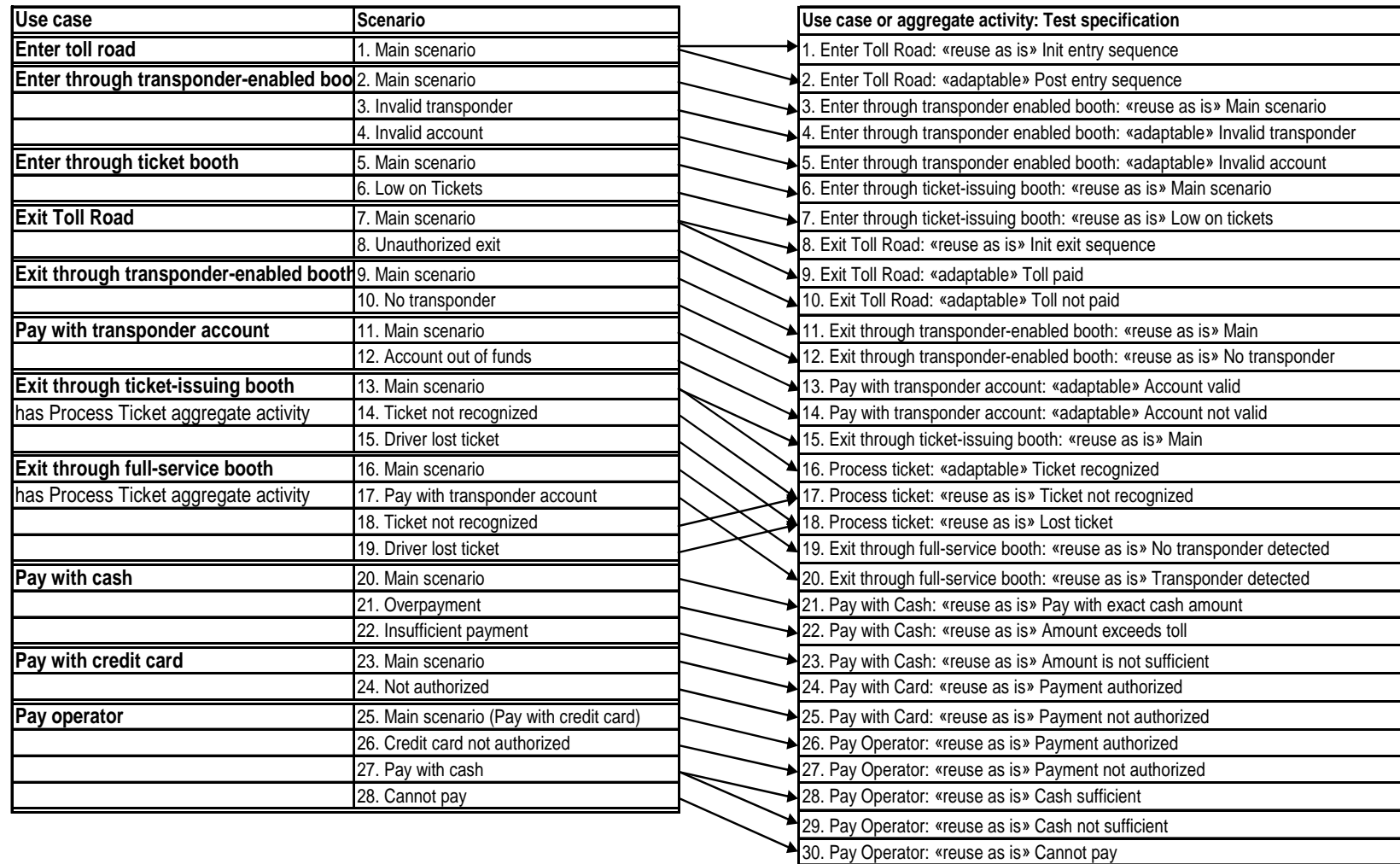


Figure 33 Relationship between use case scenarios and test specifications of AHTS SPL

Table 30 Features / test specifications relationships in AHTS SPL

Feature	Use case or aggregate activity	Test specification	Variation point
«common» Automated Toll System Kernel	Enter Toll Road	1. «reuse as is» Init entry sequence	
		2. «adaptable» Post entry sequence	
	Exit Toll Road	8.: «reuse as is» Init exit sequence	
		9. «adaptable» Toll paid	
		10. «adaptable» Toll not paid	
«optional» Ticket Entry/Exit Booths	Enter through ticket-issuing booth	6. «reuse as is» Main scenario	
		7. «reuse as is» Low on tickets	
		13. «reuse as is» Main	
«optional» Full Service Exit Booth	Exit through full-service booth	19. «reuse as is» No transponder detected	
		20. «reuse as is» Transponder detected	
«optional» Transponder Entry/Exit Booths	Enter through transponder enabled booth	3. «reuse as is» Main scenario	
		4. «adaptable» Invalid transponder	
		5. «adaptable» Invalid account	
	Exit through transponder-enabled booth	11. «reuse as is» Main	
		12. «reuse as is» No transponder	
«optional» Ticket Dispenser	Enter through ticket-issuing booth	6. «reuse as is» Main scenario	
		7. «reuse as is» Low on tickets	
«optional» Ticket Reader	Process ticket	16. «adaptable» Ticket recognized	
		17. «reuse as is» Ticket not recognized	
		18. «reuse as is» Lost ticket	
«optional» Credit Card Reader	Pay with Card	24. «reuse as is» Payment authorized	
		25. «reuse as is» Payment not authorized	
«optional» Cash Reader	Pay with Cash	21. «reuse as is» Pay with exact cash amount	
		22. «reuse as is» Amount exceeds toll	
		23. «reuse as is» Amount is not sufficient	
«optional» Operator	Pay Operator	26. «reuse as is» Payment authorized	

		27. «reuse as is» Payment not authorized	
		28. «reuse as is»Cash sufficient	
		29. «reuse as is» Cash not sufficient	
		30. «reuse as is» Cannot pay	
«optional» Transponder Account	Pay with transponder account	13. «adaptable» Account valid	
		14. «adaptable» Account not valid	
«optional» Camera	Exit Toll Road	10. «adaptable» Toll not paid	vpCamera
«optional» Barrier	Enter Toll Road	2. «adaptable» Post entry sequence	vpBarrier
	Exit Toll Road	9. «adaptable» Toll paid	vpBarrier
		10. «adaptable»Toll not paid	vpBarrier
«optional» Traffic Light	Enter Toll Road	2. «adaptable» Post entry sequence	vpLight
	Exit Toll Road	9. «adaptable» Toll paid	vpLight
		10. «adaptable» Toll not paid	vpLight
	Enter through transponder enabled booth	4. «adaptable» Invalid transponder	vpLight
		5. «adaptable» Invalid account	vpLight
	Pay with transponder account	14. «adaptable» Account not valid	vpLight
«optional» Alarm	Enter through transponder enabled booth	4. «adaptable» Invalid transponder	vpAlarm
		5. «adaptable» Invalid account	vpAlarm
	Exit Toll Road	10. «adaptable» Toll not paid	vpAlarm
	Pay with transponder account	14. «adaptable» Account not valid	vpAlarm
«alternative» Variable Toll Charge	Pay with transponder account	13.«adaptable» Account valid	vpCharge
	Process ticket	16. «adaptable» Ticket recognized	vpCharge
«alternative» Fixed Toll Charge	Pay with transponder account	13. «adaptable» Account valid	vpCharge
	Process ticket	16. «adaptable»Ticket recognized	vpCharge

7.5.2 Coverage of All Relevant Feature Combinations in AHTS SPL

Next, the researcher analyzed the feature model and test specifications of the AHTS SPL, and applied a feature-based combinatorial coverage criterion to the AHTS SPL to select a set of representative application configurations to test (as described in Phase III of CADeT in Chapter 5).

The feature model of the AHTS SPL was analyzed to calculate a total of 224 possible application configurations (as described in Phase III of Chapter 5). Next, the relationships of the features to test specifications were analyzed to determine the relevant feature combinations.

Table 31 shows the feature combinations associated with the test specifications of the AHTS SPL. The feature condition of the “Automated Toll System Kernel” feature is omitted from the analysis because it is always selected for any application derived from the SPL. The feature conditions that correspond to mutually included features are not selectable, and are denoted in italics. The largest number of feature conditions in a relevant feature combination in Table 31 is two. Thus, at least a pair-wise feature based coverage criterion is needed to check the feature combinations described by these test specifications. A pair-wise feature based coverage criterion was applied using the Jenny tool (Jenkins 2005) to select 8 representative applications for the AHTS SPL. The test plan in Table 9 in Chapter 5 describes the features selected for each of these 8 application configurations.

Table 31 Relevant feature combinations in AHTS SPL

Use case or aggregate activity	Test specification	Feature combinations
Enter Toll Road	1. «reuse as is» Init entry sequence	-
	2. «adaptable» Post entry sequence	barrier + trafficLight
Enter through transponder enabled booth	3. «reuse as is» Main scenario	transponderBooth
	4. «adaptable» Invalid transponder	(transponderBooth * trafficLight)+ (transponderBooth * alarm)
	5. «adaptable» Invalid account	(transponderBooth * trafficLight)+ (transponderBooth * alarm)
Enter through ticket-issuing booth	6. «reuse as is» Main scenario	ticketBooth * <i>ticketDispenser</i>
	7. «reuse as is» Low on tickets	ticketBooth * <i>ticketDispenser</i>
Exit Toll Road	8. «reuse as is» Init exit sequence	-
	9. «adaptable» Toll paid	barrier + trafficLight
	10. «adaptable» Toll not paid	camera + barrier + trafficLight + alarm
Exit through transponder-enabled booth	11. «reuse as is» Main	transponderBooth
	12. «reuse as is» No transponder	transponderBooth
Pay with transponder account	13. «adaptable» Account valid	<i>transponderAccount</i> * <i>tollCharge</i>
	14. «adaptable» Account not valid	(<i>transponderAccount</i> * trafficLight) + (<i>transponderAccount</i> * alarm)
Exit through ticket-issuing booth	15. «reuse as is» Main	<i>ticketBooth</i>
Process ticket	16. «adaptable» Ticket recognized	<i>ticketReader</i> * <i>tollCharge</i>
	17. «reuse as is» Ticket not recognized	<i>ticketReader</i>
	18. «reuse as is» Lost ticket	<i>ticketReader</i>
Exit through full-service booth	19. «reuse as is» No transponder detected	fullServiceBooth
	20. «reuse as is» Transponder detected	fullServiceBooth

Pay with Cash	21. «reuse as is» Pay with exact cash amount	<i>cashReader</i>
	22. «reuse as is» Amount exceeds toll	<i>cashReader</i>
	23. «reuse as is» Amount is not sufficient	<i>cashReader</i>
Pay with Card	24. «reuse as is» Payment authorized	<i>creditCardReader</i>
	25. «reuse as is» Payment not authorized	<i>creditCardReader</i>
Pay Operator	26. «reuse as is» Payment authorized	<i>operator</i>
	27. «reuse as is» Payment not authorized	<i>operator</i>
	28. «reuse as is» Cash sufficient	<i>operator</i>
	29. «reuse as is» Cash not sufficient	<i>operator</i>
	30. «reuse as is» Cannot pay	<i>operator</i>

Next, the researcher applied the parameterization variability mechanism to the AHTS test specifications, in order to automate the configuration of these test specifications for each of the representative applications of the AHTS SPL (see phase IV of CADeT in Chapter 5. The feature conditions in the feature list in Table 3 were associated with test specifications in the decision tables of the AHTS SPL, as described in Chapter 5. Then, the researcher customized these test specifications to cover all features, and relevant feature combinations described in the AHTS SPL test plan in Table 9 in Chapter 5.

Table 32 Test specifications selected for the applications in the AHTS SPL test plan

Use case or aggregate activity	Test Specifications	TS1	TS2	TS3	TS4	TS5	TS6	TS7	TS8
Enter Toll Road	1. «reuse as is» Init entry sequence	√	√	√	√	√	√	√	√
	2. «adaptable» Post entry sequence	√	√	√	√	√	√	√	√
Enter through transponder enabled booth	3. «reuse as is» Main scenario	√	√	√	√		√	√	√
	4. «adaptable» Invalid transponder	√	√	√	√		√	√	√
	5. «adaptable» Invalid account	√	√	√	√		√	√	√
Enter through ticket-issuing booth	6. «reuse as is» Main scenario	√		√	√	√			
	7. «reuse as is» Low on tickets	√		√	√	√			
Exit Toll Road	8. «reuse as is» Init exit sequence	√	√	√	√	√	√	√	√
	9. «adaptable» Toll paid	√	√	√	√	√	√	√	√
	10. «adaptable» Toll not paid	√	√	√	√	√	√	√	√
Exit through transponder-enabled booth	11. «reuse as is» Main	√	√				√	√	√
	12. «reuse as is» No transponder	√	√				√	√	√
Pay with transponder account	13. «adaptable» Account valid	√	√	√	√		√	√	√
	14. «adaptable» Account not valid	√	√	√	√		√	√	√
Exit through ticket-issuing booth	15. «reuse as is» Main	√		√		√			
Process ticket	16. «adaptable» Ticket recognized	√		√	√	√			
	17. «reuse as is» Ticket not recognized	√		√	√	√			
	18. «reuse as is» Lost ticket	√		√	√	√			
Exit through full-service booth	19. «reuse as is» No transponder detected	√		√	√				
	20. «reuse as is» Transponder detected	√		√	√				
Pay with Cash	21. «reuse as is» Pay with exact cash amount	√		√	√	√			
	22. «reuse as is» Amount exceeds toll	√		√	√	√			
	23. «reuse as is» Amount is not sufficient	√		√	√	√			

Pay with Credit Card	24. «reuse as is» Payment authorized	√		√	√	√			
	25. «reuse as is» Payment not authorized	√		√	√	√			
Pay Operator	26. «reuse as is» Payment authorized	√		√	√				
	27. «reuse as is» Payment not authorized	√		√	√				
	28. «reuse as is» Cash sufficient	√		√	√				
	29. «reuse as is» Cash not sufficient	√		√	√				
	30. «reuse as is» Cannot pay	√		√	√				
	Total number of test specifications = 149	30	12	28	27	16	12	12	12

7.5.3 Coverage of All Use Case Scenarios in each Application of the AHTS SPL

In Phase V, feature-based test derivation was applied to select the test specifications for each application configuration in the AHTS SPL test plan. Table 32 shows the test specifications selected for each application of the AHTS SPL test plan in Table 9.

Next, the researcher used the CADeT tools to generate a test execution graph for each of the eight application configuration in the AHTS SPL test plan. A set of system test sequences was traced from each customized graph for each application of the AHTS SPL. These system test sequences covered all use case scenarios of the application.

Figure 34 shows an example of a test execution graph generated for TS2 (one of the applications from the AHTS SPL test plan). TS2 contains the “Transponder Exit Booth”, “Camera”, “Barrier”, and “Alarm” features, and the alternative “Variable Toll Charge” features (see Table 9 in Chapter 5). The graph in Figure 34 shows the order in which the test specifications can be executed for application TS2. With the exception of

the “start” and “end” nodes, each node in this graph represents a test specification selected during feature-based test derivation for TS2, and each edge an execution dependency between two test specifications.

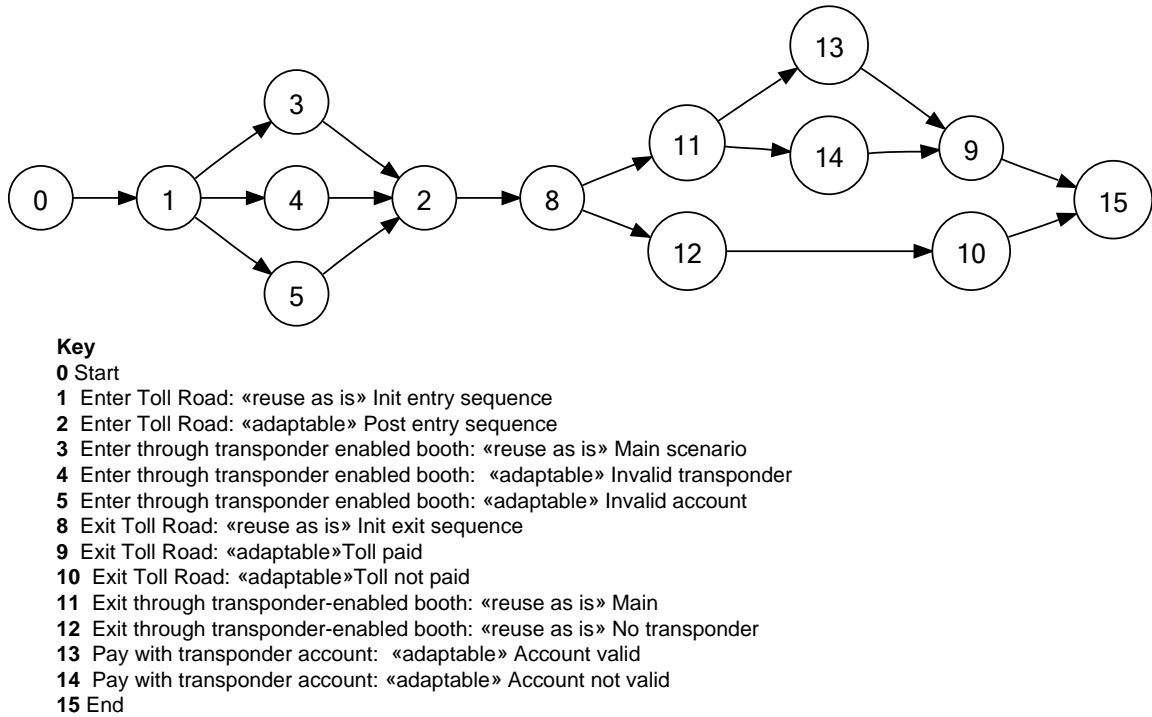


Figure 34 Test execution sequence graph for TS2¹

Next, the researcher created a test procedure document for each application of the AHTS SPL test plan. System test sequences were traced from the test execution graph, and were added to the test procedure document for an application, as described by Phase V in Chapter 5. Each test procedure document included all test specifications associated with the use case scenarios and features selected for the application.

¹ The picture of this graph was created using GraphViz

Table 33 shows a test procedure document for TS2 which includes all test specifications selected for TS2. For example, the system test sequence “System test 1” corresponds to the nodes “0-1-3-2-8-11-13-9” traced from the test execution sequence graph in Figure 35, and describes the situation where a vehicle with a valid transponder account enters a toll road through a transponder enabled entry booth and then exits the toll road through a transponder enabled exit booth.

Table 33 Test procedure for TS2

System test 1
1. Enter Toll Road: «reuse as is» Init entry sequence
3. Enter through transponder enabled booth: «reuse as is» Main scenario
2. Enter Toll Road: «adaptable» Post entry sequence
8. Exit Toll Road: «reuse as is» Init exit sequence
11. Exit through transponder-enabled booth: «reuse as is» Main
13. Pay with transponder account: «adaptable» Account valid
9. Exit Toll Road: «adaptable» Toll paid
System test 2
1. Enter Toll Road: «reuse as is» Init entry sequence
4. Enter through transponder enabled booth: «adaptable» Invalid transponder
2. Enter Toll Road: «adaptable» Post entry sequence
8. Exit Toll Road: «reuse as is» Init exit sequence
12. Exit through transponder-enabled booth: «reuse as is» No transponder
10. Exit Toll Road: «adaptable» Toll not paid
System test 3
1. Enter Toll Road: «reuse as is» Init entry sequence
5. Enter through transponder enabled booth: «adaptable» Invalid account
2. Enter Toll Road: «adaptable» Post entry sequence
8. Exit Toll Road: «reuse as is» Init exit sequence
11. Exit through transponder-enabled booth: «reuse as is» Main
14. Pay with transponder account: «adaptable» Account not valid
9. Exit Toll Road: «adaptable» Toll paid

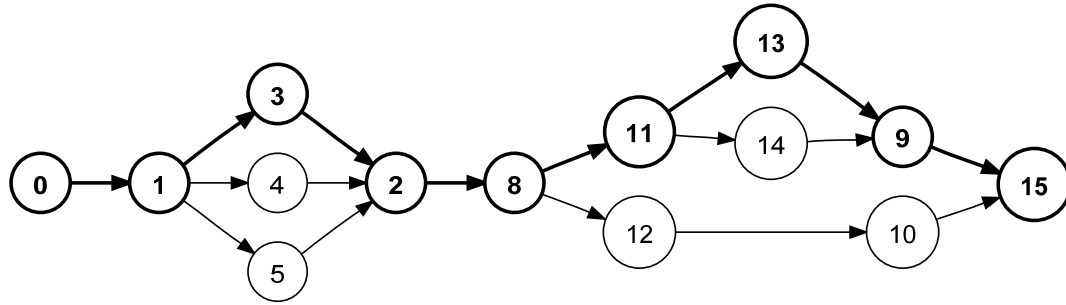


Figure 35 Execution sequence for system test 1

7.6 Application of Phases I-V: Creating and Customizing Test Specifications for the Banking System SPL

The researcher applied phase I of CADeT to create activity diagrams for each use case of the Banking System SPL in Table 29. Excerpts from these activity diagrams are shown in Appendix A.

7.6.1 Coverage of All Use Case Scenarios and All Features in Banking System SPL

Then, the researcher followed phase II of CADeT to trace paths from the activity diagrams for each use case scenario, and then map these paths to test specifications in decision tables. Excerpts from these decision tables are shown in Appendix A. The set of test specifications created for the Banking System SPL covered all use case scenarios of this SPL. A total of 23 test specifications were created to cover the 21 use case scenarios in the Banking System SPL.

Figure 36 shows the test specifications created to cover all use case scenarios in the Banking System SPL. Each scenario in Figure 36 was covered by at least one test

specification. Some scenarios, such as “Validate Pin: Main”, were covered by more than one test specification. The main scenario of the “Validate Pin” use case contained a loop (see the “Validate Pin” use case activity diagram in Figure 49 of Appendix A). The “Card is valid” and “Pin is valid” test specifications in Figure 36 were created to cover this loop. This enabled these two test specifications to be combined with other test specifications (e.g. “Pin is invalid less than max times”) to create different loop execution sequences (e.g. [Card is valid; Pin is invalid less than max times; Pin is valid]).

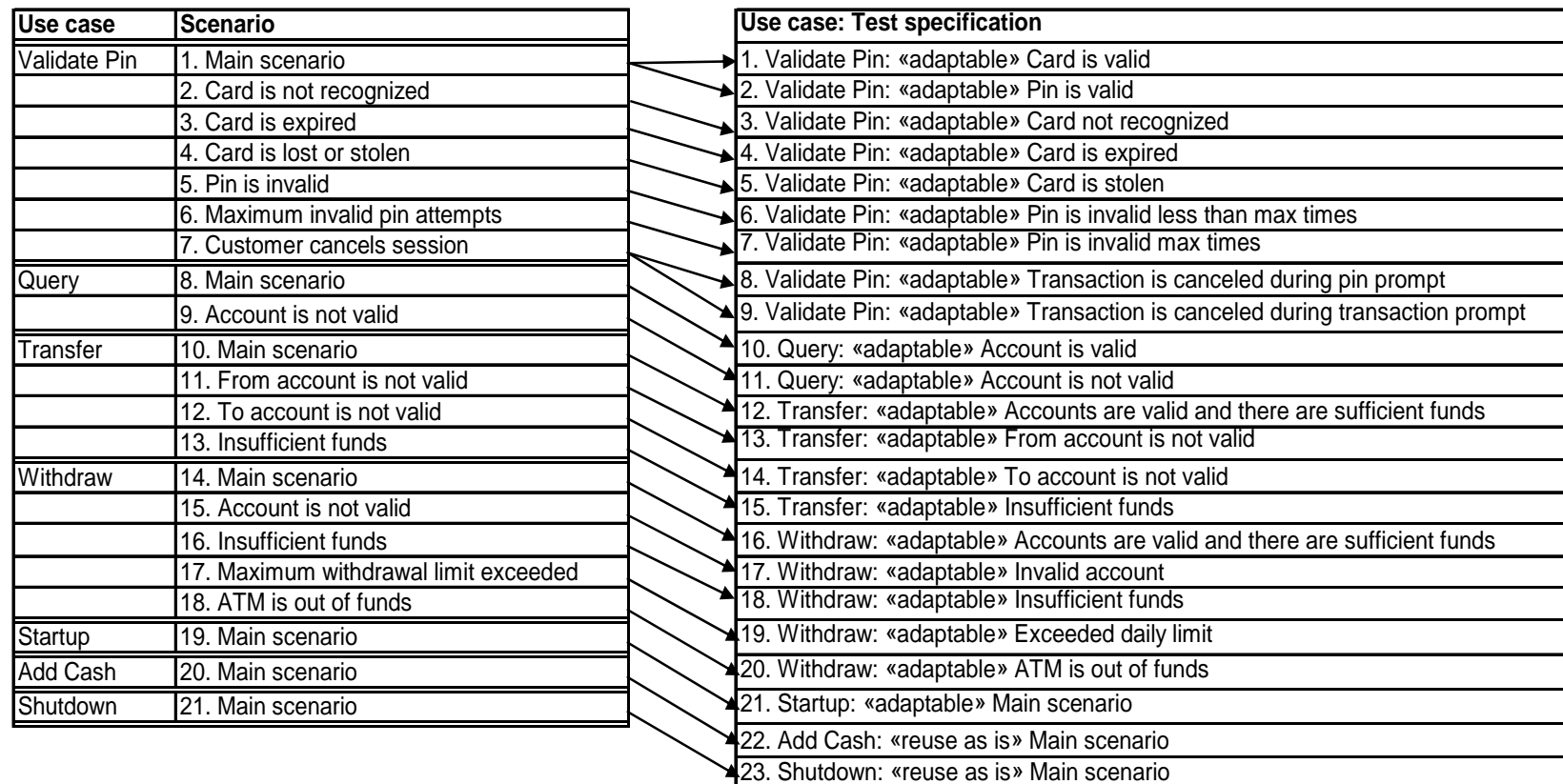


Figure 36 Relationship between use case scenarios and test specifications of Banking System SPL

The set of test specifications created for the Banking System SPL also covered all features of this SPL. Table 34 shows the relationship of each feature to the test specifications of the Banking System SPL. The “Feature” column lists the features of the Banking System SPL. Each feature in the Banking System SPL is associated with at least one test specification, and each test specification is associated with at least one feature. With the exception of the ATM Kernel, all features in the Banking System SPL are associated with a variation point parameter in one or more adaptable test specifications. Thus, all optional, alternative and parameterized features in the Banking System SPL represented a small granularity of variation in these test specifications (refer to explanation of fine-grained and coarse-grained variability in phase IV of Chapter 5).

Table 34 Features / test specifications relationships in Banking System SPL

Feature	Use case	Test specification	Variation point
«common» ATM Kernel	All	All	
«alternative» Spanish	All except Shutdown and Add cash	All except Shutdown: «reuse as is» Main scenario and Add cash: «reuse as is» Main scenario	vpLanguage
«alternative» French	All except Shutdown and Add cash	All except Shutdown: «reuse as is» Main scenario and Add cash: «reuse as is» Main scenario	vpLanguage
«alternative» English	All except Shutdown and Add cash	All except Shutdown: «reuse as is» Main scenario and Add cash: «reuse as is» Main scenario	vpLanguage
«alternative» Eject Expired Card	Validate Pin	4. «adaptable» Card is expired	vpExpiredCardAction
«alternative» Confiscate Expired Card	Validate Pin	4. «adaptable» Card is expired	vpExpiredCardAction
«optional» Call Police Action	Validate Pin	5. «adaptable» Card is stolen	vpStolenCardAction
«optional» Phone Branch Action	Validate Pin	5. «adaptable» Card is stolen	vpStolenCardAction
«optional» Alarm action	Validate Pin	5. «adaptable» Card is stolen	vpStolenCardAction
«parameterized» Pin Format	Validate Pin	2. «adaptable» Pin is valid	vpPinFormat
		4. «adaptable» Card is expired	vpPinFormat
		5. «adaptable» Card is stolen	vpPinFormat
		6. «adaptable» Pin is invalid less than max times	vpPinFormat
		7. «adaptable» Pin is invalid max times	vpPinFormat
		9. «adaptable» Transaction is canceled during transaction prompt	vpPinFormat
«parameterized» Pin Attempts	Validate Pin	6. «adaptable» Pin is invalid less than max times	vpPinAttempts
		7. «adaptable» Pin is invalid max times	vpPinAttempts
«parameterized» Greeting	Validate Pin	4. «adaptable» Card is expired	vpGreeting
		5. «adaptable» Card is stolen	vpGreeting
		8. «adaptable» Transaction is canceled during pin prompt	vpGreeting
		9. «adaptable» Transaction is canceled during transaction prompt	vpGreeting
	Startup	21. «adaptable» Main scenario	vpGreeting

7.6.2 Coverage of All Relevant Feature Combinations in Banking System SPL

Next, the researcher applied phase III of CADeT to analyze the test specifications of the Banking System SPL, and apply a feature-based test coverage criterion to select a set of representative application configurations to test, as described in Chapter 5.

The feature model of the Banking SPL was analyzed to calculate a total of 864 possible application configurations (as described in Appendix A). Next, the relationships of the features to test specifications were analyzed to determine the relevant feature combinations.

Table 35 shows the feature combinations associated with the test specifications of the Banking System SPL. The feature condition of the “ATM Kernel” feature is omitted from the analysis because it is always selected for any application derived from the SPL. The largest number of feature conditions in a relevant feature combination in Table 35 is two. Thus, at least a pair-wise feature based coverage criterion is needed to check the feature combinations described by these test specifications. A pair-wise feature based coverage criterion was applied using the Jenny tool (Jenkins 2005) to select 13 representative applications for the Banking System SPL. The test plan in Table 58 in Appendix A describes the features selected for each of these 13 application configurations.

Table 35 Relevant feature combinations in the Banking System SPL

Use case	Test specification	Feature combinations
Validate Pin	1. «adaptable» Card is valid	greeting*language
	2. «adaptable» Pin is valid	pinFormat + language

	3. «adaptable» Card not recognized	(greeting*language)
	4. «adaptable» Card is expired	(greeting*language) + pinFormat + (language*expiredCardAction)
	5. «adaptable» Card is stolen	(greeting*language) + pinFormat + callPoliceAction + phoneBranchAction + alarmAlarmAction
	6. «adaptable» Pin is invalid less than max times	pinFormat + language + pinAttempts
	7. «adaptable» Pin is invalid max times	pinFormat + language + pinAttempts
	8. «adaptable» Transaction is canceled during pin prompt	greeting*language
	9. «adaptable» Transaction is canceled during transaction prompt	(greeting*language) + pinFormat
Query	10. «adaptable» Account is valid	language
	11. «adaptable» Account is not valid	language
Transfer	12. «adaptable» Accounts are valid and there are sufficient funds	language
	13. «adaptable» From account is not valid	language
	14. «adaptable» To account is not valid	language
	15. «adaptable» Insufficient funds	language
Withdraw	16. «adaptable» Accounts are valid and there are sufficient funds	language
	17. «adaptable» Invalid account	language
	18. «adaptable» Insufficient funds	language
	19. «adaptable» Exceeded daily limit	language
	20. «adaptable» ATM is out of funds	language
Startup	21. «adaptable» Main scenario	greeting*language
Add Cash	22. «reuse as is» Main scenario	-
Shutdown	23. «reuse as is» Main scenario	-

Next, the researcher followed phase IV of CAdET to apply the parameterization variability mechanism to the Banking System SPL test specifications, in order to automate the configuration of these test specifications for each of the representative

applications of the Banking System SPL. A feature list was created for the Banking System SPL, as shown in Table 54 of Appendix A, and all decision tables of this SPL were parameterized, as illustrated by the sample decision table in Table 59 of Appendix A.

7.6.3 Coverage of All Use Case Scenarios in each Application of the Banking System SPL

In phase V, test specifications were selected and customized using the parameterization mechanism of CADeT for each of the 13 application configurations in the test plan of the Banking System SPL. All 23 test specifications of the Banking System SPL were selected and customized for each application in the test plan in Table 58.

The test procedure generator tool was used to generate a test execution graph and to create a test procedure document for each of the 13 application configurations in the Banking System SPL test plan in Table 58 (see Appendix A). A set of system test sequences was traced from each customized graph for each application of the Banking System SPL. These system test sequences covered all use case scenarios of the application.

The test execution graph of each application of the Banking System SPL contained 23 nodes, which corresponded to the 23 test specifications of the Banking System SPL, and 130 edges, which corresponded to execution dependencies between these test specifications. The test execution graph of each application was equivalent to the test execution graph of the Banking System SPL, because the common “ATM kernel”

feature was associated with all test specifications, while the variable features were associated with variation points in these test specifications.

A test procedure document was also created for each of the 13 application configurations in the test plan of the Banking System SPL. Each test procedure document included all test specifications associated with the use case scenarios and features selected for each of these applications.

7.6.4 Number of Applications Configured for each SPL

Table 36 compares the number of application configurations generated to cover the pair-wise feature combinations in the test plan of each SPL against the number of application configurations needed to cover all feature combinations in each SPL. Less application configurations were generated to cover the relevant (2-way) feature combinations in each SPL.

Table 36 Number of application configurations for each SPL

	AHTS SPL	Banking System SPL
All feature combinations	224	864
Pair-wise combinations	8	13

7.6.5 Comparison of Number of Test Specifications Created using CADeT with alternative approaches

Next, the number of test specifications created using CADeT for each SPL was compared against the number of test specifications needed to cover the same use case scenarios, features and feature combinations using two alternative approaches. The first

approach was a “no reuse” approach. In this approach, test specifications are created for each application in the SPL, without reusing test specifications created for other applications of that SPL. The number of test specifications that needed to be created using a “no reuse” approach was estimated by adding the number of test specifications created for each application in the test plan of each SPL.

The second approach was an approach that reused the coarse-grained functionality but did not reuse the fine-grained functionality described in the test specifications of a SPL. This approach is similar to the approach described in (Olimpiew and Gomaa 2005). With the “reuse coarse-grained functionality” approach, test specifications are created for each use case scenario and feature combination in the test plan of the SPL, and then reused “as is” by selecting test specifications that have already been created for another application of the SPL. The fine-grained functionality in the test specifications has to be manually customized for each application of the SPL. The number of test specifications created for the “reuse coarse-grained functionality” approach was estimated by adding the number of reuse “as is” test specifications to the number of variant test specifications generated for each application of the SPL.

The CADeT approach is used to automatically configure both the coarse-grained and fine-grained functionality in the test specifications of a SPL. The following describes the application of the “no reuse”, “reuse coarse-grained functionality” (some reuse) and “reuse fine-grained and coarse-grained functionality (CADeT) approaches to each SPL.

7.6.6 Number of Test Specifications Created for AHTS SPL

The CADeT approach was used to create a total of 30 test specifications for the AHTS SPL. Of these, 8 test specifications were identified as «adaptable», and the remaining 22 as «reuse as is». The test plan in Table 9, Chapter 5 described 8 application configurations that satisfy a pair-wise feature-based coverage criterion over the AHTS SPL.

Table 37 shows the number of test specifications created for the AHTS SPL using “no reuse”, “some reuse” and the CADeT approaches. The “some reuse” column describes the actual number of test specifications created to cover the pair wise combinations of the eight applications in the AHTS test plan. In Table 37, 30 test specifications were created using CADeT to cover all use case scenarios and pair-wise feature combinations in the AHTS SPL. Almost five times as many test specifications need to be created using the “no reuse” approach, and almost twice as many test specifications need to be created using the “some reuse” approach to cover the same use case scenarios and feature combinations of the AHTS SPL.

Table 37 Number of test specifications created for AHTS SPL

Use case or aggregate activity	Test Specification	Feature combinations	No reuse	Some reuse	CADeT
Enter Toll Road	2. «adaptable» Post entry sequence	barrier + trafficLight	8	4	1
Enter through transponder enabled booth	4. «adaptable» Invalid transponder	(transponderBooth * trafficLight) + (transponderBooth * alarm)	7	4	1
	5. «adaptable»: Invalid account	(transponderBooth * trafficLight)+ (transponderBooth * alarm)	7	4	1

Exit Toll Road	9. «adaptable» Toll paid	barrier + trafficLight	8	4	1
	10. «adaptable» Toll not paid	camera + barrier + trafficLight + alarm	8	7	1
Pay with transponder account	13. «adaptable» Account valid	transponderAccount * tollCharge	7	2	1
	14. «adaptable» Account not valid	(transponderAccount * trafficLight) + (transponderAccount * alarm)	7	4	1
Process ticket	16. «adaptable» Ticket recognized	ticketReader * tollCharge	4	2	1
*	22 «reuse as is» test specifications		93	22	22
Totals			149	52	30

7.6.7 Number of Test Specifications Created for Banking System SPL

The CADeT approach was used to create a total of 23 test specifications for the Banking System SPL. Of these, 21 test specifications were identified as «adaptable», and the remaining 2 as «reuse as is». The test plan in Table 58, Appendix A describes 13 application configurations that satisfy a pair-wise feature-based coverage criterion for the Banking System SPL.

Table 38 shows the number of test specifications created for the Banking System SPL using “no reuse”, “some reuse” and the CADeT approaches. The “some reuse” column describes the actual number of test specifications created to cover the pair wise combinations of the 13 applications in the Banking System SPL test plan.

Table 38 demonstrates that $23 * 13$, or 299 test specifications would have been created without reuse, and that that 127 test specifications (2 «reuse as is» test

specifications plus 125 variant test specifications) would have been created with “some reuse” for all of the applications in the test plan of the Banking System SPL. Using CAdET reduces that number to 23 test specifications.

Table 38 Number of test specifications created for Banking System SPL

Use case	Test specification	Feature combinations	No reuse	Some reuse	CAdET
Validate Pin	1. «adaptable» Card is valid	greeting * language	13	6	1
	2. «adaptable» Pin is valid	pinFormat + language	13	9	1
	3. «adaptable» Card not recognized	greeting * language	13	6	1
	4. «adaptable» Card is expired	(greeting * language) + pinFormat + (language * expiredCardAction)	13	11	1
	5. «adaptable» Card is stolen	(greeting * language) + pinFormat + callPoliceAction + phoneBranchAction + alarmAlarmAction	13	13	1
	6. «adaptable» Pin is invalid less than max times	pinFormat + language + pinAttempts	13	12	1
	7. «adaptable» Pin is invalid max times	pinFormat + language + pinAttempts	13	12	1
	8. «adaptable» Transaction is canceled during pin prompt	geeting * language	13	6	1
	9. «adaptable» Transaction is canceled during transaction prompt	(greeting * language) + pinFormat	13	11	1
Query	10. «adaptable» Account is valid	language	13	3	1
	11. «adaptable» Account is not valid	language	13	3	1
Transfer	12. «adaptable» Accounts are valid and there are sufficient funds	language	13	3	1

	13. «adaptable» From account is not valid	language	13	3	1
	14. «adaptable» To account is not valid	language	13	3	1
	15. «adaptable» Insufficient funds	language	13	3	1
Withdraw	16. «adaptable» Accounts are valid and there are sufficient funds	language	13	3	1
	17. «adaptable» Invalid account	language	13	3	1
	18. «adaptable» Insufficient funds	language	13	3	1
	19. «adaptable» Exceeded daily limit	language	13	3	1
	20. «adaptable» ATM is out of funds	language	13	3	1
Startup	21. «adaptable» Main scenario	greeting * language	13	6	1
Add Cash	22. «reuse as is» Main scenario	-	13	1	1
Shutdown	23. «reuse as is» Main scenario	-	13	1	1
	Totals		299	127	23

The next section describes a third and final study that was implemented to evaluate the remaining phases of CADeT and CADeT-SoC, which describe how to implement and apply each variability mechanism to configure the test specifications for a set of applications derived from a SPL, and then test these applications.

7.7 Evaluate Feasibility and Effort of Customizing Test Specifications and Testing Applications Using CADeT and CADeT-SoC

The third study evaluated the feasibility of applying the remaining phases of CADeT and CADeT-SoC to the Banking System SPL. Phases IV-VII of CADeT describe

how to apply a parameterization mechanism to the test specifications of a SPL, customize these test specifications for an application of the SPL, select test data for the customized test specifications, and then test the application. Phases IV-SoC and V-SoC of CAdET-SoC describe how to apply a separation of concerns variability mechanism instead of a parameterization mechanism to customize the test specifications of a SPL.

7.7.1 Description of Study

In this study, five graduate students (participants) from an advanced software design class took the role of subject matter experts. Participants were asked to apply phases IV-VII of CAdET and phases IV-SoC and V-SoC of CAdET-SoC to a set of decision tables and test specifications created by the researcher for the Banking System SPL. The instructions for these phases are in Chapters 5 and 6, and examples from the Banking System SPL are in Appendix A. The order in which these phases were applied by each participant is outlined below:

Customize test specifications using parameterization mechanism

1. Phase IV: Apply the parameterization variability mechanism to decision tables and test specifications of the Banking System SPL.
2. Phase V: Customize the decision tables and test specifications using the parameterization mechanism for two application configurations (assigned by researcher) from the Banking System SPL test plan.

Test two applications using customized test specifications

3. Phase VI: Select test data for the test specifications of these two applications.

4. Phase VII: Test two application implementations derived from a Banking System SPL implementation, which correspond to the assigned application configurations.

Customize test specifications using separation of concerns mechanism

5. Phase IV-SoC: Apply a separation of concerns variability mechanism to the decision tables and test specifications of the Banking System SPL.
6. Phase V-SoC: Configure test specifications for the same two application configurations using the separation of concerns variability mechanism.

Apply a pragmatic approach to customize test specifications

7. Apply a “No reuse”, or pragmatic approach to copy and modify the decision tables for two applications from the test plan.

All participants were required to participate in tutorial sessions. Tutorial sessions were given prior to the 1st, 2nd, 3rd and 5th activities. Additional meetings were held between the participant and researcher as the need arose. A week was scheduled for each activity, with some additional time for revisions. Participants were asked to enter the date, begin and end times for each activity in a time log template, as shown in Table 39. At the completion of a phase, the participant sent the results to the researcher. The researcher checked these results against the expected results, and then sent suggestions for revisions back to the participant. Besides the number of man-hours recorded in the time log, two other sources of quantitative and qualitative data were collected and analyzed in this study to evaluate the main hypothesis of this research. The quantitative data were the test results from the 4th activity, “Phase VII: Test test two application

implementations”, and the qualitative data were the results of a survey administered to the participants.

The following section describes the activities of each phase in more detail.

Table 39 Time log template

	Date ->		
I. Apply parameterization		Begin Time	End Time
Reading and understanding instructions			
Adding features to "Feature" worksheet			
Modifying Validate Pin decision table			
Modifying Query decision table			
Modifying Transfer decision table			
Modifying Withdraw decision table			
Modifying Startup decision table			
Modifying AddCash decision table			
Modifying Shutdown decision table			
	Date ->		
II. Customize decision tables using parameterization		Begin Time	End Time
Reading and understanding instructions			
Generating test specifications for <TS1>			
Creating system test sequences for <TS1>			
Generating test specifications for <TS2>			
Creating system test sequences for <TS2>			
	Date ->		
III. Application testing		Begin Time	End Time
Updating database for <TS1>			
Selecting inputs for <TS1>			
Installing <TS1>			
Running tests on <TS1> and recording results			
Updating database for <TS2>			
Selecting inputs for <TS2>			
Installing <TS2>			
Running tests on <TS2> and recording results			
	Date ->		
IV. Apply separation of concerns		Begin Time	End Time
Reading and understanding instructions			
Modifying Validate Pin decision table			
Modifying Query decision table			
Modifying Transfer decision table			
Modifying Withdraw decision table			
Modifying Startup decision table			
Exporting text files			
Installing and running SPLET			
Using SPLET to associate features with identifiers in Validate Pin			
Using SPLET to associate features with identifiers in Query			
Using SPLET to associate features with identifiers in Transfer			
Using SPLET to associate features with identifiers in Withdraw			
Using SPLET to associate features with identifiers in Startup			

	Date ->		
V. Customize decision tables using separation of concerns		Begin Time	End Time
Reading and understanding instructions			
Generating test specifications for SPL			
Exporting text files			
Using SPLET to customize test specifications for <TS1>			
Using SPLET to customize test specifications for <TS2>			
	Date ->	Begin Time	End Time
VI. Pragmatic approach			
Modifying decision tables for <TS1>			
Modifying decision tables for <TS2>			

1. Phase IV: Apply the parameterization variability mechanism to decision tables and test specifications during SPL engineering

Participants started out with seven base decision tables created from the activity diagrams of the Banking system SPL in the previous study. Each decision table was associated with a use case from the Banking system SPL use case model in Figure 46 in section A.1: Validate Pin, Query, Transfer, Withdraw, Startup, Add Cash and Shutdown. Each decision table contained one or more columns, which described 23 test specifications associated with the use case scenarios of the Banking System SPL in Figure 36. Some of these test specifications were tagged as adaptable to indicate that they needed to be customized for each application derived from the SPL. An example of a base decision table for the Validate Pin use case is shown in Table 56 in appendix A.3.

Each participant followed the instructions in section 5.5 to create the feature list in Table 54 and to associate features from the feature model to the adaptable test specifications in the decision tables using feature conditions. Participants replaced each adaptable test step with the variant or optional test steps associated with the selection of a

variable feature, and made these test steps configurable using Excel formulas. An example of a feature list for the Banking System SPL and parameterized decision table for Validate Pin use case is described in Table 59 in appendix A.5.

2. Phase V: Customize the decision tables and test specifications using the parameterization mechanism during application engineering

The participants were assigned two different applications from the Banking System SPL test plan in Table 58 in section A.4. An application implementation was derived for the corresponding application configurations from a Banking System SPL implementation (Vonteru 2001).

Each participant configured the feature list for the two assigned application configurations, which enabled test steps associated with selected features, and disabled test steps not associated with selected features in the decision tables. Then, the participant used CAdET's test specification generator tool to generate the test specifications document from customized decision tables for each of the two applications. Each column in the decision table became a test specification in the application's test specifications document. Next, the participant used CAdET's test procedure definition tool to order and compose system tests for each application, until the test procedure covered all test specifications of the application. A test procedure document and system test documents were generated for each application. An example of customizing the decision tables and generating these documents for an application derived from the Banking System SPL is described in appendix A.

4. Phase VI: Select test data for the test specifications of these two applications

Each participant followed the instructions of Phase VI in Chapter 5 to select test data for the database and system tests of each application. An example of selecting test data for the database and system tests of an application derived from the Banking System SPL is described in appendix A.

5. Phase VII: Test two application implementations

Each participant installed, executed, tested, and logged test results for two application implementations using the previously selected database and input values. An example of testing an application derived from the Banking System SPL is described in appendix A.8.

6. Phase IV_{SoC}: Apply the separation of concerns variability mechanism to decision tables and test specifications during SPL engineering

Each participant applied a separation of concerns variability mechanism to the same base decision tables created from the activity diagrams of the Banking system SPL. The participant followed the instructions for Phase IV-SoC in section 6.2 to modify the decision tables to use the separation of concerns variability mechanism, and to create the variable feature file for the Banking System SPL, as described in appendix A.9.

7. Phase V_{SoC}: Configure test specifications using separation of concerns variability mechanism during application engineering

Each participant customized the test specifications using the separation of concerns variability mechanism for the same two application configurations assigned by the researcher. The participant followed the instructions in section 6.2.4 to customize the test specifications for each application derived from the Banking System SPL using the separation of concerns variability mechanism, as described in section A.9 in Chapter 7.

8. Apply a pragmatic approach

Some participants also applied a pragmatic approach to customize the test specifications for two applications derived from the SPL. The pragmatic approach is the “No reuse” approach described earlier in the second study. Instead of applying a variability mechanism to automate the configuration of the test specifications for an application derived from the SPL, the participant copied the test specifications created for one application of the SPL, and then manually modified these test specifications for another application of the SPL.

7.7.2 Results of Applying Each Phase

Each participant attended tutorials, read detailed instructions for each phase and then followed the instructions for each activity. Before starting an activity in a phase, each participant entered the date and begin time for that activity, carried out the activity, and recorded the end time for that activity as shown in the time log template in Table 39.

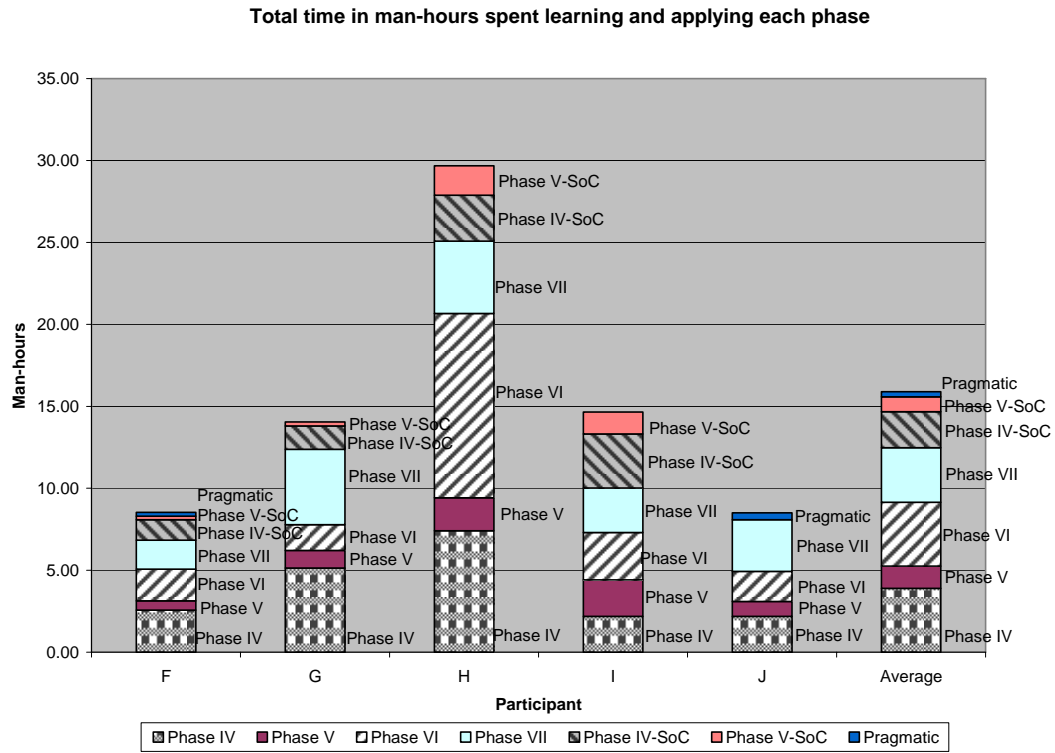


Figure 37 Total time in man-hours spent learning and applying each phase

Table 40 Total time in man-hours spent learning and applying each phase

	F	G	H	I	J	Average	Median	StdDev
Phase IV: Learn and apply parameterization variability mechanism	2.57	5.13	7.42	2.18	2.18	3.90	2.57	2.32
Phase V: Customize test specifications for two applications using the parameterization variability mechanism	0.57	1.07	2.00	2.23	0.92	1.36	1.07	0.72
Phase VI: Select test data for two applications	1.93	1.58	11.25	2.88	1.83	3.90	1.93	4.14
Phase VII: Test two applications	1.77	4.60	4.42	2.72	3.15	3.33	3.15	1.19
Phase IV-SoC: Learn and apply the separation of concerns variability mechanism	1.25	1.42	2.80	3.30	N/A	2.19	2.11	1.01
Phase V-SoC: Customize test specifications for two applications using separation of concerns variability mechanism	0.22	0.25	1.80	1.33	N/A	0.90	0.79	0.79
Apply pragmatic approach to customize test specifications for two applications	0.23	N/A	N/A	N/A	0.42	0.33	0.33	0.13
Total time	8.53	14.05	29.68	14.65	8.50	15.90		

The total time in man-hours for applying each Phase to the decision tables of the Banking System SPL is shown in a stacked column chart in Figure 37 for participants F, G, H, I and J. This chart also shows the average, median and standard deviation of the time in man-hours taken to apply each phase over all participants. The actual values and averages are shown in Table 40. In the table, an entry of “N/A” means that the participant did not perform the activity. This table shows that all participants were able to learn and apply Phases IV-VII of CADeT within an average time of 16 man-hours. All participants

who applied Phases IV-SoC and V-SoC of CADeT-SoC were also able to learn and apply these phases within the allotted time. On average, “Phase IV: Learn and apply parameterization variability mechanism” and “Phase VI: Select test data for two target systems” took the most time (about 4 man-hours), while “Phase V-SoC: Customize two target system test suites using separation of concerns variability mechanism” took the least amount of time (1 man-hour). Only two participants applied the pragmatic approach, which on average took less time than any of the phases (0.33 man-hours).

Several factors influenced the time in man-hours of applying each of these phases. The order in which the phases were applied affected the results. Initially, all participants were not familiar with the Banking System SPL requirement models and the decision tables, so the time spent learning Phase IV also included time spent understanding the Banking System SPL and the concept of decision tables. After a participant understood these concepts he was able to spend less time applying Phases IV-SoC and V-SoC, and the pragmatic approach. Another factor is that one of the participants took substantially more time than other participants to apply some of the phases, because of problems installing and using the tools. The following sections distinguish between the activities performed by each participant in each Phase.

7.8 Creating and Customizing Test Specifications Using Parameterization

7.8.1 Results for Phase IV: Apply the Parameterization Variability Mechanism

The total time in man-hours to learn and apply “Phase IV: Apply the parameterization variability mechanism” during SPL engineering is shown in Figure 38,

and the actual values, average, median and standard deviation are shown in Table 41. The time spent learning includes any time spent with the tutorials, reading and understanding instructions, and any additional meetings. The time spent applying Phase IV includes the time spent creating the feature list and modifying the decision tables to use the parameterization variability mechanism as described in Chapter 5.

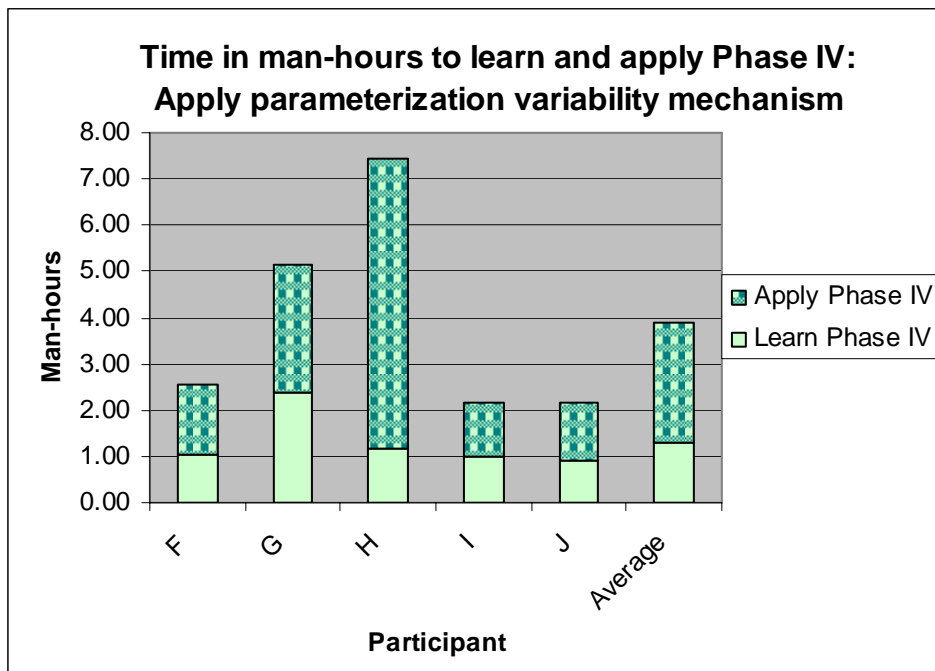


Figure 38 Time in man-hours to learn and apply Phase IV

Table 41 Time in man-hours to learn and apply Phase IV

	F	G	H	I	J	Average	Median	StdDev
Learn Phase IV	1.02	2.38	1.15	1.00	0.92	1.29	1.02	0.62
Apply Phase IV	1.55	2.75	6.27	1.18	1.27	2.60	1.55	2.14
Total time	2.57	5.13	7.42	2.18	2.18	3.90	2.57	2.32

Phase IV was a manual approach that was susceptible to a wide variety of differences in an individual's unique learning style, prior experience, and abilities. Participants G and H took a considerably longer time to learn and apply the method than the average time. Participant G encountered technical problems updating Microsoft Excel spreadsheet formulas on a Macintosh, while participant H had difficulty understanding the structure and purpose of the decision tables.

Each participant submitted a set of modified decision tables to the researcher, who compared the submitted results against expected results. If the submitted results were inconsistent with the expected results, the researcher requested revisions or additional meetings with the participant. Initially, all participants had trouble analyzing the impact of features in the decision tables, but were later able to correct these problems.

7.8.2 Results for Phase V: Customize Test Specifications for Two Applications using the Parameterization Variability Mechanism

The total time in man-hours to learn and apply "Phase V: Customize the test suites for two applications using the parameterization variability mechanism" during application engineering is shown in Figure 39, and the actual values, average, median and standard deviation are shown in Table 42. The time spent learning includes the time spent with the tutorials, reading and understanding instructions, and any additional meetings. The time spent applying Phase V to each application includes the time spent generating the test specifications document using the test specification generator tool and the time spent creating the test procedure document using the test procedure definition tool described in Chapter 5.

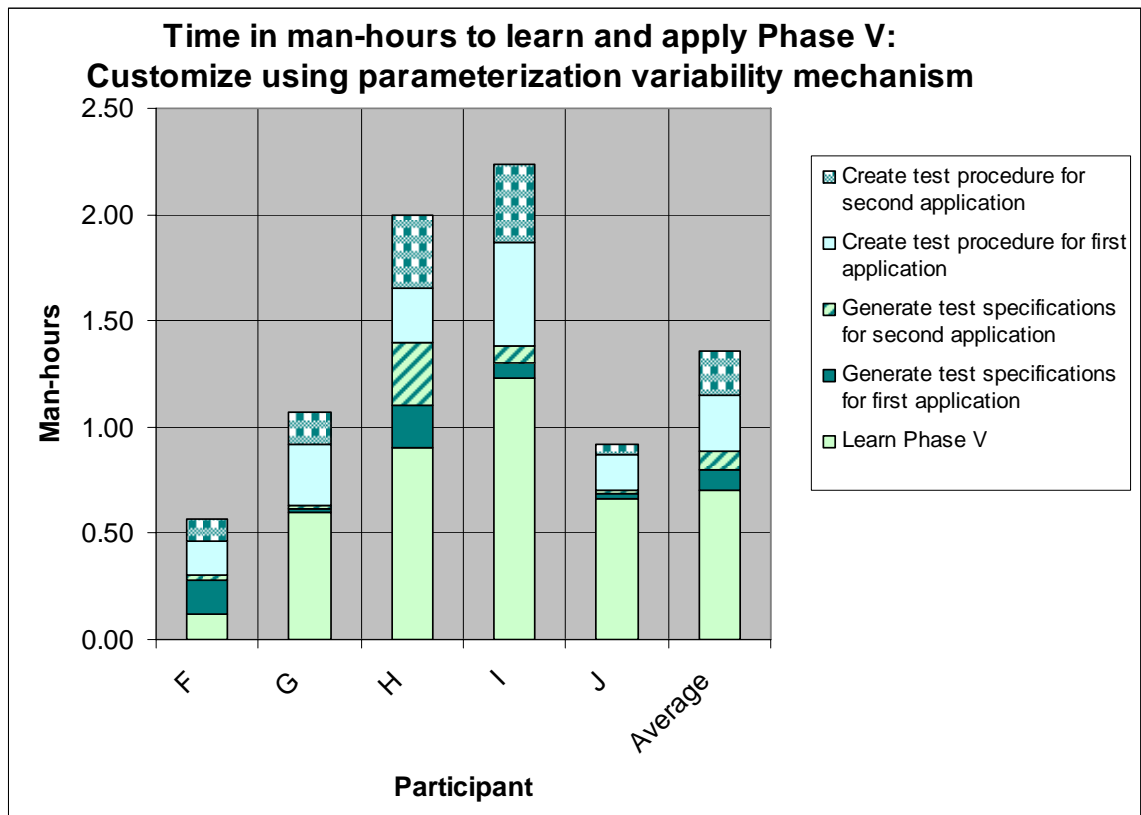


Figure 39 Time in man-hours to learn and apply Phase V

Table 42 Time in man-hours to learn and apply Phase V

	F	G	H	I	J	Average	Median	StdDev
Learn Phase V	0.12	0.60	0.90	1.23	0.67	0.70	0.67	0.41
Generate test specifications for first application	0.17	0.02	0.20	0.07	0.02	0.09	0.07	0.09
Generate test specifications for second application	0.02	0.02	0.30	0.08	0.02	0.09	0.02	0.12
Create test procedure for first application	0.17	0.28	0.25	0.48	0.17	0.27	0.25	0.13
Create test procedure for second application	0.10	0.15	0.35	0.37	0.05	0.20	0.15	0.15
Total time	0.57	1.07	2.00	2.23	0.92	1.36	1.07	0.72

Phase V was partly automated. Each participant used a test specification generator tool to generate the test specifications document and a test procedure definition tool to create the test procedure document for each application. Participants took from 0.02 to 0.20 man hours to generate the test specifications document for the first application, and from 0.02 to 0.30 man hours to generate the test specifications document for the second application using the test specification generator tool. Participants took from 0.17 to 0.48 man hours to create a test procedure document for the first application, and from 0.05 to 0.37 man hours to create a test procedure document for the second application using the test procedure definition tool.

All participants were able to understand and use the test specification generator tool to generate the test specifications document for each application. Some participants did not understand the purpose of the test procedure definition tool, and did not realize that a system test described the order in which test cases were going to be executed during system testing. All of these participants were able to correct these problems after the researcher explained and demonstrated the idea with an example.

7.9 Selecting Test Data for Customized Test Specifications

7.9.1 Results for Phase VI: Select Test Data for Two Applications

The total time in man-hours to learn and apply “Phase VI: Select test data for two applications” is shown in Figure 40, and the actual values, average, median and standard deviation are shown in Table 43. The time spent learning includes the time spent with the tutorials, reading and understanding instructions, and any additional meetings. The time

spent applying Phase VI to each application includes the time spent selecting test data for the database and system tests of each application as described in Chapter 5.

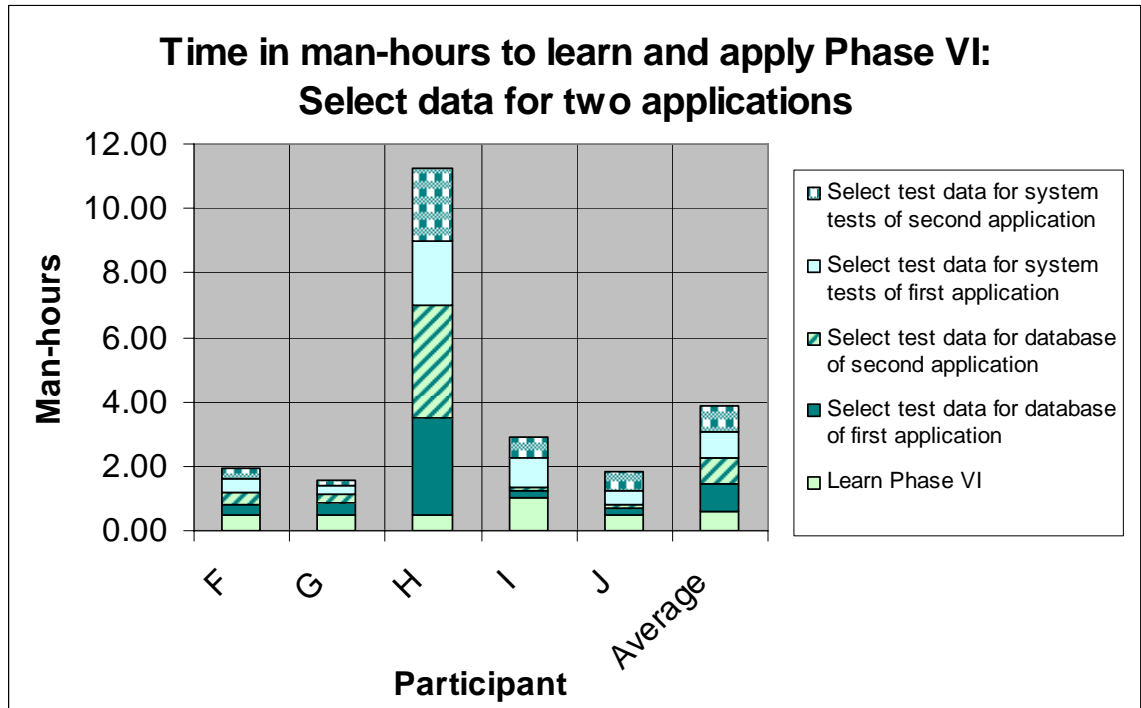


Figure 40 Time in man-hours to learn and apply Phase VI

Table 43 Time in man-hours to learn and apply Phase VI

	F	G	H	I	J	Average	Median	StdDev
Learn Phase VI	0.50	0.50	0.50	1.00	0.50	0.60	0.50	0.22
Select test data for database of first application	0.33	0.35	3.00	0.25	0.22	0.83	0.33	1.21
Select test data for database of second application	0.33	0.27	3.50	0.08	0.07	0.85	0.27	1.49
Select test data for system tests of first application	0.43	0.27	2.00	0.90	0.43	0.81	0.43	0.71
Select test data for system tests of second application	0.33	0.20	2.25	0.65	0.62	0.81	0.62	0.83
Totals	1.93	1.58	11.25	2.88	1.83	3.90	1.93	4.14

Phase VI was a manual approach that was susceptible to a wide variety of differences in an individual's unique learning style, prior experience, and abilities. Participants took from 0.22 to 3 man-hours to select test data for the database of the first application, and from 0.07 to 3.50 man hours to select test data for the database of the second application. Participants took from 0.27 man-hours to 2 man-hours to select test data for the system tests of the first application and from 0.20 to 2.25 man-hours to select test data for the system tests of the second application. Participant H took a considerably longer time to select test data than the other participants, and had some problems updating the time log in this phase. It is not clear whether the additional time was due to problems updating the time log or difficulty understanding and applying the instructions.

Almost all participants had difficulty selecting the test data for database tables and for the test specifications associated with alternative use case scenarios, such as "Validate

Pin: Customer uses a stolen card”. Some of the test data selected for these database tables and tests was incorrect because it did not satisfy the database rules or the predicates in the tests. However, most participants were able to correct these problems after the researcher pointed out the inconsistencies.

7.10 Results for Phase VII: Test Two Applications

Ten out of the thirteen applications described in the Banking System SPL test plan of Table 58 were derived from an implementation of the Banking System SPL and then tested by the participants of this study. Each participant was given two application implementations, which corresponded to the two application configurations earlier assigned to the participant.

7.10.1 Results of Executing the Tests

The total time in man-hours to learn and apply “Phase VII: Test two applications” is shown in Figure 41, and the actual values, average, median and standard deviation are shown in Table 44. The time spent applying Phase VII to test each application includes the time spent installing the application and the time spent executing each system test, observing the outcomes, and recording the results as described in Chapter 5.

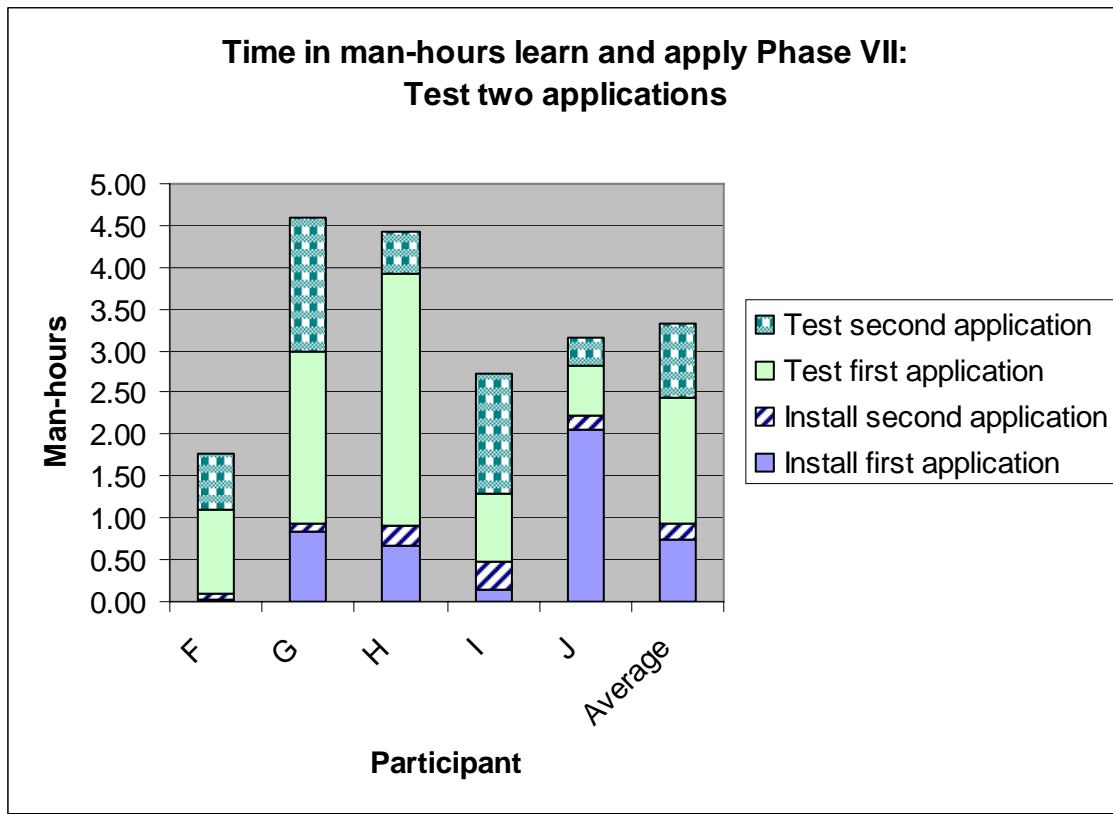


Figure 41 Time in man-hours to learn and apply Phase VII

Table 44 Time in man-hours to learn and apply Phase VII

	F	G	H	I	J	Average	Median	StdDev
Install first application	0.03	0.83	0.67	0.13	2.05	0.74	0.67	0.81
Install second application	0.07	0.10	0.25	0.33	0.17	0.18	0.17	0.11
Test first application	1.00	2.07	3.00	0.83	0.62	1.50	1.00	1.01
Test second application	0.67	1.60	0.50	1.42	0.32	0.90	0.67	0.57
Totals	1.77	4.60	4.42	2.72	3.15	3.33	3.15	1.19

Phase VII was also a manual approach that was susceptible to a wide variety of differences in an individual's unique learning style, prior experience, and abilities. Some of the variation in the time is due to unexpected problems encountered during testing. Participant J took considerably longer than the average to install the first application

(2.05 man-hours), because of a problem with an incompatible Java security policy.

Participant G installed a JDK version that was not compatible with the application, and

Participant H's computer crashed during testing for unknown reasons.

Table 45 summarizes the number of Passed, Failed and Invalid test results assigned by the participants for each test case in the system test plan of one of the applications of the Banking System SPL. A participant executed each test case against the assigned application, observed the actual outputs, and then compared the actual with the expected outputs. The "Passed" row describes the number of test cases executed for an application that were assigned a "Pass" test result (actual output = expected output). The "Failed" row describes the number of test cases executed for an application that were assigned a "Fail" result (actual output \neq expected output). The "Inconclusive" row describes the number of test cases executed for an application that were assigned an "Inconclusive" result (actual output \neq expected output). The "Not tested" result is the number of test cases in the system test plan that was not tested. The "Total tests executed" is the number of test specifications instances from the application's system test plan that were executed by a participant, and "Total in test plan" is the number of test cases in the application's test plan.

Table 45 Test results assigned by participants for test cases

	TS1	TS2	TS3	TS4	TS5	TS6	TS7	TS8	TS9	TS10
Participant	F	F	G	G	H	H	I	I	J	J
Passed	48	52	124	82	71	64	76	65	34	37
Failed	10	12	5	6	15	15	6	13	15	17
Inconclusive	4	0	0	0	1	0	2	0	0	0
Not tested	2	0	0	3	0	0	0	8	0	0
Total executed	62	64	129	88	87	79	84	78	49	54

Total in test plan	64	64	129	91	87	79	84	86	49	54
---------------------------	----	----	-----	----	----	----	----	----	----	----

The test data and test results were further verified by the researcher to make sure they were consistent with the predicates described in the test specifications, and to determine whether the failed test results described an actual fault. Table 46 shows the corrected test results. The “False positives” row describes the test results that the researcher determined to be incorrectly classified as Passed, and the “False negatives” row describes the test results that the researcher determined to be incorrectly classified as Failed. A test result was incorrectly classified if the inputs selected by the participant did not satisfy the execution conditions in the test case. The “Corrected passed”, “Corrected failed” and “Corrected total” describes the corrected test results.

Test data selection and test execution are manual processes in CADeT and depend in part on an individual’s ability to select test data to satisfy the predicates and to calculate the expected outputs from this data. On average, each participant was able to execute about 80% of all test cases in the system test plan of an application.

Table 46 Corrected test results

	TS1	TS2	TS3	TS4	TS5	TS6	TS7	TS8	TS9	TS10
Participant	F	F	G	G	H	H	I	I	J	J
False positives	1	2	29	15	3	10	15	6	6	4
Corrected passed	47	50	95	67	68	54	61	59	28	33
False negatives	1	0	3	1	10	10	5	9	12	16
Corrected failed	9	12	2	5	5	5	1	4	3	1
Total tests	64	64	129	91	87	79	84	86	49	54
Corrected total	62	62	97	75	74	59	64	71	31	34

7.10.2 Coverage of All Use Case Scenarios and All Features

Table 47 shows the features associated with each of the ten applications of the Banking System SPL. Table 48 shows the number of test cases (derived from each test specification) that were executed against each application of the Banking System SPL. These test cases executed all use case scenarios of the Banking System SPL, and also covered the feature combinations shown in Table 47. Most of the relevant feature combinations of the Banking System SPL in Table 35, such as greeting*language = {(enhanced, english), (standard, english), (enhanced, french), (standard, french), (enhanced, spanish), (standard, spanish)} are covered by these ten applications.

Table 47 Features associated with applications of the Banking System SPL

Feature condition	Feature selections	TS1	TS2	TS3	TS4	TS5	TS6	TS7	TS8	TS9	TS10
ATMKernel	T	T	T	T	T	T	T	T	T	T	T
language	{eng, fre, spa}	eng	fre	spa	fre	eng	spa	eng	fre	spa	eng
expiredCardAction	{conf, eject}	conf	eject	eject	conf	eject	conf	conf	conf	eject	conf
callPoliceAction	{T, F}	T	F	F	T	F	T	F	T	T	T
phoneBranchAction	{T, F}	F	T	F	T	T	F	F	T	T	F
alarmAction	{T, F}	F	T	T	F	T	F	F	T	F	F
pinFormat	{3, 4, 10}	3	4	10	10	3	4	4	3	3	10
pinAttempts	{1, 3, 5}	1	3	5	5	1	3	5	5	1	3
greetingPrompt	{enh, sta}	enh	sta	enh	sta	sta	enh	sta	enh	sta	sta

Table 48 Number of test cases executed against each application

Use case: Test specifications	TS1	TS2	TS3	TS4	TS5	TS6	TS7	TS8	TS9	TS10	Total test cases
1. Validate Pin: «adaptable» Card is valid	13	13	21	17	15	14	17	19	12	12	153
2. Validate Pin: «adaptable» Pin is valid	11	11	16	14	13	12	14	18	11	11	131
3. Validate Pin: «adaptable» Card not recognized	1	1	3	2	2	1	2	1	1	1	15
4. Validate Pin: «adaptable» Card is expired	1	1	3	2	1	1	2	1	1	1	14
5. Validate Pin: «adaptable» Card is stolen	1	1	5	3	2	1	4	2	1	1	21
6. Validate Pin: «adaptable» Pin invalid less than max times	1	1	17	5	1	3	6	5	0	2	41
7. Validate Pin: «adaptable» Pin invalid max times	1	1	5	2	1	1	1	1	1	1	15
8. Validate Pin: «adaptable» Transaction is canceled during pin prompt	1	1	2	1	1	1	2	2	1	1	13
9. Validate Pin: «adaptable» Transaction is canceled during transaction prompt	1	1	6	5	1	1	4	4	1	1	25
10. Query: «adaptable» Account is valid	1	1	2	2	3	2	5	5	1	1	23
11. Query: «adaptable» Account is not valid	1	1	2	1	1	1	1	2	1	1	12

12. Transfer: «adaptable» Accounts are valid and there are sufficient funds	1	1	3	2	1	1	1	2	1	1	14
13. Transfer: «adaptable» From account is not valid	1	1	0	1	1	1	1	1	1	1	9
14. Transfer: «adaptable» To account is not valid	1	1	1	1	1	1	1	1	1	1	10
15. Transfer: «adaptable» Insufficient funds	1	1	1	1	1	1	1	1	1	1	10
16. Withdraw: «adaptable» Accounts are valid and there are sufficient funds	1	1	1	1	1	1	1	1	1	1	10
17. Withdraw: «adaptable» Invalid account	1	1	2	1	1	1	0	1	1	1	10
18. Withdraw: «adaptable» Insufficient funds	1	1	1	1	1	1	1	1	1	1	10
19. Withdraw: «adaptable» Exceeded daily limit	1	1	1	2	1	1	1	2	1	1	12
20. Withdraw: «adaptable» ATM is out of funds	1	1	2	1	1	1	1	1	1	1	11
21. Startup: «adaptable» Main scenario	20	20	26	24	19	17	15	13	7	9	170
22. Add Cash: «reuse as is» Main scenario	1	1	4	1	1	1	2	1	1	1	14
23. Shutdown: «reuse as is» Main scenario	1	1	5	1	17	14	1	1	1	2	44
Total test cases	64	64	129	91	87	79	84	86	49	54	787

7.10.3 Faults Discovered

Table 49 describes the faults found in each of the ten applications derived from the Banking System SPL by the participants this study. A total of 13 faults were discovered during testing. Most of these faults (11 out of 13) were found while executing test specifications of alternative use case scenarios associated with common features of the Banking System SPL (such as transferring money from an invalid account).

The two other faults were feature related. Fault *d* in Table 49 is a feature interaction error in which a transaction receipt was printed in English when the T2 application was configured in the French language. This fault was present in all applications of the Banking System SPL that were configured with either the French or Spanish language, but was not detected by all participants. The second feature-related fault *b* was found during the execution of the “5. Validate Pin: Card is stolen” test case in application TS4: “The call police and phone branch actions do not execute after the card is determined to be stolen.” The command to configure the stolen card actions had been inadvertently commented out in the initialization file for application TS4.

The approach each participant followed to select test data and to test the applications of the Banking System SPL is described in Phases VI-VII in Chapter 5. Appendix A has additional examples that illustrate how Phases VI-VII in Chapter 5 were applied to the Banking System SPL.

Table 49 Faults found in the applications of the Banking System SPL

Test specification	TS1	TS2	TS3	TS4	TS5	TS6	TS7	TS8	TS9	TS10	Description of fault
3. «adaptable» Validate Pin: Card not recognized	X	X	X	X	X						a. Entering a non-empty string for a card id will make the application recognize the non-existent card
5. «adaptable» Validate Pin: Card is stolen				X							b. The call police and phone branch actions do not execute after the card is determined to be stolen, because application TS4 was configured incorrectly.
8. «adaptable» Validate Pin: Transaction is canceled during pin prompt		X									c. Transaction canceled message is not shown
10. «adaptable» Query: Account is valid		X									d. The receipt is not printed in French in TS2. All receipts are printed in English in all applications.
11. «adaptable» Query: Account is not valid		X				X	X				e. A receipt is printed for a phony account
12. «adaptable» Transfer: Accounts are valid and there are sufficient funds	X							X			f. Application showed a blank screen after an amount > 20 was transferred between two accounts
13. «adaptable» Transfer: From account is not valid	X	X			X						g. A bogus account was associated to a debit card in the database. The application froze on the wait screen during a transfer from this bogus account.
14. «adaptable» Transfer: To account is not valid	X	X									h. Transferred \$20 to a bogus account associated with the debit card in the database. Application deducted the funds from the first account, showed a successful transaction and printed a receipt.
14. «adaptable» Transfer: To account is not valid						X					i. Transferred \$80 to a bogus account associated with a debit card in the database. Application froze

15. «adaptable» Transfer: Insufficient funds	X	X			X			X			j. Application showed a blank screen after an amount > 20 is transferred between two accounts
16. «adaptable» Withdraw: Accounts are valid and there are sufficient funds		X						X	X	X	k. Application showed a blank screen after an amount > 20 is withdrawn from an account
17. «adaptable» Withdraw: Invalid account	X										l. Withdrew \$20 from a bogus account associated with the debit card in the database. Application showed "Insufficient funds" message instead of "Invalid account" error.
17. «adaptable» Withdraw: Invalid account		X			X	X					m. Application showed a blank screen after an amount > 20 is withdrawn from an invalid account
18. «adaptable» Withdraw: Insufficient funds	X	X		X							n. Application showed a blank screen after an amount > 20 is withdrawn from an account with insufficient funds
19. «adaptable» Withdraw: Exceeded daily limit	X	X							X		o. Application showed a blank screen after an amount > 20 is withdrawn from an invalid account
20. «adaptable» Withdraw: ATM is out of funds	X	X		X							p. Application showed a blank screen after an amount > 20 is withdrawn from an invalid account

7.11 Applying Separation of Concerns

7.11.1 Results for Phase IV_{SoC}: Learn and Apply Separation of Concerns

Variability Mechanism to Test Specifications

The total time in man-hours to learn and apply “Phase IV-SoC: Learn and apply the separation of concerns variability mechanism” during SPL engineering is shown in Figure 43, and the actual values, average, median and standard deviation are shown in Table 50. The time spent learning includes the time spent with the tutorials, reading and understanding instructions, and any additional meetings. The time spent applying phase IV-SoC includes the time spent adding insertion points to the decision tables and creating the variable feature file as described in Chapter 6.

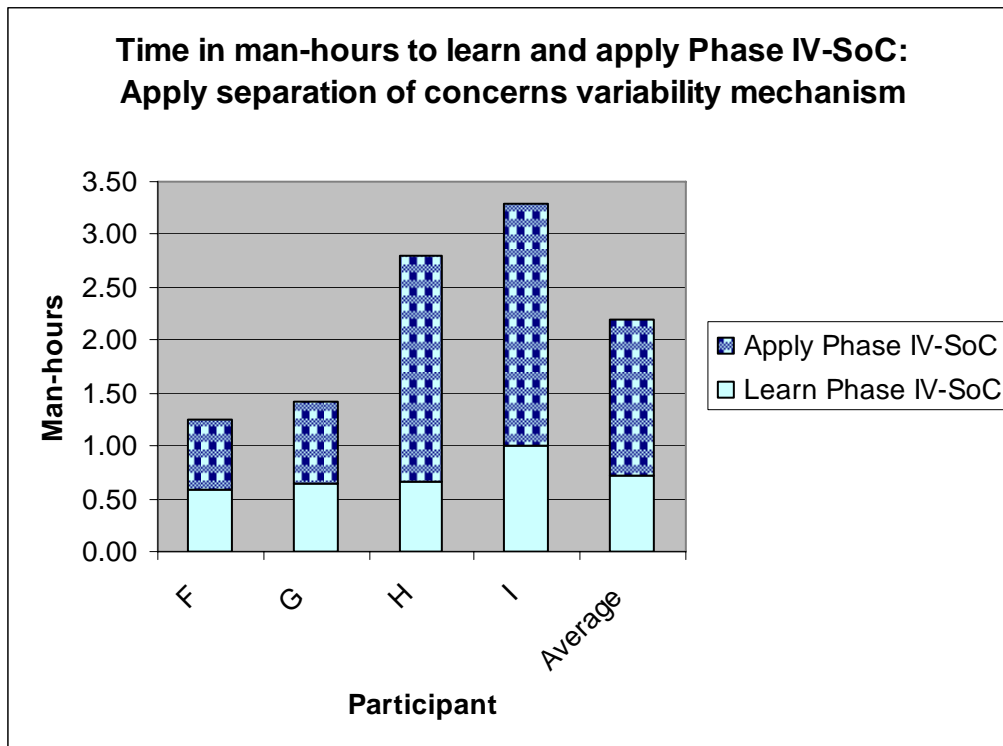


Figure 42 Time in man-hours to learn and apply Phase IV-SoC

Table 50 Time in man-hours to learn and apply Phase IV-SoC

	F	G	H	I	Average	Median	StdDev
Learn Phase IV-SoC	0.58	0.65	0.67	1.00	0.73	0.66	0.19
Apply Phase IV-SoC	0.67	0.77	2.13	2.30	1.47	1.45	0.87
Totals	1.25	1.42	2.80	3.30	2.19	2.11	1.01

Phase IV_{SoC} was partly automated. Participants H and I took a considerably longer time to apply the method than the average time. Participant H had difficulty using the SPLET tool (Saleh and Gomaa 2005) to generate the variable feature file but it is not known why participant I took a longer than average time.

Each participant submitted a set of modified decision tables and a variable feature file to the researcher, who compared the submitted results against expected results.

Initially, participants G, H, and I were missing descriptions of the optional and variant test steps associated with the expired card and stolen card actions. After the researcher requested revisions to the file, all participants were able to correct these problems.

7.11.2 Results for Phase V_{SoC}: Customize Test Specifications for Two Applications using the Separation of Concerns Variability Mechanism

The total time in man-hours to learn and apply “Phase V-SoC: Customize test specifications for two applications using the separation of concerns variability mechanism” during application engineering is shown in Figure 43 and the actual values, average, median and standard deviation are shown in Table 51. The time spent learning includes the time spent with the tutorials, reading and understanding instructions, and any additional meetings. The time spent applying Phase V-SoC to customize the test specifications for each application includes the time spent running the SPLET tool (Saleh and Gomaa 2005) to weave the variable test steps with the test specifications as described in Chapter 6.

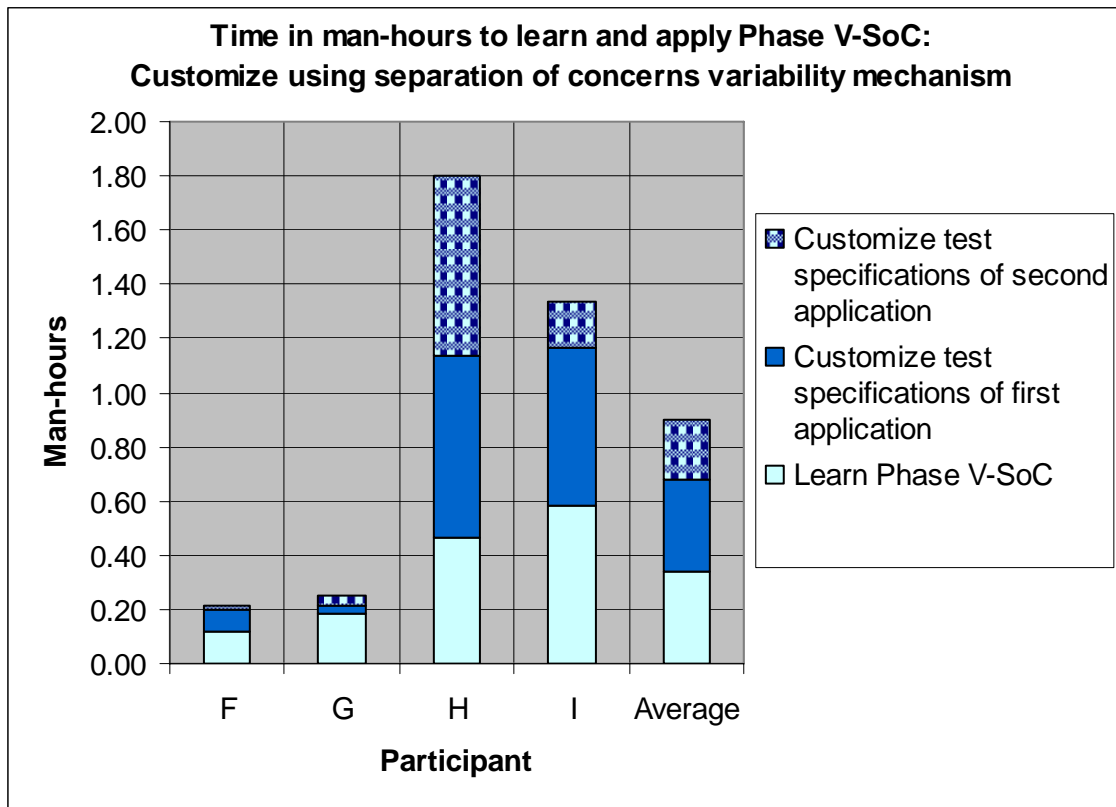


Figure 43 Time in man-hours to learn and apply Phase V-SoC

Table 51 Time in man-hours to learn and apply Phase V-SoC

	F	G	H	I	Average	Median	StdDev
Learn Phase V-SoC	0.12	0.18	0.47	0.58	0.34	0.33	0.22
Customize test specifications of first application	0.08	0.03	0.67	0.58	0.34	0.33	0.33
Customize test specifications of second application	0.02	0.03	0.67	0.17	0.22	0.10	0.30
Totals	0.22	0.25	1.80	1.33	0.90	0.79	0.79

Phase V-SoC was automated using the SPLET tool. Table 51 shows that participants H and I took a considerably longer time to customize the test specifications for the first and second applications, because of problems using the SPLET tool.

Participant H's feature file was formatted incorrectly, causing the SPLET tool to crash; and participant I found a bug during code weaving in the SPLET tool. Both participants were able to resolve these problems after consulting with the researcher.

7.12 Applying a Pragmatic Approach

7.12.1 Results of Applying Pragmatic Approach to Create Test Specifications for Two Applications

The total time in man-hours to learn and apply the pragmatic approach to create test specifications for two applications is shown in Figure 44 and Table 52. Because of time constraints, only two participants were able to apply the pragmatic approach. It took less time for each participant to learn and apply the pragmatic approach (0.33 man-hours) as compared with learning and applying the parameterization mechanism (3.9 man-hours in Phase IV, and 1.36 man-hours in Phase V in Table 40) and learning and applying the separation of concerns mechanism (2.19 man-hours in Phase IV-SoC, and 0.9 man-hours in Phase V-SoC in Table 40). However, several factors biased the results in favor of the pragmatic approach. The pragmatic approach was applied after the participants had become familiar with the test specifications of the Banking System SPL. Also, uncontrolled external factors (such as using the CADeT tools on a Mac) caused some participants spent substantially more time than other participants learning and applying CADeT.

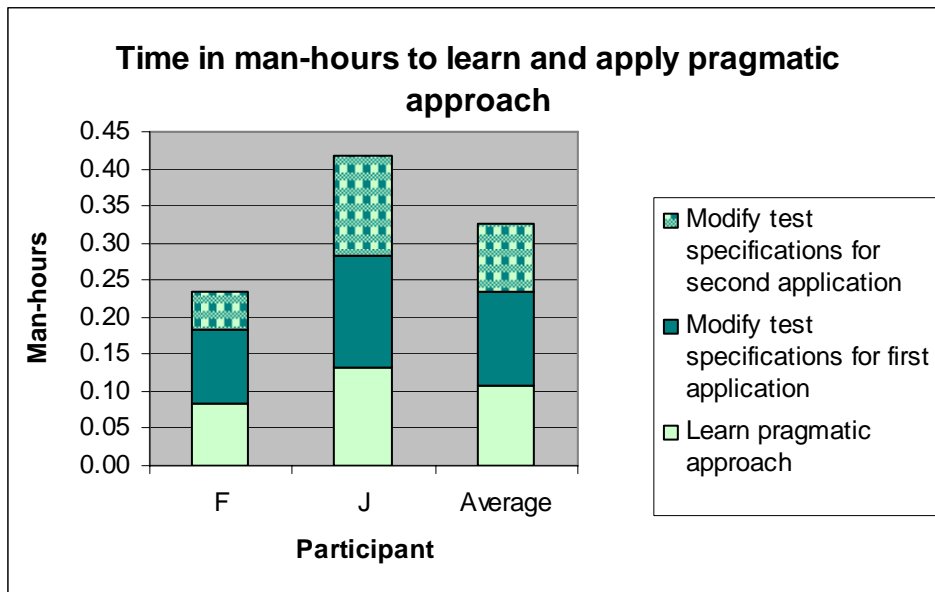


Figure 44 Time in man-hours to learn and apply pragmatic approach

Table 52 Time in man-hours to learn and apply pragmatic approach

	F	J	Average	Median	StdDev
Learn pragmatic approach	0.08	0.13	0.11	0.11	0.04
Modify test specifications for first application	0.10	0.15	0.13	0.13	0.04
Modify test specifications for second application	0.05	0.13	0.09	0.09	0.06
Totals	0.23	0.42	0.33	0.33	0.13

7.13 Results of Questionnaire

Table 53 shows the results of the questionnaire given to each participant after the applied project was completed. Most participants thought that applying the parameterization customization method was the most difficult part of the project. The participants that tried out the pragmatic approach thought that this was the easiest part of the project.

Table 53 Results of Questionnaire

	F	G	H	I	J
Experience with UML modeling methods	Two classes and some industry	At least one class	Three classes	Two classes	Two classes
Experience with software testing	Almost none	One class and some industry	Some industry	None	Almost none
Phases worked on	All	All except pragmatic	All except pragmatic	All except pragmatic	All except separation of concerns
Easiest phase	Pragmatic	Separation of concerns	Parameterization	Application testing	Pragmatic
Most difficult phase	Parameterization	Parameterization	Separation of concerns	Parameterization	Parameterization
How can project be improved?	Explain outputs	State all system requirements before testing system	Clarify error messages in SPLET	More detailed instructions	More detailed instructions

7.14 Interpretation of 3rd Study Results

All participants were able to apply the parameterization mechanism to automate the customization of the test specifications of the Banking System SPL for two applications, and then test these applications. All participants were also able to apply the separation of concerns variability mechanism to automate the customization of the test specifications of the Banking System SPL for the same two applications.

However, several factors affected the validity of the questionnaire and time log results results, such as the order in which the phases were applied, and uncontrolled external factors. These factors should be addressed in future studies to determine when a

break-even point is reached, where the benefits of automatically deriving test specifications for a set of applications from the SPL exceeds the initial effort spent implementing the variability mechanism.

7.15 Comparison of CAdET with Previous Research on SPL Testing Methods

Existing functional testing methods for SPLs such as PLUTO (Bertolino and Gnesi 2003), Nebut's method (Nebut, Fleurey et al. 2003), ScenTED (Reuys, Kamsties et al. 2005), and Geppert's method (Geppert 2004) address the problem of systematic reuse, but do not provide a feature-based approach of selecting representative application configurations to test. In some SPLs the set of possible application configurations is not entirely predetermined, and it is not feasible to test all possible application configurations.

CAdET can be used to apply a feature-based approach to select representative application configurations to test for a SPL. This reduces the number of representative application configurations to test, while covering all features, all use case scenarios and all relevant feature combinations of the SPL.

Other methods, such as Scheidemann's (Scheidemann 2006) address the problem of selecting representative application configurations to test from the potentially large configuration space of an SPL, but do not provide a method for creating reusable functional test specifications that can be configured during feature-based test derivation for an application derived from a SPL. A method of creating reusable test specifications is necessary to reduce the effort needed to create test specifications for each representative application of a SPL.

CADeT can be used to create reusable test specifications for a SPL, which can be customized during feature-based test derivation for an application of the SPL. Besides combining a feature-based coverage criterion with a use case scenario-based coverage criterion, CADeT differs from other functional testing methods for SPLs by distinguishing between two variability mechanisms to configure the fine-grained variability in the test specifications of a SPL. The parameterization variability mechanism of CADeT or separation of concerns variability mechanism of CADeT-SoC can be used to automate the configuration of the fine-grained variability in the test specifications of a SPL.

CADeT is flexible and can be incorporated with other functional testing methods for SPLs. For instance, a feature model and feature to use case relationship table can be used to analyze the relationships of features to activity diagrams in ScenTED (Reuys, Kamsties et al. 2005). Alternatively, the method of creating reusable test specifications from activity diagrams in CADeT can be integrated with another feature-oriented method of selecting representative configurations to test, such as the method described by (Scheidemann 2006).

8 Conclusions

The approaches described in this research builds on the ideas of systematically reusing and configuring the test specifications of a SPL and of selecting representative configurations to test. CAdET and CAdET-SoC are test design methods for SPLs that combine a use case scenario-based test coverage criterion to provide functionality coverage together with a feature-based test coverage criterion to provide variability coverage of a SPL. In these test design methods, features from a feature model are associated with activity diagrams created from the use case descriptions of a SPL. Reusable test specifications are traced from the use case activity diagrams and described in decision tables. The relationships of features to activity diagrams are also portrayed in decision tables, and a feature-based test coverage criterion is applied to select representative application configurations for the SPL. Next, the reusable test specifications are selected and customized for each application configuration, and then used to test the corresponding application implementation.

CAdET uses a parameterization variability mechanism to select test specifications, and then customize the test specifications during feature-based test derivation for an application derived from the SPL. However, implementing the parameterization mechanism requires additional effort to customize the test specifications of a SPL with many variation points, when these variation points have many variable test

steps that are repeated across several use cases. CADeT-SoC extends CADeT by applying a separation of concerns variability mechanism, rather than parameterization, to customize the test specifications during feature-based test derivation. Separation of concerns is more suitable than parameterization for customizing the test specifications of a SPL with many variation points repeated across several use cases. Using CADeT-SoC reduced the number of variable test steps that needed to be defined to realize the variation points in each SPL.

The feasibility of the CADeT and CADeT-SoC methods were evaluated in three studies on two SPLs. The results of these studies show that CADeT and CADeT-SoC can be used to create reusable test specifications to satisfy use case-based and feature-based coverage criteria for these two SPLs. The feature model of a SPL, and the relationships of features to test specifications were analyzed to determine the relevant feature combinations, and a feature-based coverage criterion was applied to reduce the number of application configurations to test. Test specifications were created to cover all use case scenarios, features, and relevant feature combinations in each SPL. Using CADeT reduced the number of test specifications to satisfy these criteria, as compared to using two alternative approaches.

8.1 Contributions

The main contribution of this research is a feature-based test design method that can be used to create reusable and functional test specifications from the use case and feature models of a SPL, and then customize these test specifications during feature-

based application derivation for any application from that SPL. The other contributions are described next:

8.1.1 Application of a Feature-Based Coverage Criterion with a Use Case-Based Coverage Criterion

Previous research (McGregor 2001; Bertolino and Gnesi 2003; Nebut, Fleurey et al. 2003; Geppert 2004; Reuys, Kamsties et al. 2005; Scheidemann 2006) has not applied a feature-based coverage criterion together with a use case-based coverage criterion for a SPL. CADeT and CADeT-SoC apply both criteria to a SPL, which reduces the number of application configurations to test and the number of test specifications created for each SPL.

8.1.2 Distinguishing Between Coarse-Grained and Fine-Grained Functional Variability

Previous research (McGregor 2001; Bertolino and Gnesi 2003; Nebut, Fleurey et al. 2003; Geppert 2004; Reuys, Kamsties et al. 2005; Scheidemann 2006) has not distinguished between the representation and binding times of coarse-grained and fine-grained variability in the functional requirements of a SPL. This research distinguished between the representation of coarse-grained and fine-grained functional variability in the activity diagrams, and between the binding times of coarse-grained and fine-grained functional variability during feature-based test derivation. Delaying the binding of the fine-grained variability during feature-based test derivation reduced the number of test specifications that needed to be created for each SPL.

8.1.3 Using Separation of Concerns to Customize the Test Specifications of a SPL

Previous research has used the parameterization variability mechanisms to customize functional test specifications for a SPL (Bertolino and Gnesi 2003; Kamsties, Pohl et al. 2003; Nebut, Fleurey et al. 2003; Geppert 2004). Pesonen et al suggested using separation of concerns to automatically configure the test specifications of a SPL (Pesonen, Katara et al. 2005), but did not apply this mechanism to customize functional test specifications for a SPL.

This research described and applied parameterization and separation of concerns variability mechanisms to customize reusable test specifications during feature-based application derivation for a set of applications of a SPL. These two mechanisms were applied and evaluated on two SPLs. Mazen Saleh's method and tool (Saleh and Gomaa 2005) was extended in Static Customization of Test specifications (SCT) to apply separation of concerns to the test specifications of a SPL (see Chapter 6).

8.1.4 Prototype Tools to Customize SPL Test Specifications

A set of prototype tools were developed in this research to automate the customization of the test specifications during feature-based test derivation in the CADeT approach. These tools applied the parameterization variability mechanism to select and customize these test specifications for an application derived from the SPL. Furthermore, an algorithm was developed to automate the generation of a test order graph for an application derived from the SPL.

8.1.5 An Evaluation of CAdET and CAdET-SoC on Two SPLs

This research evaluated all phases of CAdET and CAdET-SoC on two SPLs: An Automated Highway Toll System (AHTS) SPL, and a Banking System SPL. The results demonstrated that CAdET and CAdET-SoC can be used to create reusable and functional test specifications to cover all use case scenarios, features and relevant feature combinations of each SPL, and then configure these test specifications during feature-based test derivation to test a set of applications derived from each SPL.

8.2 Further Study

This approach has revealed several areas of further study, such as determining a break-even point, and automating more phases of the CAdET and CAdET-SoC test design methods, which are described next.

8.2.1 Determining a Break-Even Point

Extra effort (time in man-hours) was needed to learn and apply CAdET and CAdET-SoC to create feature-based reusable test specifications for a SPL. Although this reduced the number of test specifications and application configurations to test for each SPL, in general it is not clear when a break even point will occur. Further research needs to be done to determine when a break-even point will occur, where the benefits of automatically deriving test specifications for a set of applications from a SPL exceeds the initial effort spent applying CAdET or CAdET-SoC.

8.2.2 Automating More Phases of CADeT and CADeT-SoC

Some manual processes in CADeT and CADeT-SoC could be automated in order to scale these methods to larger SPLs. Tools developed to automatically generate decision tables from the activity diagrams of a single application (Vauthier 2006) could be extended to automatically generate decision tables from the activity diagrams of a SPL in phase II. Further, tools could be developed to partly automate the analysis of relevant feature combinations in phase III.

In phase VI, input data was selected by hand to satisfy the execution conditions in the test specifications of an application. Tools developed to automatically generate input data to satisfy the test predicates of a single application (Grieskamp, Tillmann et al. 2004) could be extended to automatically generate input test data to satisfy the execution conditions of the test specifications created with CADeT. The execution conditions of the test specifications would need to be formalized according to the language used by the tool.

In phase VII, test cases were manually executed against each application implementation. Tools developed to automate test execution (such as IBM Rational Functional Tester) could be extended to automate the execution of test cases created with CADeT or CADeT-SoC.

8.2.3 Incorporating Feature-Based and Use Case Scenario-Based Coverage

Criteria with Unit and Integration Testing Criteria

This research did not describe how feature-based and use case scenario-based coverage criteria could be incorporated with unit and integration testing criteria.

Integration testing includes testing of the variability in components and component connectors. Analyzing the impact of features on the implementation of a SPL can uncover feature interactions and implicit feature dependencies in the implementation.

8.2.4 Evaluating the Impact of SPL Evolution

This research did not evaluate the impact of SPL evolution on the test specifications created using CADeT and CADeT-SoC. Additional studies could be done to evaluate the effect of changing the feature and use case models of a SPL on the test specification suite of the SPL.

8.2.5 Resolving Inconsistencies between Requirement Models

Additional studies could be done to identify and resolve inconsistencies between the feature model and the functional requirements of a SPL. For instance, a feature interaction or implicit feature dependency in the functional requirements may indicate an undesirable inconsistency between the functional requirements and the feature model.

8.2.6 Incremental Testing of SPL

Instead of testing all features of a SPL, incremental testing could be applied to test a subset of all features, and then test additional features as the need arises. The test specification suite of the SPL could be extended and updated incrementally.

8.2.7 Integrating Additional Variability Mechanisms

Additional studies could be done to evaluate the feasibility and cost in man-hours of applying other variability mechanisms, such as XVCL Frames (Zhang and Jarzabek 2004), to customize the functional test specifications of a SPL.

8.2.8 Detecting Feature-Based Faults

Testing applications derived from an implementation of a Banking System SPL revealed some feature-based faults (see Table 49). However, this research did not compare the effectiveness of using CAdET against the effectiveness of using other functional testing approaches to discover feature-based faults in a SPL implementation. Additional studies could be done to compare the effectiveness of using CAdET against the effectiveness of using other functional testing methods to discover feature-based faults in a SPL implementation.

8.2.9 Evaluating CAdET and CAdET-SoC on Industrial SPLs

CAdET and CAdET-SoC were applied to create reusable test specifications from the requirement models of an AHTS SPL and a Banking System SPL. These SPLs were academic studies that had 12 to 16 features and 7 to 11 use cases. Additional studies could be done to evaluate CAdET and CAdET-SoC, on more realistic, industrial size problems (e.g. SPLs with several hundred features and use cases).

Appendix A: Banking System SPL case study

The Banking System Case Study Common Core (BSCS-CC) from (Webber 2001) described a use case model and variation points for a reusable version of the Banking System Case Study (BSCS) in (Gomaa 2000). Vonteru in (Vonteru 2001) implemented BSCS-CC and four target system implementations, which were used as a proof of concept for Webber's Variation Point Model (VPM) in (Webber 2001). In order to be able to apply CADeT and CADeT-SoC to BSCS-CC, these models were extended in this research to include a feature model, feature to use case relationship table, and use case descriptions in the format used by the PLUS method in (Gomaa 2005) .

This chapter describes the feature model, use case model, and feature to use case relationship table of the Banking System SPL, and then describes how CADeT and CADeT-SoC were applied to these models to create test specifications for the Banking System SPL.

A.1 Requirement Models for Banking System SPL

Figure 45 describes the feature model for the Banking System SPL, which contains the optional Call police, Phone branch, and Alarm actions, an exactly-one-of Language feature group with alternative English, Spanish and French features, an exactly-one-of Expired Card action feature group with alternative Confiscate and Eject

actions, and three parameterized features: Greeting, Pin format and Pin attempts. The Greeting feature refers to the welcome text prompt displayed at each ATM in the banking system. The Pin format feature refers to the pin length, which is set to a default of three numeric characters, but can vary depending on the application configuration. The Pin attempts feature refers to the maximum number of invalid pin attempts that can be entered by a customer in an ATM transaction.

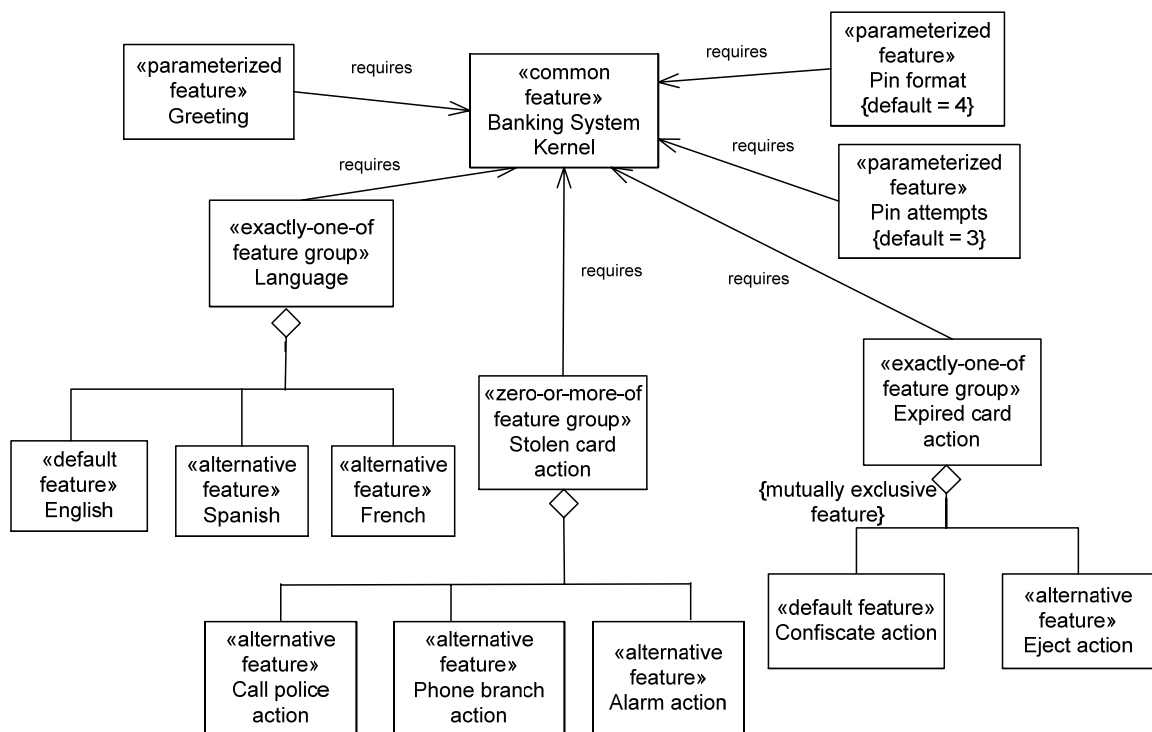


Figure 45 Feature model for Banking System SPL

Figure 46 describes the use case model for the Banking System SPL, which contains use cases initiated by Customer and Operator actors. The customer initiates the kernel use cases Query Account, Transfer Funds, and Withdraw Funds, all which include

the abstract Validate Pin use case. The operator initiates the kernel use cases Startup, Shutdown, and Add cash to ATM.

The Validate Pin use case contains several variation points, which are shown graphically in the use case model of Figure 46 and written up in the use case description of Figure 47: variation points vpGreeting, vpExpiredCardAction, vpPinFormat, vpMaxPinAttempts, vpStolenCardAction, and vpLanguage.

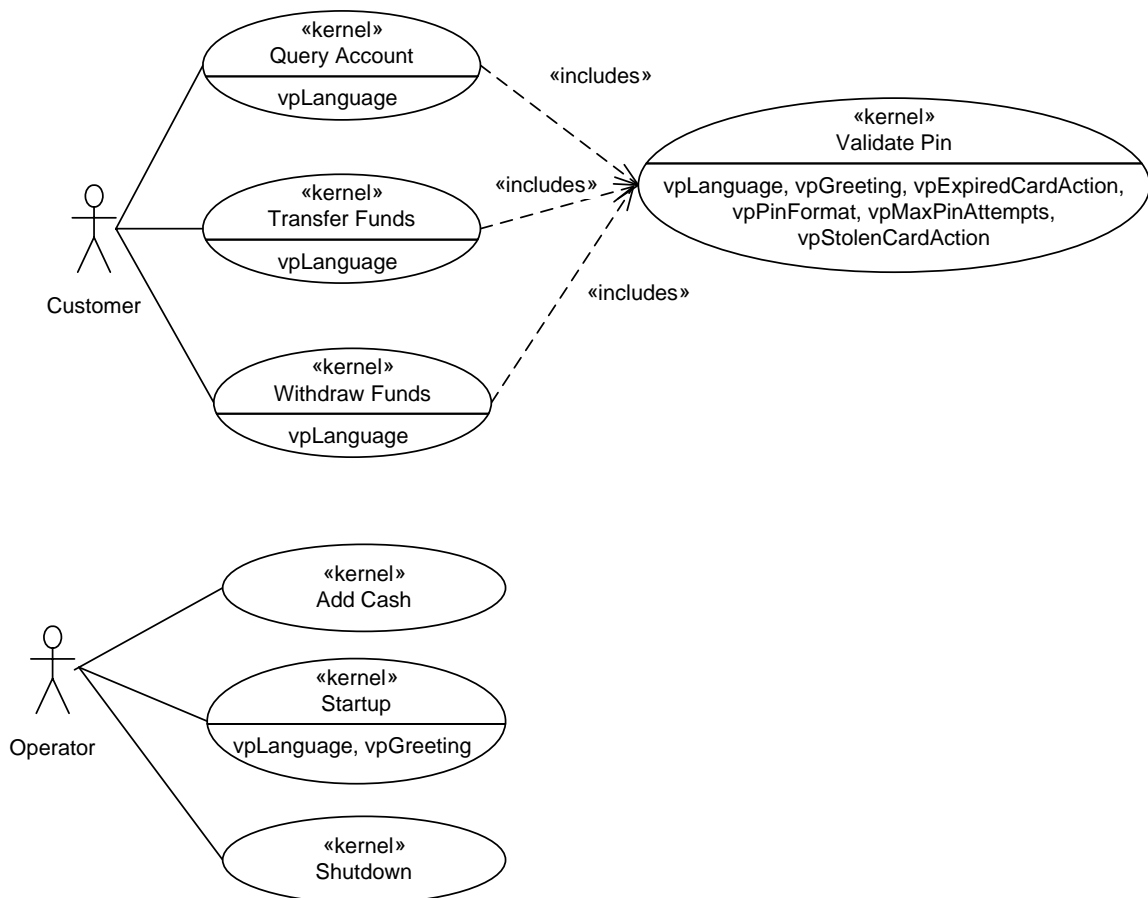


Figure 46 Use case model for Banking System SPL

Name: Validate PIN Abstract use case
Reuse category: kernel
Related to features: ATM kernel
Summary: System validates customer PIN
Actor: ATM Customer
Precondition: ATM is idle and displaying a welcome message
Description:

1. Customer inserts the ATM Card into the Card Reader
2. If the system recognizes the card, it reads the card number
3. System prompts customer for PIN number
4. Customer enters PIN
5. System checks the expiration date and whether the card is lost or stolen
6. If card is valid, the system then checks whether the user-entered PIN matches the card PIN in the system.
7. If PIN numbers match, the system checks what accounts are accessible with the ATM Card
8. System displays customer accounts and prompts customer for transaction type: Withdrawal, Query or Transfer

Alternatives

- Line 2: If the system does not recognize the card, the card is ejected
- Line 5: If the system determines that the card date has expired, the card is confiscated
- Line 5: If the system determines that the card has been reported lost or stolen, the card is confiscated
- Line 6: If the customer-entered PIN does not match the card PIN, the system re-prompts for the PIN
- Line 6: If the customer enters the incorrect PIN three times, the system confiscates the card
- Lines 3, 8: If the customer enters Cancel, the system cancels the transaction and ejects the card.

Postcondition (for main scenario): ATM is waiting for transaction
Variation points

Name: vpExpiredCard; **Type of functionality:** alternative
Lines: 5
Description: If the card is expired, the system can be configured to specify an alternate action besides confiscating the card. Confiscating the card is the default action

Name: vpPinFormat; **Type of functionality:** parameterized
Lines: 4
Description: The system can be configured to have an alternate PIN format. The default value is four.

Name: vpPinAttempts; **Type of functionality:** parameterized
Lines: 6
Description: The system can be configured to specify the maximum number of times a customer can enter a PIN. Three times is the default action.

Name: vpStolenCard; **Type of functionality:** optional
Lines: 5
Description: The system can be configured to add an optional action as a result of a stolen card. The default action is to confiscate the card.

Name: vpLanguage; **Type of functionality:** mandatory alternative
Lines: 3, 8
Description: Prompts and error messages will be in one of the specified languages: English, Spanish or French.

Figure 47 Validate pin use case description

A.2 Example of Phase I: Create Activity Diagrams during SPL Engineering

CADeT's Phase I (described in Chapter 5) was applied to create activity diagrams from each use case of the Banking System SPL. A use activity diagram was created from each use case description of the Banking System SPL. The use case steps and conditions in each use case description were mapped to activity nodes and decision nodes in the activity diagram, and additional control flows and activities were added to make the sequencing of activities more precise. An ATM system state variable was added to store the precondition and precondition states of each use case activity diagram. Then, the use case activity diagrams of the Banking System SPL were combined in a system level activity diagram (shown in Figure 48).

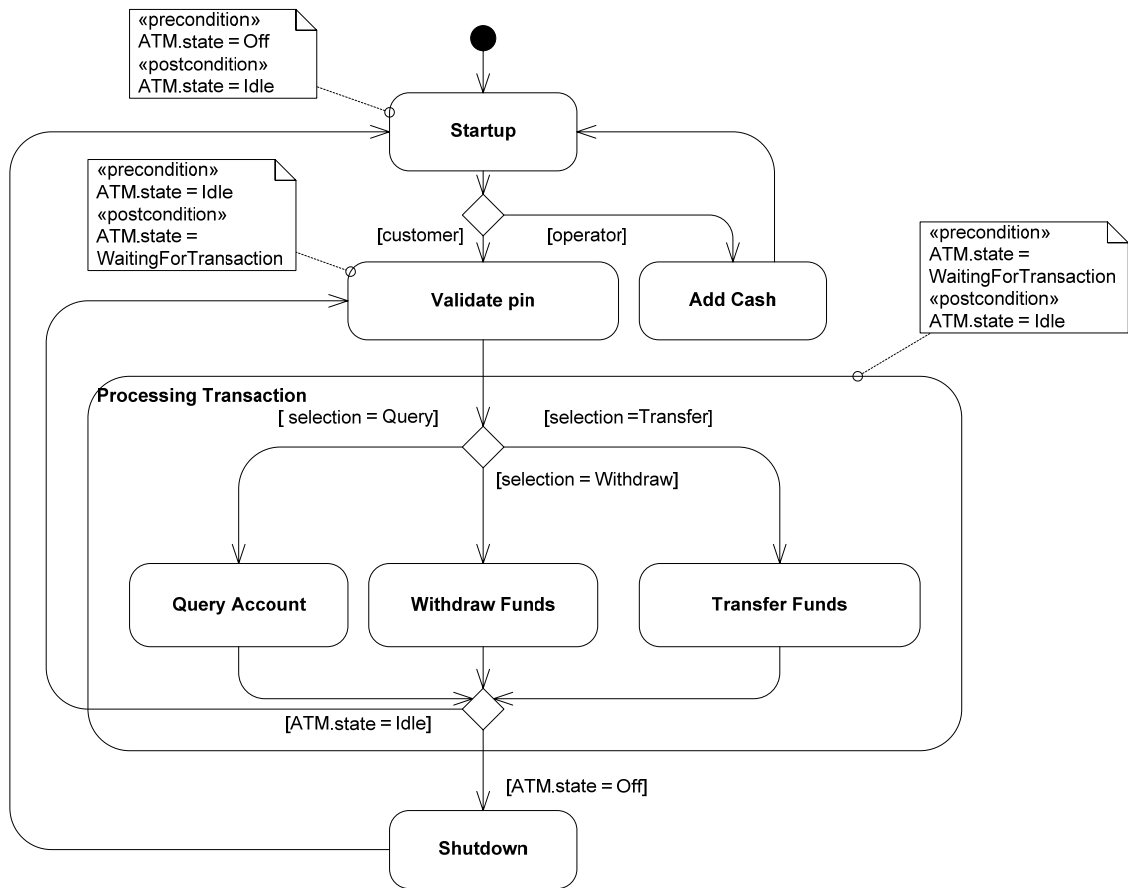


Figure 48 System level activity diagram for Banking System SPL

A feature list was created to describe the feature conditions and feature selections that correspond to features of the Banking System SPL in Table 54. The ATM Kernel feature condition represents the ATM Kernel feature and has a feature selection of T, which means that this feature is always selected for an application derived from the SPL. The alternative features english, french, spanish are represented by a language feature condition with $\{English, French, Spanish\}$ feature selection values. The alternative features confiscate and eject expired card are represented by an expiredCardAction

feature condition with $\{confiscate, eject\}$ feature selection values. The optional features call police action, phone branch action, and alarm action are represented by feature conditions with $\{T, F\}$ feature selections. The parameterized feature pinFormat refers to the pin length, which can range from 3 to 10 numeric digits; the parameterized feature pinAttempts refers to the maximum number of invalid pin attempts, which can range from one to five attempts; and the parameterized feature greetingPrompt refers to the welcome text prompt.

Table 54 Feature list for Banking System SPL

Feature condition	FeatureSelection
ATMkernel	T
language	$\{English, French, Spanish\}$
expiredCardAction	$\{confiscate, eject\}$
callPoliceAction	$\{T, F\}$
phoneBranchAction	$\{T, F\}$
alarmAction	$\{T, F\}$
pinFormat	$[3..10]$
pinAttempts	$[1..5]$
greetingPrompt	$[0..max]$ of character

Then, the impact of the SPL features on the activity diagrams was analyzed using the feature to use case relationship table together with the use case descriptions, as described in Chapter 5. Table 55 describes an excerpt from a feature to use case relationship table created for the Banking System SPL. The initial activity diagrams for the Banking System SPL were modified to describe feature conditions and reuse stereotypes, in order to explicitly associate activities in the activity diagrams to features in the feature model.

Table 55 Feature to use case relationship table for Banking System SPL

Feature Name	Feature Category	Use Case Name	Use Case Category / Variation Point (vp)	Variation Point Name
ATM kernel	common	Validate Pin Query Account Transfer Funds Withdraw Funds Add Cash Startup Shutdown	kernel kernel kernel kernel kernel kernel kernel	
english	default, alternative	Validate Pin Query Account Transfer Funds Withdraw Funds Startup	vp vp vp vp vp	vpLanguage vpLanguage vpLanguage vpLanguage vpLanguage
french	alternative	Validate Pin Query Account Transfer Funds Withdraw Funds Startup	vp vp vp vp vp	vpLanguage vpLanguage vpLanguage vpLanguage vpLanguage
spanish	alternative	Validate Pin Query Account Transfer Funds Withdraw Funds Startup	vp vp vp vp vp	vpLanguage vpLanguage vpLanguage vpLanguage vpLanguage
confiscate expired card action	default, alternative	Validate Pin	vp	vpExpiredCard
eject expired card action	alternative	Validate Pin	vp	vpExpiredCard
call police action	optional	Validate Pin	vp	vpStolenCard
Phone branch action	optional	Validate Pin	vp	vpStolenCard
alarm action	optional	Validate Pin	vp	vpStolenCard
confiscate stolen card action	common	Validate Pin	vp	vpStolenCard
greeting	parameterized	Startup, Validate Pin	vp	vpGreeting
pin attempts	parameterized	Validate Pin	vp	vpPinAttempts
pin format	parameterized	Validate Pin	vp	vpPinFormat

The feature to use case relationship table in Table 55 shows that the ATM kernel feature is associated with all use cases in the Banking System SPL, and that each optional and alternative feature in the feature model is associated with a variation point in one or more use cases in the Banking System SPL. For example, the vpLanguage variation point in the feature to use case relationship table is associated with the alternative English, French and Spanish language features. The vpLanguage variation point impacts all display prompt and error message activity nodes in the Validate Pin use case activity diagram in Figure 49. These activity nodes have been stereotyped as adaptable. Also, the vpLanguage variation point has been added to the parameter list of each activity node.

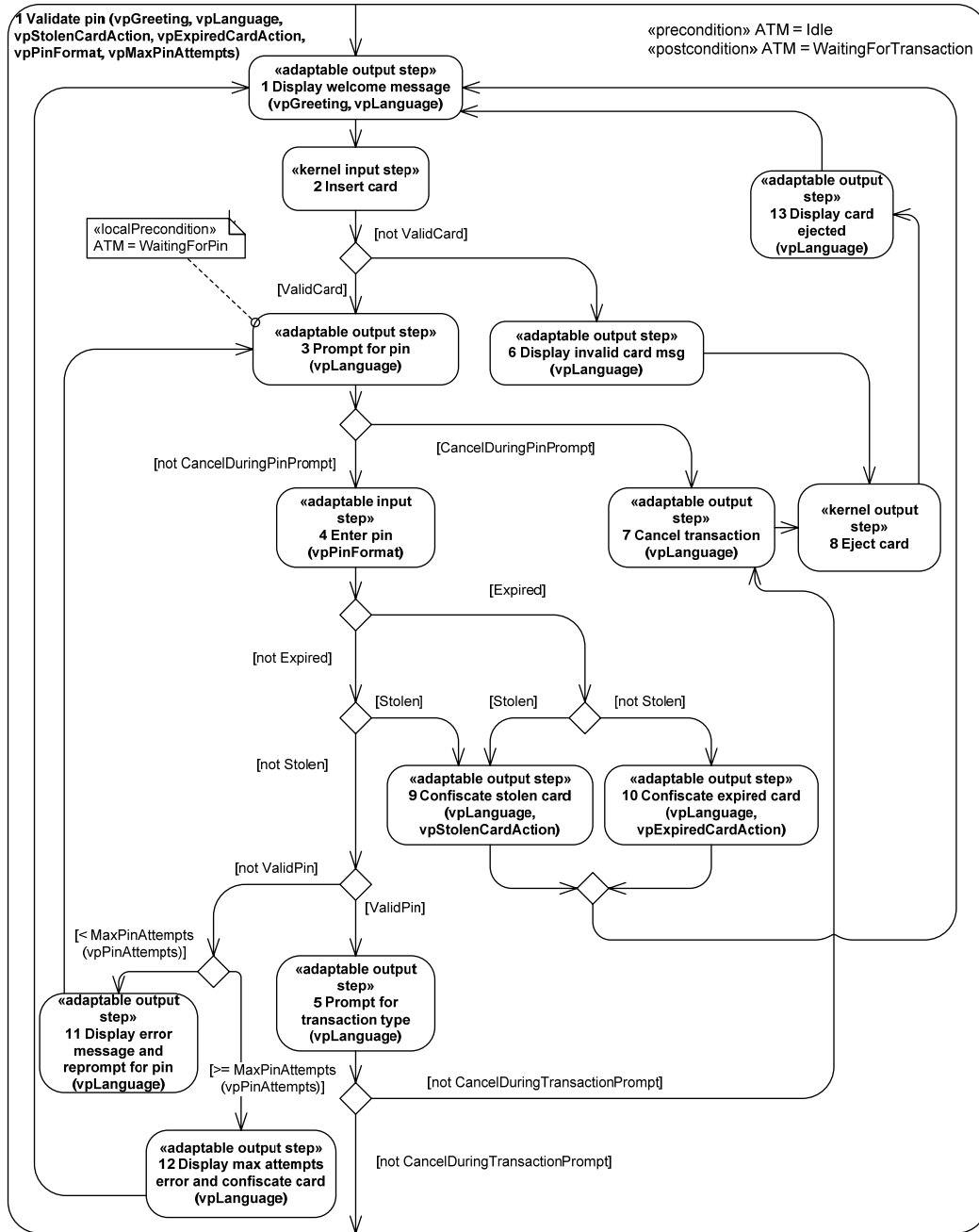


Figure 49 Activity diagram for “Validate pin” use case

Figure 50 shows sub-activity diagrams created for the “Confiscate stolen card”, “Confiscate expired card”, and generic “Display message” adaptable activity nodes. The feature conditions are underlined to distinguish them from the execution conditions in the activity diagrams.

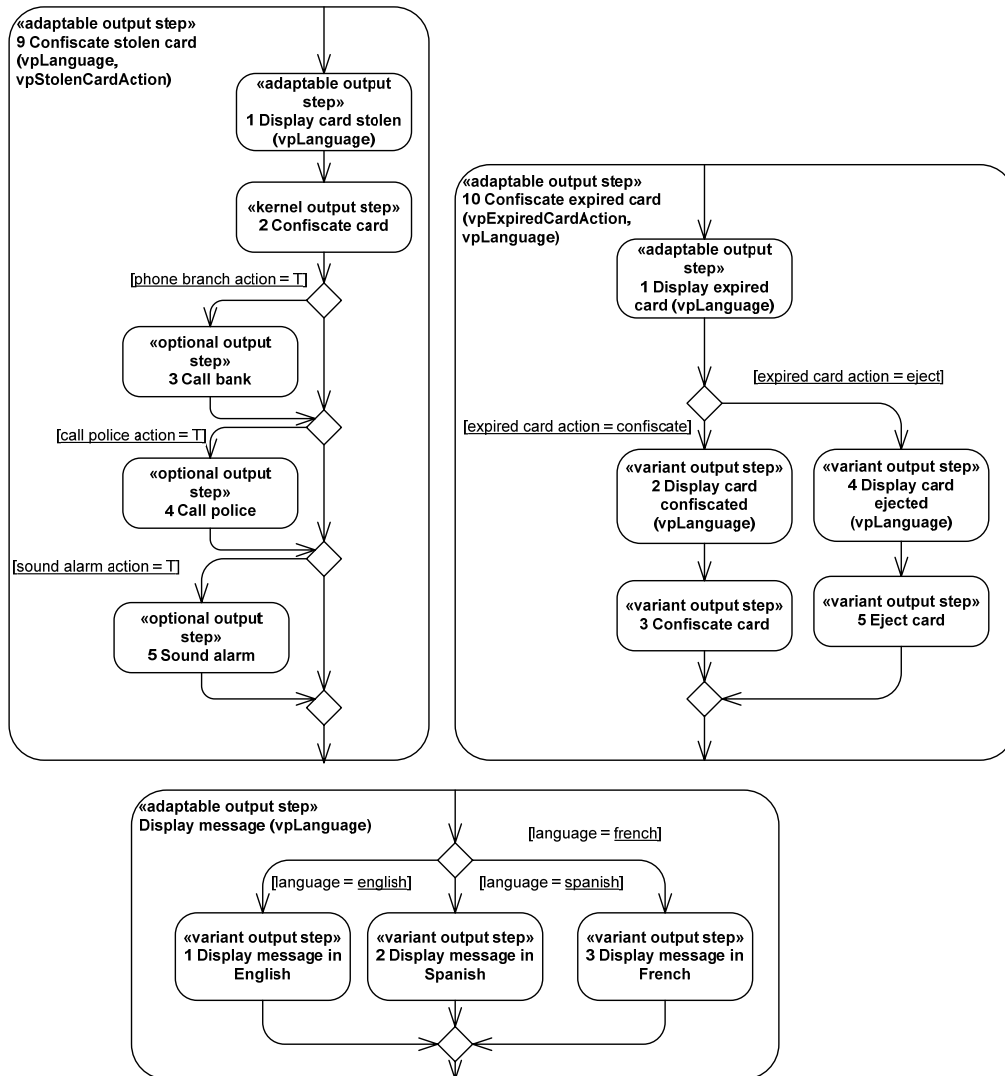


Figure 50 Sub-activity diagrams for adaptable nodes in “Validate pin” activity diagram

A.3 Example of Phase II: Create Decision Tables and Test Specifications from Activity Diagrams

CADeT's Phase II (described in Chapter 5) was applied to create a decision table from each use case of the Banking System SPL. This section describes how a decision table was created from the "Validate Pin" use case activity diagram in Figure 49.

Table 56 shows an excerpt from a decision table created from the "Validate Pin" use case activity diagram. The preconditions, feature conditions, executions conditions and postconditions from the activity diagram were added to rows in the decision table. The activity nodes from the "Validate Pin" use case activity diagram were mapped to test steps in the decision table. Further, test specifications were traced on the activity diagram for each use case scenario of the "Validate Pin" use case. The "Validate Pin" use case contains one main scenario and six alternative scenarios. Nine test specifications were traced for these seven scenarios. Some scenarios, such as the main scenario of "Validate Pin", covered a loop in the activity diagram, and needed to be covered by more than one test specification.

Table 56 Excerpt of decision table for “Validate pin” use case

1	Validate Pin	1	2	3	4	5	6	7
	Test specification	«adaptable» Card is valid	«adaptable» Pin is valid	«adaptable» Card not recognized	«adaptable» Card is expired	«adaptable» Card is stolen	«adaptable» Pin is invalid less than max times	«adaptable» Pin is invalid max times
Feature conditions	ATM Kernel	T	T	T	T	T	T	T
Preconditions	ATM	Idle	WaitingForPin	Idle	Idle	Idle	WaitingForPin	WaitingForPin
Execution conditions	Valid card	T	T	F	T	T	T	T
	CancelDuringPinPrompt		F		F	F	F	F
	Expired		F		T		F	F
	Stolen		F			T	F	F
	Valid pin		T				F	F
	>= Max attempts						F	T
	CancelDuringTrans Prompt		F				F	F
Actions								
1	«adaptable output step» System displays welcome message (vpGreeting, vpLanguage)	X		X	X	X		
2	«kernel input step» Insert card	X		X	X	X		
3	«adaptable output step» Prompt for pin (vpLanguage)	X			X	X		

4	«adaptable input step» Enter pin (vpPinFormat)		X		X	X	X	X
5	«adaptable output step» Prompt for transaction type (vpLanguage)		X					
6	«adaptable output step» Display invalid card msg (vpLanguage)			X				
7	«adaptable output step» Cancel transaction and display transaction canceled (vpLanguage)							
8	«kernel output step» Eject card							
9	«adaptable output step» Confiscate stolen card (vpLanguage, vpStolenCardAction)					X		
10	«adaptable output step» Confiscate expired card (vpLanguage, vpExpiredCardAction)				X			
11	«adaptable output step» Display error message and reprompt for pin (vpLanguage)						X	
12	«adaptable output step» Display max attempts error and confiscate card (vpLanguage)							X
13	«adaptable output step» Display card ejected (vpLanguage)							
Post conditions	ATM	WaitingForPin	WaitingFor Transaction	Idle	Idle	Idle	WaitingForPin	Idle

A.4 Example of Phase III: Define Feature-Based Test Plan

A feature-based test plan was created for the Banking System SPL. The Banking System SPL has a total of 12 features: one common feature, the ATM Kernel; three alternative features, English, Spanish, and French, which are part of an exactly-one-of Language feature group; three parameterized features, Greeting, Pin Format, and Pin Attempts; three optional features, Call Police action, Phone Branch action, and Alarm action; and two alternative features Confiscate action and Eject action, which are part of an exactly-one-of Expired card action feature group.

The Greeting, Pin Format, and Pin Attempts parameterized features describe values which must be defined during application derivation. The type and range of values was defined for each parameterized feature, and then the boundary-value test selection criterion was applied to select discrete values for each of these features. The type, range, and discrete values of each parameterized feature were defined as follows:

Greeting: The ATM user interface can display zero to four lines of text to greet the customers. The Greeting feature consists of a set of four text strings, where each text string has [0..50] characters. Two discrete values were selected for the Greeting parameterized feature: standard greeting and enhanced greeting. An application can be configured to have a standard greeting, which uses two lines of text to welcome the customer to a bank, or an enhanced greeting, which uses all four lines of text to welcome the customer and to advertise a bank service.

Pin Format: The Banking System can be configured to allow a customer to enter a pin of a specific length. The Pin Format feature is an integer, defined to be within the

range [3..10]. Three discrete values were selected for the Pin Format feature. An application can be configured to have a minimum pin length of three numeric characters, a maximum pin length of ten numeric characters, or a default pin length of four numeric characters.

Pin Attempts: The Banking System can be configured to allow a customer a maximum number of invalid pin attempts. The Pin Attempts feature is an integer, defined to be within the range [1..5]. Three discrete values were selected for the Pin Attempts feature. An application can be configured to allow a customer a minimum number of one invalid pin attempt, a maximum number of five invalid pin attempts, or a default number of three invalid pin attempts.

With these restrictions an application engineer can configure a total of:

$2^1 \times 3^2 \times 3^1 \times 2^3 \times 2^1$, or 864 possible application configurations, where 2^1 refers to the 2 values selected for the Greeting feature; 3^2 refers to the 3 values selected for the Pin Format and Pin Attempts features; 3^1 refers to the 3 feature selections of the alternative language features; 2^3 refers to the two feature selections (T or F) of each of the 3 optional features: Call Police action, Phone Branch action, and Alarm action; and 2^1 refers to the 2 feature selections of the alternative features Confiscate action and Eject action.

Then, the relationships of features to test specifications of the Banking System SPL were analyzed as described in Chapter 5. Table 57 shows an excerpt of a feature / test specification relationship table for the Banking System SPL. This table shows the feature combination functions associated with two test specifications from the “Validate

Pin” decision table of the Banking System SPL. The first test specification “1. «adaptable» Validate Pin: Card is valid” has two adaptable test steps. One of these adaptable test steps, “«adaptable output step» System displays welcome message (vpGreeting, vpLanguage)” contains two variation points, which are impacted by the greeting and language feature conditions, respectively. The sub-activity diagram associated with this adaptable node (in Figure 51) shows that the language features interact with the greeting features: a standard or enhanced greeting will be displayed in each language. Thus, in Table 57 the feature combination function *greeting*language* is associated with this adaptable test step. Next, the functions of each adaptable test step in the test specification are combined to create a feature combination function for the entire test specification. The functions *greeting*language* and *language* are combined to create the feature combination function *greeting*language+ language* for the “Card is valid” test specification. The former is associated with the “«adaptable output step» System displays welcome message”, and the latter is associated with the «adaptable output step» Prompt for pin” in Table 57. This feature combination function can be simplified by removing feature conditions that are already part of a feature interaction. For example, *language* was removed from the *greeting*language+ language* feature combination function of the “Card is valid” test specification to get *greeting*language*.

In contrast, the second test specification in Table 57 has two adaptable test steps that do not describe a feature interaction. The pinFormat and language features impact each adaptable test step, but these features do not interact, and are described by the feature combination function *pinFormat + language*.

Table 57 Excerpt of feature / test specification relationships in Banking System SPL

Use case: Test specification	Feature to test specification	Adaptable test steps	Feature to adaptable test step	Feature combination function
Validate Pin : 1. «adaptable» Card is valid	ATM Kernel = T	«adaptable output step» System displays welcome message (vpGreeting, vpLanguage)	greeting*language	greeting*language
		«adaptable output step» Prompt for pin (vpLanguage)	language	
Validate Pin: 2. «adaptable» Pin is valid	ATM Kernel = T	«adaptable input step» Enter pin (vpPinFormat)	pinFormat	pinFormat + language
		«adaptable output step» Prompt for transaction type (vpLanguage)	language	

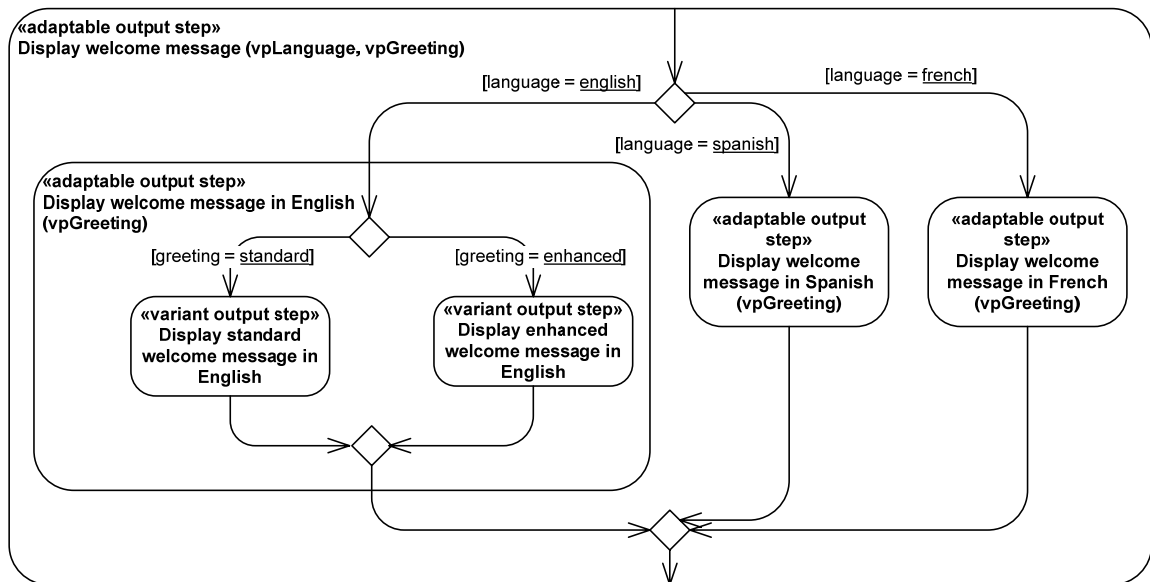


Figure 51 “Display welcome message” adaptable node

The largest number of feature conditions in a relevant feature combination function in Table 57 is 2. The remaining test specifications in the feature / test specification relationship table of the Banking System SPL were analyzed, and the largest number of feature conditions in a relevant feature combination in this table was also determined to be 2. Thus, at least a 2-way, or pair-wise combinatorial testing strategy was needed to check the feature combinations described by the test specifications of the Banking System SPL. Table 58 shows a feature-based combinatorial test plan that was generated to cover all valid pair-wise feature combinations in the Banking System SPL using the Jenny tool (Jenkins 2005). Thirteen application configurations were generated to cover all valid pair-wise feature combinations of features in the Banking System SPL.

Table 58 Pair-wise coverage criterion applied to Banking System SPL

TEST PLAN for Banking System													
Features:	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13
ATM Kernel													
a. TRUE	x	x	x	x	x	x	x	x	x	x	x	x	x
Language													
a. English	x				x		x			x			x
b. French		x		x				x			x	x	
c. Spanish			x			x			x				
Expired card action													
a. Confiscate action	x			x		x	x	x		x	x		x
b. Eject action		x	x		x				x			x	
Call police action													
a. TRUE	x			x		x		x	x	x			
b. FALSE		x	x		x		x				x	x	x
Phone branch action													
a. TRUE		x		x	x			x	x			x	x
b. FALSE	x		x			x	x			x	x		
Alarm action													
a. TRUE		x	x		x			x				x	x
b. FALSE	x			x		x	x		x	x	x		
Pin format [3..10]													
a. Default: 4		x				x	x					x	
b. Min: 3	x				x			x	x				x
c. Max: 10			x	x						x	x		
Pin attempts [1..5]													
a. Default: 3		x				x				x			x
b. Min: 1	x				x				x		x	x	
c. Max: 5			x	x			x	x					
Greeting													
a. Standard		x		x	x		x		x	x	x		
b. Enhanced	x		x			x		x				x	x

A.5 Example of Phase IV: Apply Parameterization Variability Mechanism

The parameterization variability mechanism was applied to the decision tables of the Banking System SPL.

The feature list in Table 54 was created to show all feature conditions and feature selections associated with the features of the Banking System SPL. This feature list was used to customize the decision tables during feature-based test derivation for each application of the Banking System SPL.

Table 59 shows an excerpt from the “Validate Pin” decision table, which has been modified to describe the feature conditions and variable (variant or optional) test steps associated with the variation points in the test specifications. The parameterization variability mechanism was applied to the “Validate Pin” decision table in Table 56 to create the modified decision table in Table 59.

The language, pinFormat, and greeting feature conditions (which are associated with vpLanguage, vpPinFormat, and vpGreeting variation points in the adaptable test steps in Table 56) were added to the “Validate Pin” decision table. Functions were added to associate the feature selections of a feature condition with test specifications in Table 59. For example, the language feature condition impacts the “«adaptable» Card is valid” and “«adaptable» Pin is valid” test specifications. A spreadsheet function was added to associate the {*English, French, Spanish*} feature selections of the language feature condition with each of these test specifications.

The adaptable steps in the “Validate Pin” decision table in Table 56 have been replaced with variant test steps in Table 59. These variant test steps are associated with variant activity nodes in the sub-activity diagram of the “«adaptable output step» Display message” node in Figure 50. Functions were added to enable or disable variant test steps depending on whether the corresponding feature is selected. Table 59 shows an

$\{X, Null\}$ entry in the intersection of the variant output test steps with the “Card Is Valid” and “Pin is Valid” test specifications. Selecting a language feature, such as English, during feature-based test derivation will enable the variant test steps associated with the English feature (replace $\{X, Null\}$ with X), and disable variant test steps associated with other features (replace $\{X, Null\}$ with $Null$)

Table 59 Excerpt from modified “Validate pin” decision table

1	Validate Pin	1	2
	Test specification	«adaptable» Card is valid	«adaptable» Pin is valid
Feature conditions	ATM Kernel	T	T
	language	{english, french, spanish}	{english, french, spanish}
	greeting	[0..max]	
	pinFormat		[3..10]
Preconditions	ATM	Idle	WaitingForP in
Execution conditions	Valid card	T	T
	CancelDuringPinPrompt		F
	Expired		F
	Stolen		F
	Valid pin		T
	>= Max attempts		
	CancelDuringTransPrompt		F
Actions			
1a	«variant output step» System displays welcome in English (vpGreeting)	{X, Null}	
1b	«variant output step» System displays welcome in Spanish (vpGreeting)	{X, Null}	
1c	«variant output step» System displays welcome in French (vpGreeting)	{X, Null}	
2	«kernel input step» Insert card	X	
3a	«variant output step» Prompt for pin in English	{X, Null}	
3b	«variant output step» Prompt for pin in Spanish	{X, Null}	
3c	«variant output step» Prompt for pin in French	{X, Null}	
4	«adaptable input step» Enter pin (vpPinFormat)		X
5a	«variant output step» Prompt for transaction type in English		{X, Null}
5b	«variant output step» Prompt for transaction type in Spanish		{X, Null}
5c	«variant output step» Prompt for transaction type in French		{X, Null}
Post conditions	ATM	WaitingForPin	WaitingFor Transaction

A.6 Example of Phase V: Customize Decision Tables and Test Specifications using Parameterization Variability Mechanism

The following example describes the customization process for TS1, one application from the test plan of the Banking System SPL in Table 58. Table 60 shows the feature list of the Banking System SPL customized for TS1. The features selected for application TS1 are the English language feature, Confiscate action for an expired card, Call police action for a stolen card, a pin format of length 3, a maximum of one invalid pin attempt, and an enhanced greeting prompt.

Table 60 Feature selections for application TS1 from the Banking System SPL

Feature condition	Feature selection
ATM kernel	T
language	English
expired card action	Confiscate
call police action	T
phone branch action	F
alarm action	F
pinFormat	3
pinAttempts	1
greetingPrompt	Enhanced

Setting the feature conditions in the feature list customizes the decision tables for TS1. Table 61 is an excerpt of a customized decision table for the “Validate Pin” use case of the Banking System SPL. The English language, Enhanced greeting prompt, and pinFormat = 3 features have been selected for TS1, and the values of the feature conditions associated with these features in the “Validate Pin” decision table have been set accordingly. In the “Card is Valid” test specification, the test steps “1a «variant output

step» System displays welcome in English” and “3a «variant output step» Prompt for pin” in English have been enabled because these steps are associated with the English feature, but the test steps associated with the French and Spanish features are disabled in all test specifications of the Banking System SPL.

In the “Pin is Valid” test specification the test step “«variant output step» Prompt for transaction type in English” is enabled because it is associated with the English feature. The parameterized feature pinFormat constrains the possible values of the pin input variable in test step “4 «adaptable input step» Enter pin (vpPinFormat)”. The value of this input variable is selected in Phase VI of CADeT.

Next, the test specification generator tool is used to generate the test specifications document from the customized decision tables. Table 62 is an example of a test specification generated for the “Card is Valid” test specification of the Validate Pin decision table in Table 61.

Table 61 Excerpt of customized decision table for “Validate pin” use case

1	Validate Pin		
	Test specification	«adaptable» Card is valid	«adaptable» Pin is valid
Feature conditions	ATM Kernel	T	T
	Language	English	English
	Greeting	Enhanced	
	pinFormat		3
Preconditions	ATM	Idle	WaitingForPin
Execution conditions	Valid card	T	T
	CancelDuringPinPrompt		F
	Expired		F
	Stolen		F
	Valid pin		T
	>= Max attempts		
	CancelDuringTrans Prompt		F
Actions			
1a	«variant output step» System displays welcome in English	X	
1b	«variant output step» System displays welcome in Spanish		
1c	«variant output step» System displays welcome in French		
2	«kernel input step» Insert card	X	
3a	«variant output step» Prompt for pin in English	X	
3b	«variant output step» Prompt for pin in Spanish		
3c	«variant output step» Prompt for pin in French		
4	«adaptable input step» Enter pin (vpPinFormat)		X
5a	«variant output step» Prompt for transaction type in English		X
5b	«variant output step» Prompt for transaction type in Spanish		
5c	«variant output step» Prompt for transaction type in French		
Post conditions	ATM	WaitingForPin	WaitingForTransaction

Table 62 Card is valid test specification

Use case name	Validate Pin	
Test specification	«adaptable» Card is valid	
Feature conditions		
	ATM kernel	T
	language	English
	greeting	Enhanced
Preconditions		
	ATM	Idle
Execution conditions		
	Valid card	T
Actions		
	«variant output step» System displays welcome message in English	
	«kernel input step» Insert card	
	«variant output step» Prompt for pin in English	
Post conditions		
	ATM	WaitingForPin

Next, the test procedure definition tool is used to create a test procedure document for application TS1. The test procedure definition tool in CADeT generates a graph from the test specifications selected for TS1 and allows a test engineer to trace paths through this graph in order to create system tests. The tool sorts the test specifications according to the preconditions and postconditions described in the system level activity diagram of the Banking System SPL in Figure 48. Each path is saved as a system test in the test procedure document for TS1. The test procedure definition tool also keeps track of the percentage of test specifications covered by the system tests in the test procedure document.

Figure 52 is an excerpt of the graph generated for TS1, which shows the execution dependencies between some of the test specifications selected for TS1. Table 63 shows two system tests traced from this graph for the test procedure of TS1. The first system test (highlighted in Figure 52) describes the situation where the ATM customer enters a valid card and valid pin, and then queries his or her account. The second system test describes the situation where the ATM customer enters an invalid pin. Since the maximum number of pinAttempts in TS1 is 1, entering an invalid pin once causes the application to display a max invalid pin attempts error and to eject the card.

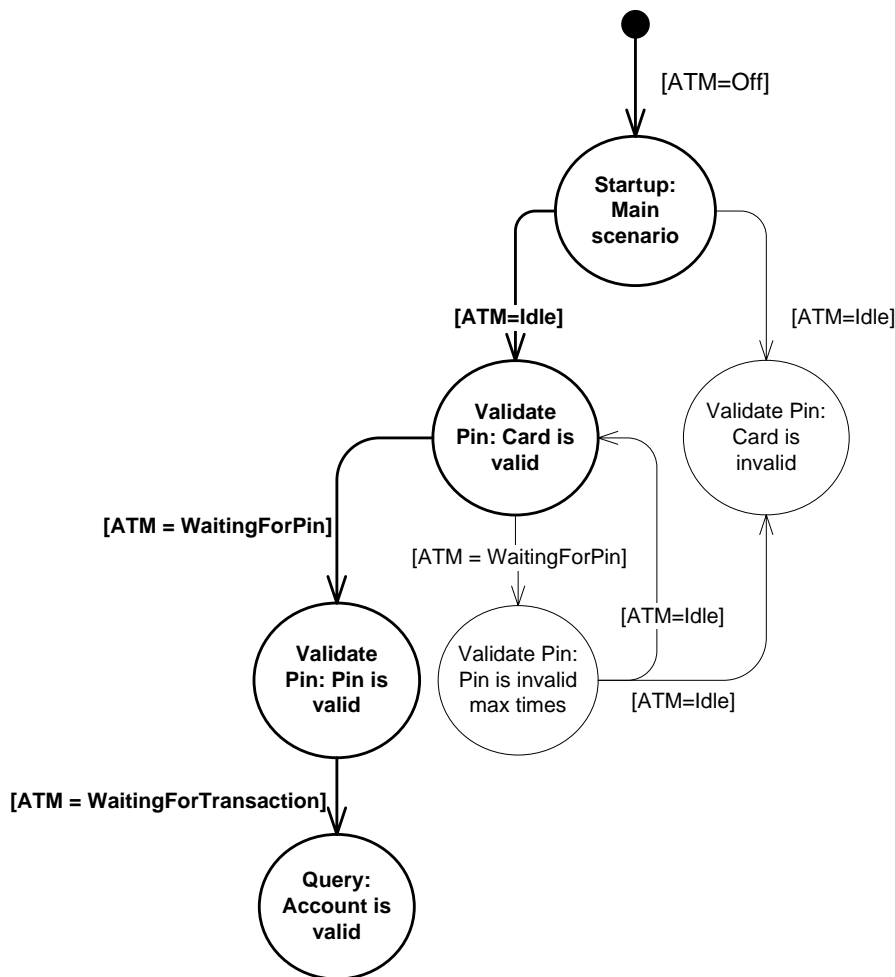


Figure 52 Dependencies between test specifications of TS1

Table 63 Example of system test sequences for TS1

System test 1
Startup: «adaptable» Main scenario
Validate Pin: «adaptable» Card is valid
Validate Pin: «adaptable» Pin is valid
Query: «adaptable» Account is valid
System test 2
Startup: «adaptable» Main scenario
Validate Pin: «adaptable» Card is valid
Validate Pin: «adaptable» Pin is invalid max times

The test specifications of the Banking System SPL were refined to describe the actual input and environment variables used by the Banking System SPL implementation, and a system tests document was generated for application TS1. Table 64 shows an excerpt of “System test 1”, which is referenced in the test procedure in Table 63. In Table 64 the execution condition “Valid card” is described in terms of the input variable cardId and the attributes from the DebitCard class, which are described in Figure 53. The static model for the Banking System SPL is described in Figure 54.

Table 64 Excerpt from “System test 1” of TS1

Use case name	Validate Pin		Inputs & Outputs	Pass / Fail
Test specification	«adaptable» Card is valid			
Feature conditions				
	ATM kernel	T		
	language	English		
	greeting	Enhanced		
Preconditions				
	ATM	Idle		
Execution				

conditions				
	Valid card: (cardId? = *DEBIT_CARD.cardId AND DEBIT_CARD.status = 0)	T		
Actions				
	«variant output step» System displays welcome message in English (out greeting)		greeting!	
	«kernel input step» Insert card (in cardId)		cardId?	
	«variant output step» Prompt for pin in English			
Post conditions				
	ATM	WaitingFor Pin		
Use case name	Validate Pin			
Test specification	«adaptable» Pin is valid			
Feature conditions				
	ATM kernel	T		
	Language	English		
Preconditions				
	ATM	WaitingFor Pin		
Execution conditions				
	Valid card	T		
	CancelDuringPinPrompt	F		
	Expired: DEBIT_CARD.exp_date < today	F		
	Stolen	F		
	Valid pin: DEBIT_CARD.pin = pin?	T		
	CancelDuringTransPrompt	F		
Actions				
	«variant input step» Enter pin (in pin)		pin?	
	«variant output step» Prompt for transaction type in English			
Post conditions				
	ATM	WaitingFor Transaction		

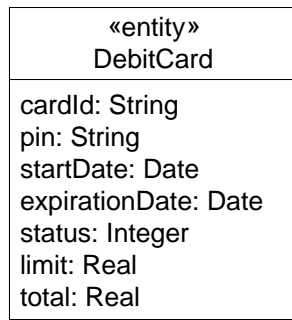


Figure 53 Debit card class

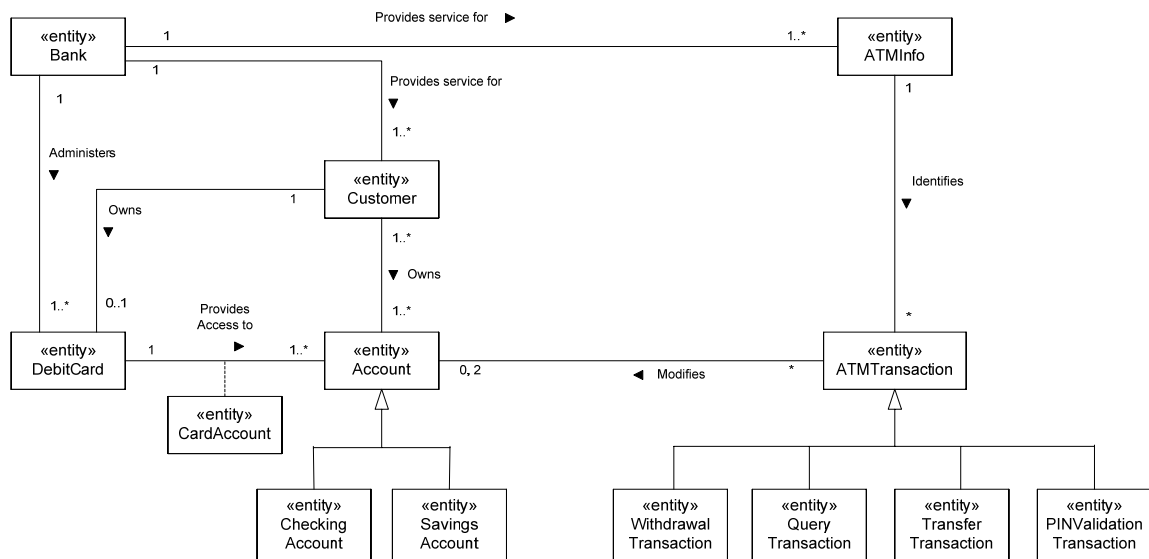


Figure 54 Static model for Banking System SPL

A.7 Example of Phase VI: Select Test Data

Next, input data was selected for the database and system tests of application TS1. An excerpt of the database for the application TS1 is shown in Table 65. The tables in the database correspond to classes with the same name in the static model in Figure 54. In Table 65, the association constraints between the classes in the static model have been

mapped to test data selection rules. Table 65 shows input data selected to satisfy the test data selection rules in the database of TS1.

Table 65 Example of input data selected for database of TS1

1. There exists at least one bank

BANK	
bank_name	address
Bank One	ABCD

2. There exists at least one ATM in ATM INFO, and a bank is associated to at least one ATM

ATM_INFO			
id	location	address	bank_name
1	XXXX	YYYY	Bank One

3. There exists at least one customer in CUSTOMER

CUSTOMER		
id	name	address
5678	Nelly	ZZZZ

4a. There exists at least one checking account

CHECKING_ACCOUNT		
account_number	balance	last_deposit
12345	50	0

4b. There exists at least one savings account

SAVINGS_ACCOUNT		
account_number	balance	interest
23456	1000	4

5. A customer has one or more accounts

CUSTOMER_ACCOUNT		
cust_id	account_number	
5678	12345	
5678	23456	

6. A customer has zero or more debit cards. There exists at least one valid card, one expired card, and one stolen card.

The debit card pin must conform to the pin format selected for this target system

DEBIT_CARD							
card_id	pin	start_date	exp_date	status	limit	total	cust_id
10	123	1/1/2004	1/1/2010	0	150	0	5678
11	123	1/1/2004	1/1/2006	0	150	0	5678
12	123	1/1/2004	1/1/2010	220	150	0	5678

7. A debit card is associated to one or more accounts

CARD_ACCOUNT		
card_id	account_number	
10	12345	
10	23456	
11	12345	
12	12345	

Test data was selected to satisfy the execution conditions in the system tests document of TS1. Table 64 describes the execution condition “Valid card: (cardId? = *DEBIT_CARD.cardId AND DEBIT_CARD.status = 0) = T” in the “«adaptable» Card is valid” test case in System test 1 of TS1. The “card id?” input variable refers to a card id entered by an ATM customer. This input must match a selection of any “card_id” with a 0 (valid) status from the “DEBIT_CARD” database table of the application. For example, “card id?=10” will satisfy this condition since the database table of TS1 contains a valid card id of “10” in Table 65. In a similar manner, test data was selected to satisfy the execution condition “Valid pin: DEBIT_CARD.pin = pin?” in Table 64. The “pin?” input variable refers to a pin entered by an ATM customer. This pin must match the pin of the previously selected card_id from “DEBIT_CARD” database table of the application. For example, “pin?=123” will satisfy this condition since card_id “10” in the DEBIT_CARD table has “pin=123” in Table 65.

A.8 Example of Phase VII: Test Application

An application was derived from the BSCS-CC implementation of the Banking System SPL by Vonteru in (Vonteru 2001) for each application described in the test plan for the Banking system SPL in Table 58. Then, the test cases were executed against each

application implementation of the test plan. This section explains how test cases were executed against the implementation of application TS1.

The database of TS1 was initialized to contain the values in Table 65. Then, client and server programs of application TS1 were installed on a Personal Computer (PC) with Windows O/S and the Java Run Time library (JRE). Then, the server and client program were initialized. Figure 55 shows the ATM user interface for the client program of TS1.

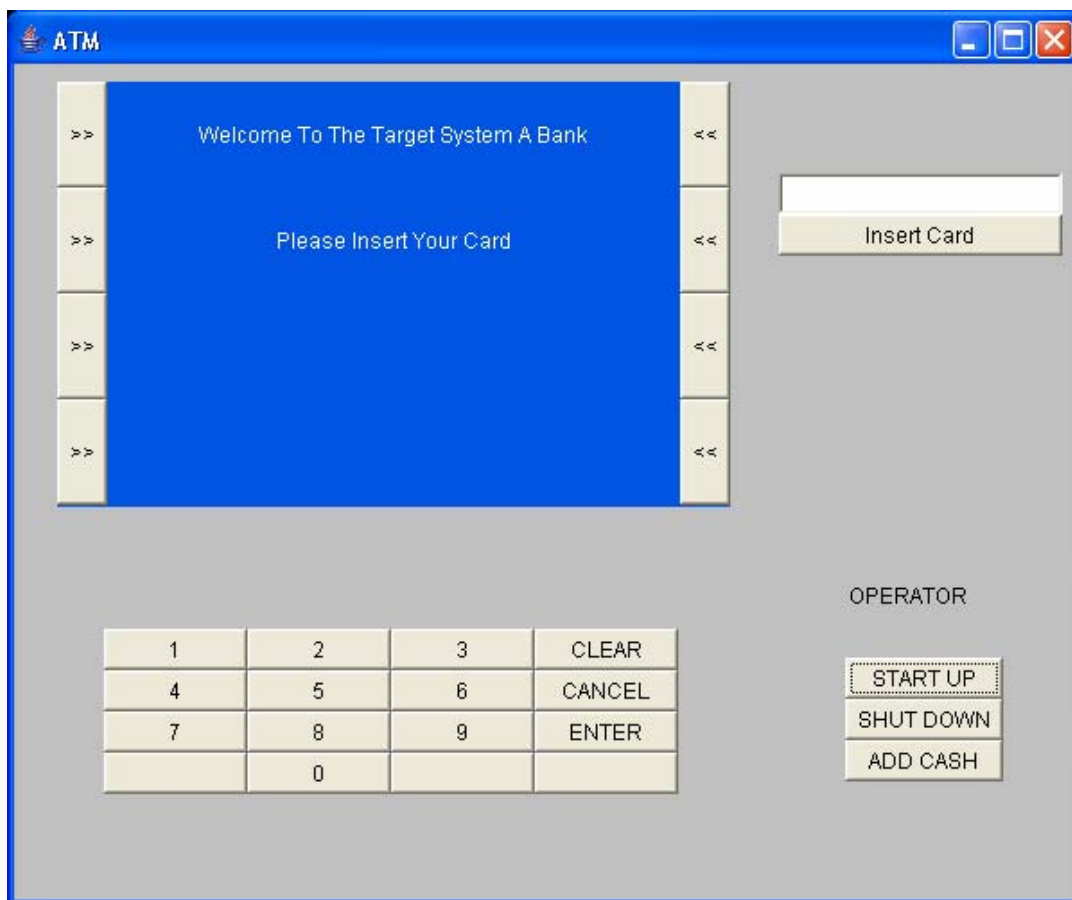


Figure 55 ATM user interface for TS1

Next, test cases from the system tests document of TS1 were executed against the client program by manually entering the inputs, observing the outputs, and comparing the actual outputs with the expected outputs of each test step. The test engineer assigned a Pass result to a test case if the actual outputs matched the expected outputs; else, the test engineer assigned a Fail result along with an explanation of the failure.

Table 66 shows how test results were entered for the “Card is valid” test case in System test 1. The test engineer simulated a card insertion in the ATM client program in Figure 55 by entering 10 for the card id, and then clicking on the “Insert Card” button in the user interface. The ATM program validated the card id and then displayed a pin prompt in English. This observed sequence of events matched the expected sequence of test steps in the “Card is valid” test case. Thus, Pass results were assigned by the test engineer to all test steps, and a Pass result was inferred for the entire test case.

Table 66 Test results for “Card Is Valid” test case in system test 1

Use case name	Validate Pin		Inputs & Outputs	Pass / Fail
Test specification	«adaptable» Card is valid			
Feature conditions				
	ATM kernel	T		
	Language	English		
	Greeting	Enhanced		
Preconditions				
	ATM	Idle		
Execution conditions				
	Valid card: (cardId? = DEBIT_CARD.card_id AND DEBIT_CARD.status = 0)	T		
Actions				
	«variant output step» System displays welcome message in English			Pass

	«kernel input step» Insert card (in cardId)		cardId? 10	Pass
	«variant output step» Prompt for pin in English			Pass
Post conditions				
	ATM	WaitingForPin		

A.9 Example of Applying Separation of Concerns Variability Mechanism in CAdET-SoC

The separation of concerns variability mechanism was applied to the decision tables created from the use case activity diagrams of the Banking System SPL in Phase II. The following example describes how Phase IV-SoC in chapter 6 was applied to define test insertion points in the decision table of the “Validate Pin” use case in Table 67, and to create the variable test step file in Figure 56.

Table 67 shows unique test insertion points added to each adaptable test step in the “Validate Pin” use case decision table. The test insertion points \$START insA, \$START insB and \$START insC precede the adaptable output steps “System displays welcome message (vpGreeting, vpLanguage)”, “Prompt for pin (vpLanguage)”, and “Prompt for transaction type (vpLanguage).” This decision table was exported into a text file format, to be compatible with the SPLET tool (Saleh and Gomaa 2005).

Figure 56 shows an excerpt from the variable test step file created for the Banking System SPL. The variable test step file describes the variable test steps and test insertion points associated with each feature. The “«adaptable output step» System displays welcome message (vpGreeting, vpLanguage)” is associated with the parameterized Greeting and alternative Spanish, French, and English language features. In Figure 56,

each language feature is associated with variable test steps that impact insertion points
insA, insB, and insC.

Table 67 Insertion points in “Validate pin” decision table

1	Validate Pin		
	Test specification	«adaptable» Card is valid	«adaptable» Pin is valid
Feature conditions	ATM Kernel	T	T
Preconditions	ATM	Idle	WaitingForPin
Execution conditions	Valid card	T	T
	CancelDuringPinPrompt		F
	Expired		F
	Stolen		F
	Valid pin		T
	>= Max attempts		
	CancelDuringTrans Prompt		F
Actions			
1	\$START insA «adaptable output step» System displays welcome message (vpGreeting, vpLanguage)	X	
2	«kernel input step» Insert card	X	
3	\$START insB «adaptable output step» Prompt for pin (vpLanguage)	X	
4	«adaptable input step» Enter pin (vpPinFormat)		X
5	\$START insC «adaptable output step» Prompt for transaction type (vpLanguage)		X
Post conditions	ATM	WaitingForPin	WaitingForTransaction

```

////////////////////////////////////
$FEATURE[English]

$START insA
<variant output step> System displays welcome message in English.
$END insA

$START insB
<variant output step> Prompt for pin in English.
$END insB

$START insC
<variant output step> Prompt for transaction type in English.
$END insC

$ENDFEATURE[English]

////////////////////////////////////
$FEATURE[French]

$START insA
<variant output step> System displays welcome message in French.
$END insA

$START insB
<variant output step> Prompt for pin in French.
$END insB

$START insC
<variant output step> Prompt for transaction type in French.
$END insC

$ENDFEATURE[French]

////////////////////////////////////
$FEATURE[Spanish]

$START insA
<variant output step> System displays welcome message in Spanish.
$END insA

$START insB
<variant output step> Prompt for pin in Spanish.
$END insB

$START insC
<variant output step> Prompt for transaction type in Spanish.
$END insC

```

Figure 56 Excerpt of variable test step file for Banking System SPL

The following example describes how test specifications were customized for TS1 (from the test plan of the Banking System SPL in Table 58), using the Static Customization of Test specifications (SCT) approach described in Phase V-SoC in chapter 6.

The English language, Enhanced greeting features, Confiscate expired card action, call police action, pinFormat = 3 and pinAttempts = 1 features were selected for TS1. A test specifications document was generated from the modified decision tables, and the test specifications in this document were exported to text files. Then, the code weaver component of the SPLET tool (Saleh and Gomaa 2005) was applied to customize the adaptable test steps in the test specifications selected for TS1. Figure 58 shows how the “Card is Valid” test specification Figure 57 was customized for TS1. The insertion points \$START insA and \$START insB have been replaced with the variant output test steps “System displays welcome message in English” and “Prompt for pin in English”, since both of these test steps are associated with the English language feature.

Use case name: Validate Pin
Test specification name: «adaptable» Card is valid
Feature conditions:
 ATMKernel = T
Preconditions:
 ATM = Idle
Execution conditions:
 ValidCard = T
Actions:
 \$START insA «adaptable output step» System displays welcome message
 (vpGreeting, vpLanguage)
 «kernel input step» Insert card
 \$START insB «adaptable output step» Prompt for pin (vpLanguage)
Post conditions:
 ATM WaitingForPin

Figure 57 “Card is valid” test specification

Use case name: Validate Pin
Test specification name: «adaptable» Card is valid
Feature conditions:
 ATMKernel = T
Preconditions:
 ATM = Idle
Execution conditions:
 ValidCard = T
Actions:
 //\$START insA «adaptable output step» System displays welcome message
 (vpGreeting, vpLanguage)
 «variant output step» System displays welcome message in English.
 «kernel input step» Insert card
 //\$START insB «adaptable output step» Prompt for pin (vpLanguage)
 «variant output step» Prompt for pin in English.
Post conditions:
 ATM WaitingForPin

Figure 58 “Card is valid” test specification customized for TS1

Appendix B: Glossary

Activity diagram: A UML diagram that shows the decomposition of an activity into its parts, which may contain other activities. An activity is a behavioral specification that describes the sequential and concurrent steps of a computational procedure (Rumbaugh, Jacobson et al. 2005).

Adaptable test specification: A test specification created with the CADeT method that is customized and then selected for an application of the SPL during feature-based test derivation.

Aggregate step: A role stereotype assigned to an activity node (in an activity diagram created with the CADeT method) that describes a group of related activities corresponding to a sequence of events in a use case description.

Aspect: In aspect-oriented programming, a modular, cohesive unit of functionality that physically separates and encapsulates a cross-cutting feature from the rest of the code.

Binding time: The time at which the variability is set, or fixed, for a test specification, for example, at design time, during feature-based test derivation, or at run-time.

Case study research method: An empirical investigation of the effect of a contemporary phenomenon (method, tool, etc...) within its real life context, when the

boundaries between the phenomenon and context are not clearly distinguishable, and in which multiple sources of evidence are used (Yin 2003).

Code tangling: In separation of concerns, a concern that is interwoven with the functional code of modules in an application, and with the code of other concerns in the application. See *separation of concerns*, *cross-cutting concern* and *scattering*.

Configurability: A quality criterion used to evaluate the ease with which the test specifications can be configured for the applications of a SPL

Controlled inputs: A test input sent by actor into an application, that satisfies the precondition and execution conditions of a test specification.

Cross-cutting concern: In separation of concerns, an aspect, or system characteristic, that cuts across other aspects and across executable code. These system characteristics cannot be decoupled from other functions in the program, and cannot be encapsulated in a generalized procedure. Some examples of cross-cutting concerns are non-functional features that affect the semantics or performance of a program.

Cross-cutting feature: A feature that cuts across other features and across executable code.

Customization method: A method of applying a variability mechanism to the reusable assets of an SPL, so that these assets can be customized for each application derived from the SPL.

Decision table: A table used in CADeT to describe and group the test specifications associated with a use case's scenarios. Each column describes one test specification and each row describes the conditions or steps of the test specifications. A

check in the intersection of a condition or step row with a test specification column indicates that the condition or step is relevant for the test specification.

Dynamic Customization of Client applications (DCAC): A method of customizing the code for an application of a SPL at system initialization time during application engineering. This method reads a customization file that contains the application's selected features and values of parameterized variables (Saleh 2005).

Execution condition: A test predicate in a test specification that constrains the values of controlled input variables and environment variables. This predicate must be satisfied during testing in order to execute the test steps in the expected sequence.

Expected output: A test output that is expected to be sent from an application to an actor during testing.

Explanatory case study: A case study research method where a hypothesis is formulated and then tested to evaluate the cause and effect relationships of a contemporary phenomenon (method, tool, etc...) on one or more cases. Analytic, rather than statistical generalization is used to relate the results to hypothesis (Yin 2003).

Exploratory case study: A case study research method where the effects of a contemporary phenomenon (method, tool, etc...) are explored on one or more cases to develop or refine a hypothesis (Yin 2003).

Feature: A requirement or characteristic that is provided by one or more applications of a software product line.

Feature-based application derivation: A method of deriving an application implementation based on the features selected for an application of a SPL.

Feature-based test derivation: A method of deriving test specifications based on the features selected for an application of a SPL.

Feature condition: A variable that associates a model element to features in a feature model, in which the variable values represent possible feature selections. In an activity diagram, a feature condition associates the control flow and activities in an activity diagram to features in a feature model.

Feature dependency: A configuration constraint where the selection of one feature requires or excludes the selection of another feature.

Feature interaction: A functional behavior that is enabled for a feature combination selected for an application derived from the SPL, but that is not enabled when any feature of the combination is selected separately.

Feature to use case relationship table: A table used with the PLUS method to associates a feature in the feature model with one or more use cases or use case variation points (Gomaa 2005).

Insertion point: A notation used by the SCAC and DCAC methods to uniquely identify and name a location of variation in the code of a SPL (Saleh 2005). See *variable feature file*, *Static Customization of Client Applications (SCAC)* and *Dynamic Customization of Client applications (DCAC)*.

Modifiability: A quality criterion used to evaluate the ease with which the test specifications can change when iterative changes are made to the feature model of the SPL.

Pair-wise testing: A combinatorial testing criterion where each feature must be tested with another feature at least once, in order to reveal faults caused by combinations of at most two features.

Parameterization variability mechanism: A technique that uses feature conditions to enable the automatic configuration of the variability in an application's test specifications during feature-based test derivation. Feature conditions are associated with the features of a SPL, and the values of a feature condition represent possible feature selections.

Precondition: A predicate which describes the value of a state-dependent variable, which must be satisfied before an activity or test specification can be executed.

Postcondition: A predicate which describes the value of a state-dependent variable, which must be satisfied after an activity or test specification is executed.

Reuse stereotype: A UML notation used in the PLUS method to classify a modeling element in a SPL by its reuse properties (Gomaa 2005).

Role stereotype: A UML notation used in the PLUS method to classify a modeling element by the role it plays in the application (Gomaa 2005).

Scattering: A separation of concerns quality criterion used to describe the dispersion of the code implementing a concern over the system modules. See *separation of concerns*, *cross-cutting concern* and *code tangling*.

Static Customization of Test Specifications (SCT): A technique that extends SCAC to separate variable test steps from the test specifications of a SPL, and then

associate these test steps with an alternative or optional feature from the SPL. See *Static Customization of Client applications (SCAC)*.

Separation of concerns: A quality criterion used to evaluate the ease with which the variability and commonality in the implementation can be decoupled and associated with a concern. See *cross-cutting concern*.

Separation of concerns variability mechanism: A variability mechanism used in CAdET-SoC to achieve feature-based separation of concerns in the test specifications of a SPL.

Simple path: A sequence of unique activities traced from an activity diagram, which starts at a precondition and ends at a postcondition in the activity diagram, and does not contain repeated activity nodes.

Software Product Line (SPL): A collection of applications that have so many features in common that it is worthwhile to study and analyze the common features as well as analyzing the features that differentiate these applications, in order to efficiently develop next generation applications.

Static Customization of Client applications (SCAC): A method of customizing an application of a SPL at pre-compile time by integrating common source code with only the optional and alternative source code selected for the application (Saleh 2005).

Sub-activity diagram: An activity diagram that describes the decomposition of a structured activity node.

Structured activity node: An activity node that groups subordinate activity nodes in an activity diagram (OMG 2007).

System level activity diagram: An activity diagram that describes the sequencing between the activity diagrams associated with the use cases of an application.

System state variable: A variable that encodes the system states of an application. These states can be described in the precondition and postcondition of a use case scenario.

Test case: An instance of a reusable test specification that describes the input and output data values selected to satisfy the predicates in the test specification.

Test design specification: A document in a test plan that specifies the details of a test approach for a software feature or a combination of software features, and identifies the associated tests (IEEE 1998).

Test driver: A software program that invokes a system under test, provides test inputs to the system, controls and monitors the execution of the tests, and reports test results.

Test insertion point: A notation used to uniquely identify and name a location of variation in the decision tables in SCT. *See Static Customization of Test Specifications (SCT).*

Test procedure specification: A document specifying the sequence of actions for the execution of a test (IEEE 1998). In CADeT, it is a document describing the order in which test cases will be executed for an application of the SPL.

Test specification: A document in a test plan that specifies the inputs, predicted results, and a set of execution conditions for a test item (IEEE 1998). In CADeT, this

document describes the predicates, controlled input variables and expected output variables of reusable test specifications for a SPL.

Traceability: A quality criterion that evaluates the ease with which the test specifications can be traced to the requirements models of a SPL.

Use case scenario: A sequence of actions that illustrate the execution of a use case instance (Rumbaugh, Jacobson et al. 2005).

Variability mechanism: A technique that enables the representation and automatic configuration of the variability in an application's requirements, models, implementation and tests.

Variable feature file: A document used by SCAC and DCAC that represents the relationships between the features, insertion points and variable code to the common code of a SPL. *See insertion point, Static Customization of Client Applications (SCAC) and Dynamic Customization of Client Applications (DCAC).*

Variable test step file: A document used by SCT that represents the relationships between the features, test insertion points, and variable test steps to the decision tables and test specifications of a SPL. *See Static Customization of Test Specifications (SCT).*

Variation point: A notation that identifies one or more locations at which change will occur, and the mechanism for a reuser to extend it (Webber 2001).

REFERENCES

- Aksit, M., B. Tekinerdogan, et al. (1996). "Achieving Adaptability through Separation and Composition of Concerns." Special Issues in Object-Oriented Programming: 12-23.
- Allen, G., T. C. Wang, et al. (1994). Applications of Feasible Path Analysis to Program Testing. Proceedings of the 1994 ACM SIGSOFT Int'l Symposium on Software Testing and Analysis, Seattle, Washington, ACM.
- Amla, N. and P. Ammann (1992). Using Z Specifications in Category Partition Testing. 7th Annual Conference on Computer Assurance (COMPASS). Gaithersburg, MD: 3-10.
- Anastasopoulos, M. and C. Gacek (2001). Implementing Product Line Variabilities. Proceedings of the 2001 Symposium on Software Reusability (SSR'01). Toronto, Canada.
- Bassett, P. G. (1987). Frame-Based Software Engineering. IEEE Software. **4**: 9-16.
- Bassett, P. G. (1996). Framing Software Reuse: Lessons from the Real World. Upper Saddle River, New Jersey, Prentice Hall.
- Beizer, B. (1990). Software Testing Techniques. N.Y., Van Nostrand Reinhold.
- Bertolino, A. and S. Gnesi (2003). PLUTO: A Test Methodology for Product Families. 5th Int'l Workshop on Software Product-Family Engineering. Siena, Italy.
- Binder, R. (2002). Testing Object-Oriented Systems: Models, Patterns, and Tools. Reading, MA, Addison-Wesley.
- Bouzeghoub, M. G. Gardarin, et al. (1997). Object Technology Concepts and Methods, International Thomson Computer Press.
- Briand, L. C. and Y. Labiche (2001). A UML-Based Approach to System Testing. Proc. 4th Int'l Conf. on the Unified Modeling Language (UML). Toronto, Canada: 194-208.

- Bryce, R. and C. Colbourn (2006). Test Prioritization for Pairwise Interaction Coverage. Advances in Model-based Testing, St. Louis, Missouri, IEEE Computer Society.
- Chow, T. S. (1978). "Testing Software Design Modeled by Finite State Machines." IEEE Transactions on Software Engineering **SE-4**(3): 178-187.
- Clarke, L., A. Podgurski, et al. (1989). "A Formal Evaluation of Data Flow Path Selection Criteria." IEEE Transactions on Software Engineering **15**(11): 1318-1332.
- Clements, P. and L. Northrop (2002). Software Product Lines Practices and Patterns. Boston, MA, Addison-Wesley.
- Cohen, D. M., S. Dalal, et al. (1997). "The AETG System: An Approach to Testing Based on Combinatorial Design." IEEE Trans. on Software Engineering **23**(7): 437-444.
- DeMillo, R., A and A. J. Offutt (1991). "Constraint-Based Automatic Test Data Generation." IEEE Transactions on Software Engineering **17**(9): 900-910.
- DeMillo, R. A., R. J. Lipton, et al. (1978). Hints on Test Data Selection: Help for the Practicing Programmer. IEEE Computer. **11**: 34-41.
- Diller, A. (1994). Z: An Introduction to Formal Methods, 2nd Edition, Chichester, John Wiley & Sons.
- Geppert, B., J. Li, F. Roessler, and D. M. Weiss (2004). Towards Generating Acceptance Tests for Product Lines. 8th Int'l Conference on Software Reuse. J. a. C. K. Bosch. Madrid, Spain, Springer-Verlag. **3107**: 35-48.
- Gomaa, H. (2000). Designing Concurrent, Distributed and Real-Time Applications with UML, Addison-Wesley.
- Gomaa, H. (2005). Designing Software Product Lines with UML: From Use Cases to Pattern-based Software Architectures, Addison-Wesley.

- Gomaa, H. and E. Olimpiew (2005). The Role of Use Cases in Requirements and Analysis Modeling. Workshop on Use Cases in Model-Driven Software Engineering. Montego Bay, Jamaica.
- Gomaa, H. and E. Olimpiew (2008). Managing Variability in Reusable Requirement Models for Software Product Lines. Proc. 10th International Conference on Software Reuse. Beijing, China.
- Gomaa, H. and M. Saleh (2005). Software Product Line Engineering for Web Services and UML. IEEE Int'l Conference on Computer Systems and Applications. Cairo, Egypt.
- Gomaa, H. and M. E. Shin (2004). A Multiple-View Meta-modeling Approach for Variability Management in Software Product Lines. Int'l Conf. on Software Reuse. J. Bosch and C. Krueger, Springer-Verlag: 274-285.
- Gomaa, H. and D. L. Webber (2004). Modeling Adaptive and Evolvable Software Product Lines Using the Variation Point Model. Hawaii Int'l Conference on System Sciences. Hawaii.
- Grieskamp, W., N. Tillmann, et al. (2004). "Instrumenting Scenarios in a Model-Driven Development Environment." Information and Software Technology **46**: 1027-1036.
- Grindal, M. (2007). Handling Combinatorial Explosion in Software Testing. Department of Computer and Information Sciences. Linkopings, Sweden, Linkopings University.
- Hartmann, J., M. Vieira, et al. (2004). A UML-based Approach for Validating Software Product Lines. 1st Int'l Software Product Line Testing Workshop (SPLiT '04). Boston, U.S.A., Avaya Labs Technical Report: 58-64.
- Hoffer, J. A., J. F. George, et al. (2005). Chapter 10: Designing Databases. Modern System Analysis and Design. Upper Saddle River, NJ, Pearson Education, Inc.: 328-367.
- IEEE (1998). IEEE Standard for Software Test Documentation. IEEE Std 829-1998.

- Jacobson, I., M. Christerson, et al. (1992). Object-Oriented Software Engineering: A Use Case Driven Approach. Reading, MA, Addison-Wesley.
- Jacobson, I., Martin Griss, et al. (1997). Software Reuse: Architecture, Process and Organization for Business Success. Reading, Massachusetts, Addison-Wesley.
- Jarzabek, S., W. Chun Ong, and Hongyu Zhang (2003). "Handling Variant Requirements in Domain Modeling." Journal of Software and Systems **68**(3): 171-182.
- Jenkins, B. (2005). Jenny: A Pairwise Testing Tool at <http://burtleburtle.net/bob/math/jenny.html>. **Last accessed on March 2008.**
- Kamsties, E., K. Pohl, et al. (2003). Testing Variabilities in Use Case Models. 5th Int'l Workshop on Software Product-Family Engineering Siena, Italy.
- Kang, K. (1990). Feature Oriented Domain Analysis. Pittsburg, PA, Software Engineering Institute.
- Kang, K. C., S. Kim, et al. (1998). "FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures." Annals of Software Engineering **5**: 143-168.
- Kiczales, G. (1996). "Aspect-Oriented Programming." ACM Computing Surveys **28**(4es): 154.
- Kiczales, G., J. Lamping, et al. (1997). Aspect-Oriented Programming. European Conference on Object-Oriented Programming. Finland, Springer-Verlag.
- Kishi, T. and N. Noda (2004). Design Testing for Product Line Development based on Test Scenarios. Software Product Line Testing Workshop (SPLiT). Boston, MA: 19-26.
- Kishi, T., N. Noda, et al. (2005). Design Verification for Product Line Development. Software Product Line Conference (SPLC '05). Rennes, France.

- Kolb, R. (2003). A Risk-Driven Approach for Efficiently Testing Software Product Lines. 2nd Int'l Conference on Generative Programming and Component Engineering. Erfurt, Germany.
- Kruchten, P. (1995). Architectural Blueprints - The "4+1" View Model of Software Architecture. IEEE Software. **12**: 42-50.
- Laski, J. W. and B. Korel (1983). "A Data Flow Oriented Testing Strategy." IEEE Transactions on Software Engineering **9**(3): 347-354.
- Lee, K., K. C. Kang, et al. (2006). Combining Feature-Oriented Analysis and Aspect-Oriented Programming for Product Line Asset Development. 10th Int'l Software Product Line Conference. Baltimore, MD, IEEE Computer Society: 103-112.
- Loughran, N. and A. Rashid (2004). Framed Aspects: Supporting Variability and Configurability for AOP. Software Reuse: Methods, Techniques and Tools: 8th Int'l Conference (ICSR). Madrid, Spain.
- Loughran, N., A. Rashid, et al. (2004). Supporting Product Line Evolution with Framed Aspects. 3rd AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS). Lancaster, UK.
- M. L. Griss, J. Favaro, et al. (1998). Integrating Feature Modeling with the RSEB. International Conference on Software Reuse. Victoria, Canada, IEEE Computer Society.
- Mayrhauser, A. v., R. T. Mraz, et al. (1994). Domain Based Testing: Increasing Test Case Reuse. Proceedings of the IEEE Int'l Conference on Computer Design. Boston, MA: 484-491.
- Mayrhauser, A. v., R. T. Mraz and P. Ocken (1996). On Domain Models for System Testing. Proc. of the 4th Int'l Conf. on Software Reuse. Orlando, FL: 114-123.
- Mayrhauser, A. V. and N. Zhang (1999). "Automated Regression Testing using DBT and Sleuth." Journal of Software Maintenance: Research and Practice **11**(2): 93-116.

- McGregor, J. D. (2001). Testing a Software Product Line, SEI.
- McGregor, J. D., P. Sodhani, et al. (2004). Testing Variability in a Software Product Line. Software Product Line Testing Workshop (SPLiT). Boston, MA, Avaya labs: 45-50.
- Miller, E., M. Paige, et al. (1974). Structured Techniques of Program Validation. COMPCON '74. San Francisco, USA: 161-164.
- Morell, L. and B. Murrill (1993). "Semantic Metrics through Error Flow Analysis." Journal of Systems and Software **20**(3): 253-265.
- Morell, L. J. (1990). "A Theory of Fault-Based Testing." IEEE Transactions on Software Engineering **16**(8): 844-857.
- Murrill, B., L. Morell, et al. (2002). A Perturbation-Based Testing Strategy. Proceedings of the 8th Int'l Conf. on Engineering of Complex Computer Systems. Greenbelt, MD, IEEE Computer Society: 145-152.
- Muthig, D. and T. Patzke (2003). "Generic Implementation of Product Line Components." Lecture Notes in Computer Science **2591/2003**: 313-329.
- Nebut, C., F. Fleurey, et al. (2003). A Requirement-Based Approach to Test Product Families. Software Product-Family Engineering: 5th Int'l Workshop. Siena, Italy.
- Offutt, J. (1992). "Investigations of the Software Testing Coupling Effect." ACM Transactions on Software Engineering and Methodology **1**(1): 3-18.
- Offutt, J. and A. Abdurazik (1999). Generating Tests from UML Specifications. Proc. of the 2nd IEEE Int'l Conf. on the Unified Modeling Language. Fort Collins, CO. **1723**: 416-429.
- Offutt, J., Y. Xiong, et al. (1999). Criteria for Generating Specification-Based Tests. Proceedings of the 5th IEEE Int'l Conference on Engineering of Complex Computer Systems (ICECCS '99). Las Vegas, NV: 119-129.

- Olimpiew, E. M. and H. Gomaa (2005). Executing Reusable System Tests for Applications Derived from Software Product Lines. 2nd Int'l Workshop on Software Product Line Testing (SPLiT 2005). Rennes, France.
- Olimpiew, E. M. and H. Gomaa (2005). Model-based Testing for Applications Derived from Software Product Lines. Advances in Model-based Testing Workshop. St. Louis, Missouri.
- Olimpiew, E. M. and H. Gomaa (2006). Customizable Requirements-Based Test Models for Software Product Lines. 3rd Int'l Workshop on Software Product Line Testing (SPLiT 2006). Baltimore, MD, Mannheim University of Applied Sciences.
- OMG (2007). Unified Modeling Language: Superstructure, version 2.1, Object Management Group.
- Ostrand, T. J. and M. J. Balcer (1988). "The Category-Partition Method for Specifying and Generating Functional Tests." Communications of the ACM **31**(6): 676-686.
- Parnas, D. L. (1978). Designing Software for Ease of Extension and Contraction. Proc. of the 3rd Int'l Conference on Software Engineering. Atlanta, Georgia, U.S.A.: 264 - 277.
- Pesonen, J., M. Katara, et al. (2005). Evaluating an Aspect-Oriented Approach for Production-Testing Software. Proceedings of the 4th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software. Chicago, Illinois, College of Computer and Information Science, Northeastern University: 36-40.
- Poston, R. M. (1996). Automated Testing from Object Models. Automating Specification-Based Software Testing. R. Poston. Los Alamitos, CA, IEEE Computer Society Press: 24-35.
- Rapps, S. and E. Weyuker (1985). "Selecting Software Test Data Using Data Flow Information." IEEE Transactions on Software Engineering **11**(4): 367-375.
- Reuys, A., E. Kamsties, et al. (2005). "Model-Based Testing of Software Product Families." Lecture Notes in Computer Science **3520**: 519-534.

- Rothermel, G. and M. J. Harrold (1994). A Framework for Evaluating Regression Test Selection Techniques. Proc. of the 16th Int'l Conf. on Software Engineering. Sorrento, Italy: 201-210.
- Rumbaugh, J., I. Jacobson, et al. (2005). The Unified Modeling Language Reference Manual, 2nd Ed. Boston, MA, Addison-Wesley.
- Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy and W. Lorensen (1991). Object-Oriented Modeling and Design. Englewood Cliffs, NJ, Prentice Hall, Inc.
- Saleh, M. (2005). Software Product Line Engineering based on Web Services. Information and Software Engineering. Fairfax, VA, George Mason University: 290.
- Saleh, M. and H. Gomaa (2005). Separation of Concerns in Software Product Line Engineering. Workshop on the Modeling and Analysis of Concerns in Software Product Line Engineering. St. Louis, Missouri.
- Scheidemann, K. (2006). Optimizing the Selection of Representative Configurations in Verification of Evolving Product Lines of Distributed Embedded Systems. 10th Int'l Software Product Line Conf. Baltimore, MD, IEEE Computer Society Press: 75-84.
- Stocks, P. and D. A. Carrington (1996). "A Framework for Specification-Based Testing." IEEE Transactions on Software Engineering **22**(11): 777 -793.
- Tai, K.-C. (1996). "Theory of Fault-Based Predicate Testing for Computer Programs." IEEE Transactions on Software Engineering **22**(8): 552 - 562.
- Tevanlinna, A., J. Taina, et al. (2004). "Product Family Testing: A Survey." SIGSOFT Software Engineering Notes **29**(2): 1-6.
- Tirila, A. (2002). Variability Enabling Techniques for Software Product Lines. Department of Information Technology, Tampere University of Technology: 59.

- Vauthier, J.-C. (2006). Decision Tables: A testing technique using IBM Rational Functional Tester at <http://www-128.ibm.com/developerworks/rational/library/jun06/vauthier/>. IBM DeveloperWorks, IBM. **Last accessed on March 2008.**
- Vieira, M., Johanne Leduc, et al. (2006). Automation of GUI Testing Using a Model-Driven Approach. Proceedings of the 2006 Int'l workshop on Automation of Software Test Shanghai, China, ACM: 9-16.
- Vonteru, V. (2001). Java Implementation of Banking System Case Study. Fairfax, VA, George Mason University.
- Warmer, J. and A. Kleppe (1999). The Object Constraint Language: Precise Modeling with UML. Reading, MA, Addison-Wesley.
- Webber, D. (2001). Variation Point Model for Software Product Lines. Fairfax, VA, George Mason University.
- Weiss, D. M. and C. T. R. Lai (1999). Software Product-Line Engineering: A Family-Based Software Development Process. Reading, MA, Addison-Wesley.
- Yin, R. K. (2003). Case Study Research Design and Methods. Thousand Oaks, California, Sage Publications.
- Yourdon, E. (1989). Modern Structured Analysis. Upper Saddle River, NJ, Yourdon Press.
- Zave, P. (2004). FAQ Sheet on Feature Interaction at <http://www.research.att.com/~pamela/faq.html>. **Last accessed on March 2008.**
- Zhang, H. and S. Jarzabek (2004). "XVCL: A Mechanism for Handling Variants in Software Product Lines." Science of Computer Programming **53**(3): 381-407.

CURRICULUM VITAE

Erika Mir Olimpiew was born on December 25, 1972, in Belgium. Originally from Brasil, she became an American citizen in May 1998. She graduated from South Lakes High School, in Reston, Virginia, in 1991. She received her Bachelor of Science degree in Mathematical Sciences and her Master of Science degree in Computer Science from Virginia Commonwealth University in 1995 and in 1997, respectively. Ms. Olimpiew worked as a software engineer for a satellite communications company from July 1996 to December 2002, and as a teaching assistant for George Mason University from January 2003 until May 2008.