TEST CASES REDUCTION IN SOFTWARE PRODUCT LINE USING REGRESSION TESTING

by

Vivek Hari Mugunthan

Submitted in partial fulfilment of the requirements
for the degree of Master of Computer Science

at

Dalhousie University
Halifax, Nova Scotia
March 2012

© Copyright by Vivek Hari Mugunthan, 2012

DALHOUSIE UNIVERSITY

FACULTY OF COMPUTER SCIENCE

The undersigned hereby certify that they have read and recommend to the Faculty of Graduate Studies for acceptance a thesis entitled "**TEST CASES REDUCTION IN SOFTWARE PRODUCT LINE USING REGRESSION TESTING**" by Vivek Hari Mugunthan in partial fulfilment of the requirements for the degree of Master of Computer Science.

Dated: March 28, 2012

Co-Supervisors: ______________________

____

Reader: ______________________

DALHOUSIE UNIVERSITY

DATE: March 28, 2012

AUTHOR: Vivek Hari Mugunthan

TITLE: **TEST CASES REDUCTION IN SOFTWARE PRODUCT LINE USING REGRESSION TESTING**

DEPARTMENT OR SCHOOL: Faculty of Computer Science

DEGREE: MCSc CONVOCATION: October YEAR: 2012

Permission is herewith granted to Dalhousie University to circulate and to have copied for non-commercial purposes, at its discretion, the above title upon the request of individuals or institutions. I understand that my thesis will be electronically available to the public.

The author reserves other publication rights and neither the thesis or extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

The author attests that permission has been obtained for the use of any copyrighted material appearing in the thesis (other than the brief excerpts requiring only proper acknowledgement in scholarly writing), and that all such use is clearly acknowledged.

Signature of Author

Table of Contents

LIST OF FIGURES vii

ABSTRACT viii

LIST OF ABBREVIATIONS USED ix

GLOSSARY x

CHAPTER 1: INTRODUCTION 1

1.1 CHALLENGES 2

1.2 OBJECTIVE 3

CHAPTER 2: LITERATURE SURVEY 5

2.1 SOFTWARE PRODUCT LINE TESTING 5

2.2 MODEL-BASED TESTING FOR SOFTWARE PRODUCT LINES 6

2.3 REGRESSION TESTING IN SOFTWARE PRODUCT LINE 8

2.4 TEST SELECTION IN SOFTWARE PRODUCT LINE 8

CHAPTER 3: A PROPOSED FRAMEWORK FOR SOFTWARE PRODUCT LINE TESTING 11

3.1 OVERVIEW 11

3.2 OUTLINE 12

3.3 REQUIREMENT SPECIFICATION FOR A SPL 14

3.4 LEVEL I: MODELING SOFTWARE PRODUCT LINE USING FEATURE MODEL DIAGRAM 14

3.4.1 Concepts of Feature Model *15*

3.4.2 Derivation of Specific Feature Models from the SPL *16*

3.5 LEVEL II: REUSABILITY OF TEST MODELS 16

3.5.1 Object Model Diagram *17*

3.5.2 State Chart diagram *18*

3.6 LEVEL III: TEST CASE DERIVATION 19

3.6.1 Sequence Diagram *19*

3.7 TEST CASE REPOSITORY 21

3.8 COMPARATOR 21
3.9 LEVEL IV: TEST CASE RUN AND EXECUTION 21

CHAPTER 4: OPERATIONS 22

4.1 PHASE A 22
4.1.1 Pure::variants 22
4.1.2 Rational Rhapsody 23
4.1.3 Procedure for Phase A 23
4.2 PHASE B 24
4.2.1 Test Conductor 24
4.2.2 Procedure for Phase B 25
4.3 PHASE C 26
4.3.1 IBM Clear Case 26
4.3.2 Procedure for Phase C 27
4.4 PHASE D 29

CHAPTER 5: IMPLEMENTATION 30

5.1 MODELING VENDING MACHINE SPL USING FEATURE MODEL DIAGRAM 31
5.2 GENERATING OBJECT MODEL FOR THE FEATURE MODEL OF VENDING MACHINE... 33
5.3 GENERATING STATE CHART DIAGRAM FOR COMPONENTS OF OBJECT MODEL DIAGRAM 34
5.4 UNIT TESTING FOR EACH COMPONENTS OF OBJECT MODEL DIAGRAM 35
5.5 GENERATION OF SYSTEM TEST CASES BY ATG 37
5.6 COMPARATOR 37
5.6.1 Graphical Comparison 38
5.7 OUTPUT REPORT OF TEST CLASSIFIERS 39
5.7.1 Obsolete Test Cases 40
5.7.2 Reusable Test Cases 40
5.7.3 Re-testable Test Cases 41

CHAPTER 6: EVALUATION 43

6.1 DIFFICULTY IN ASSESSING A CORRECT SEPARATION OF THE COMMON AND PRODUCT SPECIFICATIONS 43
6.2 TEST CASE REDUCTIONS ON THE VENDING MACHINE SPL 43
CHAPTER 7: DISCUSSION 45
CHAPTER 8: CONCLUSION 48
REFERENCES 50

LIST OF FIGURES

Figure 2-1 Model-based Testing in a SPL [2] 6
Figure 3-1 The proposed framework for SPL Testing 13
Figure 3-2 Weather Sensor Feature Model 15
Figure 3-3 Object diagram for Weather Sensor 17
Figure 3-4 State Chart Model for Weather Sensor 19
Figure 3-5 Sequence Diagram for Weather Sensor 20
Figure 4-2 Test Conductor Features [46] 25
Figure 4-3 Comparison window from Diffmerge 28
Figure 4-4 Diffmerge Report differences 28
Figure 5-1 Feature Model Diagram for Vending Machine SPL. 31
Figure 5-2 Deriving Product Variants 32
Figure 5-3 Object Model Diagram for Vending Machine SPL 33
Figure 5-4 State Diagram for Choice panel of Vending Machine SPL 34
Figure 5-5 State chart diagram for Choice panel of 'Simple' Vending Machine 35
Figure 5-6 Test architecture of Vending Machine SPL 'Choice Panel' Component 36
Figure 5-7 Test cases of selecting water from Choice panel Component 37
Figure 5-8 Snapshot of Comparator 38
Figure 5-9: Graphical Comparison between SPL and a product variant 39
Figure 5-10 Obsolete test case of Water Vending Machine 40
Figure 5-11 Reusable test case of Water Vending Machine 41
Figure 5-12 Re-testable Test case of Vending Machine SPL and Simple Vending Machine 42

ABSTRACT

Application Engineering is a field where software organizations develop software products from a predefined Software Product Line. The time and cost allotted to come up with a new product variant is limited. Lack of systematic support in testing leads to redundancy. Redundancy in this context can be found in test-cases that do not contribute towards fault-detection and testing leads to an increased testing effort. This thesis work proposes a framework to reduce the testing effort, aimed at avoiding testing redundancy. Feature Model diagrams have been constructed from the assumed specification requirements. These Feature Model diagrams have been used to derive test models such as Object Model diagram and State Chart diagram. Unit testing and System testing have been performed on test models to obtain test cases that have been stored in the repository. Regression testing has been applied to these test cases to classify them into Reusable, Re-testable and Obsolete.

LIST OF ABBREVIATIONS USED

SPL	-	Software Product Line
UML	**-**	Unified Modified Language
SysML	-	Systems Modeling Language
ATG	-	Automatic Test Generator
SUT	-	System Under Test
TC	**-**	Test Conductor
SCM	-	Software Configuration Management

GLOSSARY

Software Product Line - A Software Product Line (SPL) is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.

UML (Unified Modeling Language): It is a standard notation for the modeling of real-world objects as a first step in developing an object-oriented design methodology. Its notation is derived from and unifies the notations of three object-oriented design and analysis methodologies.

Regression Testing: Regression testing is performed when changes are made to existing software; the purpose of regression testing is to provide confidence that the newly introduced changes do not obstruct the behaviors of the existing, unchanged part of the software.

Application Engineering in SPL: Application engineering is the phase where an individual software product is built in accordance with the specific product requirements.

Domain Engineering in SPL: Domain engineering is the phase where the variability and commonality of a system is identified, and a core set of reusable assets is developed.
The assets include the requirements, design model, implementation, testing and other assets used in software development.

Variability: The variable components across the structure of Software Product Line
Variation points: the specific representations of the variability in software artefacts.
Variants: The representation of a particular instance of a variable.

Software Components: A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

CHAPTER 1: INTRODUCTION

The software industry faces variety of challenges, with their main intention being to deliver quality software to meet the requirements of the customers and clients. The introduction of Software Development Life Cycles has aided in the development of reliable software to a great extent but still it has been difficult to provide software that satisfies the needs of the customer [1] [2]. The process of developing software has been dependent on three major factors: cost, time and quality. The organizations have followed the process of combining software that share common features but differ in their purpose. Software Product Line has been adapted by many software organizations recently with a view of reducing cost and time, meanwhile improving the quality of the products [3, 4, 5]. In addition to these features, Software Product Lines also ensure low cost maintenance, mass customization, alignment and improved efficiency of the product [4] [5]. The method of extracting commonality and variability among products seems to be practical because organizations tend to produce families of similar systems, differentiated by features. There also remains the potential for proactive reuse of these systems and procedures. Software Product Lines have varied from the conventional methods of single system development in the following two ways. First, the process of development has been divided into two different procedures: Domain Engineering, where systems are grouped based on their common features and Application Engineering, where individual systems are instantiated and built. Second, the differences and variations in these systems have been recorded and controlled [1].

Testing has been one of the most significant features of the Software Development Life Cycle that finds faults in the software. Testing of a single system has been primarily focused on testing the code of the system, identifying problems and rectifying those problems; but when the concept of testing needs to be applied to a whole family of systems, code-based testing could prove to be tedious and time-consuming. Hence, testing can be conducted at a higher level, in addition to the code level and use this higher level testing to decrease the redundancy among code-level tests. System models provide a representation of the functionality of the system. Testing these models may be considered a form of black-box testing as it performs functional tests. This method has been termed

‘Model-based Testing’. This method differs from the conventional testing process in that it provides an abstract test suite from which test cases can be derived. But these test cases cannot be used to test the actual Software Product Line in question. Equivalent test cases that represent these abstract test cases need to be identified from the system.

Testing variability is one of the challenges in Software Product Line engineering. The variability of a product line specifies the differences among systems to be built. Functionality may be considered as a variant whenever it is not planned to be part of all applications. For example, credit card payment is a variant in an eShop, because it is not a part of all applications. There are various methods to cope with variability, but adaptation of the test models helps in test case derivation, and representation of test cases[2] [3].

Meanwhile, the testing procedures that apply to an individual product also apply to the Software Product Line. It helps to identify the differences in system processes that have developed over time. In practice, these processes are accumulated under a single component to maintain a semantic relationship between the functions of the system. Unit testing provides unit of code behaves as expected in the functionality of these components. These unit tests can then be reused on different versions of the product line component sets and individual products. System testing is that the entire system is tested as per the requirements. Black-box type testing that is based on overall requirements specifications, covers all combined parts of a system. In order to ensure that these changes do not introduce new errors in the system, regression testing is performed. Regression testing is testing software that has been modified in order to ensure that additional bugs have not been introduced. When software is enhanced, testing is often done only on the new features. However, adding source code to software often introduces errors in other routines, and many of the old and stable functions must be retested along with the new ones.

1.1 Challenges

Development of Software Product Line from each component and mass configuration of each product variant is a complex task. Software Product Line Testing is a tedious

process that involves testing test cases that have been generated from the product line and product variants that tends to evolve over time. Each product variant has to only be developed according to customer requirements and complete testing is not required for each product since they share common components from SPL. To test a complete Software Product Line, it is necessary to test all the components that have been developed at domain engineering level and also to test all the configured variants developed using those components at application engineering level. This is not an easy task as it involves keeping track of each and every component change. To do this, a certain selection process is helps to extract only specific test cases to test the variants.

All the possible product variants and its components may evolve with time and this number may grow over a period of time. So, tracking all the variation points distributed all over the product line and testing them becomes more complex and so it requires more unit and system level testing. Since components may also tend to evolve from time to time, unit testing helps to keep track of those components.

Each product is derived from the same product line specification and this gives us opportunity in locating the redundancy of features that would help reduce the number of test cases. The research question here is: 'How to reduce redundant testing in Software Product Line?' In conjunction with the above mentioned challenge, there also exists an unanswered question: How to classify and reduce those redundant test cases from large pool of test case suite for a Software Product Line. Test cases are derived from the existing test model which helps to reuse those models in each product variant. The regression testing approach helps to not only select the test cases from a test suite, but also to reduce the number of test cases by identifying redundant test cases without reducing the efficiency of the product line. An effective test selection strategy may help to avoid the redundant testing activities and to explore the reusable testing scenarios. Various selection strategies have been studied and compared to determine the right technique that is more appropriate for the current situation.

1.2 Objective

The main objective of this work is to propose a framework and thereby validate using a Vending Machine SPL. This thesis work uses regression testing selection method to

reduce the testing effort by extracting obsolete, reusable, and re-testable test cases from a repository of test cases for a Software Product Line. The test selection method has been adapted only to identify those redundant test cases.

CHAPTER 2: LITERATURE SURVEY

Software Product Line Testing is a wide field for researchers to come up with new methods and approaches to reduce the testing effort with respect to both time and cost. Efficient selective testing approaches helps in developing a Software Product Line in many software industries to reduce the development time for product variants by reusing their components. Identifying the redundant test cases helps to reduce the testing effort in developing product variants in Software Product Line.

2.1 Software Product Line Testing

A Software Product Line can be defined as a set of software products holding a common platform [4]. A set of products can be developed from this common structure using its set of generic product components [5]. The process of developing the platform is named domain engineering, and the process of deriving specific products from the platform is named application engineering [5] [6]. Also, domain testing and application testing, are performed. The variable components across the structure of Software Product Line are called Variability and the specific representations of the variability in software artefacts are called variation points, while the representation of a particular instance of a variable characteristic is called a variant [4, 5, 6].

The main objective of testing approaches for single applications is to write tests that reveal any faults in the corresponding applications. Many SPL based testing methods are adapted from strategies of single system based application testing [6] [7]. But, Software Product Line testing differs from testing single applications in the methods used to create, manage and reuse those testing assets.

McGregor created a testing process [8] using the specification requirements of core assets and application of a Software Product Line. This testing strategy completely describes the management of test assets for the test specification developed within a SPL process. McGregor faced many problems in Software Product Line testing on reusing the generic test assets and results, also selecting the variants that need to be tested from large pool of variants [9] [10]. In order to reduce the test effort, McGregor proposes a combinatorial test design where pairwise combinations of variants are systematically selected to be

tested instead of all possible combinations [9]. His testing strategies in the Software Product Line reduce the cost and time in generating and maintaining the usage of reusable test assets across many product variants. McGregor's was based on UML designing for SPL and selecting only the appropriate variants that are needed to be tested.

2.2 Model-based Testing for Software Product Lines

Model-based Testing is software testing in which test cases are generated in whole or in part from a model that describes some (usually functional) aspects of the system under test (SUT) [11, 12, 13].

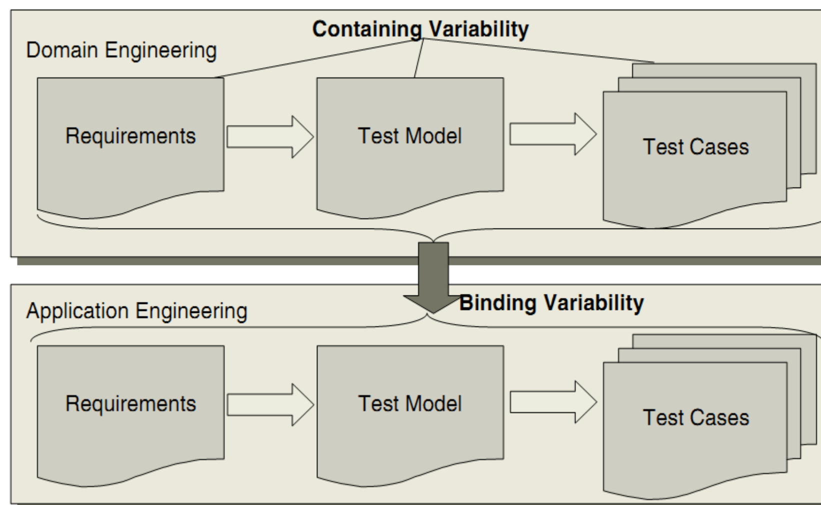


Figure 2-1 Model-based Testing in a SPL [2]

Model-based testing can be performed in domain engineering as well as in application engineering. It is conducted in domain engineering for two reasons. First, the domain test model and the test cases are used to facilitate an early validation of the domain requirements. Second, the test cases are created for reuse in application engineering. There are several advantages of Model-based testing such as test cases can written in systematic order, e.g. stopping rules and can be used repeatedly [11]. Model-based Testing can be used easily for automated test case generation which is very useful in large software applications [12] [13]. The test models are created by test engineers to validate the specification requirements of the client. Defects in requirements can be detected during the development of the test model, which is cheaper than correcting them

in later development phases. These benefits can be realized in Software Product Line engineering by adapting Model-based testing [14]. It also helps to keep track of code based creation of test cases since all the changes in models may not be reflected in the attributes of the corresponding code. This may lead to many serious errors in the Software Product Line Testing.

Hartman et al. presented a Model-based testing approach on UML design testing tools and it was adapted by major Model-based Software Product Line researches [13, 14, 15]. The derivation of test cases for domain engineering from Model-based testing are represented in use cases, state chart and class diagram which depict the variability in the test cases [16]. These derivations of models are not enough to describe all the specification from the client and Feature Model diagrams helps to describe the depth of requirements. Also, many researches [16, 17, 18] have been proposed and evaluated on designing test automation software which is based on correspondence of variability and commonality in a Software Product Line and its product variants. However, adaption of test automation throughout SPL reduces the testing effort; it is susceptible to errors that can impact testing process due to traceability of all test cases and it can be avoided by tracking them with the specification requirements of the clients.

Condron proposed a strategy [19] to generate all test cases in Model-based Software Product Line Testing using a combination of test automation frameworks from different locations of testing assets. Two algorithms [20] [21] have been proposed and implemented to automatically generate product variant test cases from SPL requirements, expressed in UML design. These proposed techniques were only described on selecting test variants from large pool of testing variants. Most of these researches were only concentrated on modeling product variants rather than test cases. So, methods to model and classify test cases that can reduce the testing effort are required.

Another approach named ScenTED [22] has the detailed analysis of test case generation from UML models. A few research studies [21, 22, 23] in Software Product Line on Model-based Testing were using diagrams, stereotypes and tagged values from UML notations which were illustrated through experimental results. Kang et al. proposed yet

another process [23] based on UML use cases describing variability and commonality. Dueñas et al. proposed another approach [24], based on the UML testing profile. These methods were based on building a testing profile using only use cases. It is useful to derive the test cases with more UML diagrams such as State Chart Diagrams, Object Model diagrams and Sequence diagrams.

2.3 Regression Testing in Software Product Line

Emelie Engström states "Regression testing is not an isolated one-off activity, but rather an activity of varying scope and preconditions, strongly dependent on the context in which it is applied" [25]. There exist techniques for regression test selection for single applications that can be easily adapted on any context of testing process. These approaches have been implemented in Software Product Line Testing to reduce the testing effort considering user acceptance in the end. By focusing the testing on changes to the system and re-executes past tests to ensure that recent changes haven't broken other parts that the code was _not_ changed, by definition. Regression testing aims at verifying that previously working software still works after a change [25]. Since many variants derived from the common platform and testing to each product with respect to variants is not an easy task.

In order to test the complete product line, all generic components have to be tested for all the products. A major challenge in Software Product Line testing is that a large number of test cases will be stored for both product line and product variants [25] [26]. Such testing throughout the product line is infeasible and a test selection strategy is required. The variants that are derived from the Software Product Line are closely related and large amount of testing can be saved and removed by identifying the redundant test cases [26]. Recent research studies have considered the challenge to evaluate possible approaches aiming to minimize the amount of redundant testing in Software Product Line [25, 26, 27, 28]. The regression test selection approach can be adapted to minimize those redundant test cases from large pool of test cases.

2.4 Test Selection in Software Product Line

Several techniques for regression test selection are proposed and evaluated empirically by researchers and have also been used for the Software Product Line context. The adoption

of a regression test selection technique is useful in many scenarios of Software Product Line testing. It selects a set of test cases from existing test suites to test the SPL or its product variants, avoiding the execution of all test cases [26] [27]. However, extraction of these test cases may lead to error due to reduction of the test case that is required to find the fault. The classification of test cases may help to select the test case that can reduce and reuse effectively. For example, in one Microsoft Product Line, the reduction of one test case may save testing resources such as cost and time all over the product line or its one product variant [27]. Also, the test selection technique is only justifiable when the cost to select test cases is less than running the entire test suite.

In this sense, taking advantage of regression test selection methods, selecting and classifying test cases helps to reduce testing effort. Research and empirical studies have been doing regression testing in Software Product Line since 1980 [28, 29, 30]. The main focus has been on 'how to select tests based on information about changes in the system since the latest tested version' [28]. However, test case selection can be adapted from single application and needs to be used in SPL with proper methods without reducing its efficiency. The prioritization, selection and classification of test cases in a regression test suite helps to reduce the testing effort [20] [31]. To bring structure to the topics, researchers have typically divided the field of regression testing into test selection, modification identification, test execution, and test suite maintenance. This review is focused on test selection techniques for regression testing.

Regression test selection strategies proposed in the literature are not easily applicable in the product line context [25, 26, 27, 28]. The research strategies discussed above have mostly concentrated on applying regression testing for a single system. Paulo Anselmoda et al, came up with an alternative [26] to apply regression testing approach to the entire Software Product Line and its variants. His work was aimed at identifying reusable test cases, significant test cases and prioritizing them. This way of identifying reusable test cases reduced the testing effort considerably. Sebastian Oster et al, proposed and implemented a tool [32] called Mosopolite, pairwise configuration selection component on the basis of a Feature Model. This component implements a 100% pairwise interaction and it uses flattening algorithm. This tool can also be used as plugin in any testing tool

while constructing Feature Model diagrams and the product variants that have to be tested can be reduced. This method helped to keep track of changes in the Model-based design and ensure its proper working even after the changes were introduced. Such a framework methodology is required to avoid redundant testing.

CHAPTER 3: A PROPOSED FRAMEWORK FOR SOFTWARE PRODUCT LINE TESTING

The goal of our research is to improve the control of the testing and reduce the amount of redundant testing in the product line context by applying regression test selection strategies [25] [26]. Regression testing is highly inviting in the field of SPL since each product derived from the source and act as different variants [34]. By making use of this advantage, regression test selection methods can be used to select sets of efficient test cases that can reduce the testing effort. The test cases for Regression Test Selection are categorized into three different types as follows [25, 26, 27]

- **Reusable Tests:** These test cases are used for testing an unmodified portion of the requirements and that need to be rerun for regression testing.
- **Re-testable Tests:** A test case that should be rerun as they represent the modified version in the model and separates those test cases that need not to be rerun for regression testing.
- **Obsolete Tests:** A test case that can be ignored in the newer version of the SPL, as it has become invalid in the context. Classification of tests cases as obsolete means that they need not be either re-tested or reused.

3.1 Overview

The adaption of Model-based Testing in this proposed framework helps to reduce the testing effort in SPL since the code based creation would cost more time and money. The model diagrams such as Feature Model, State chart, Object Model and sequence diagrams have been used throughout this proposed approach by considering their various advantages [35]. Software Product Lines are constructed using Feature Models from the specification obtained from the consumers and their product variants are derived using the same model. These Feature Models provide a method for commonalities and variabilities within the SPL.

Each and every product variant is also represented in the Feature Model. Once we have common and variable features derived from the SPL, corresponding Object Models and state diagrams can be extracted. An Object Model provides a structural view by each class/component and it helps to understand the structure and its relationship among

different features of a SPL and its variants. A State Chart model gives the behavior view of each component obtained from the Object Model. These test models can be reused among SPL and product variants.

Unit testing provides a set of test cases for each component from the Object Model diagram. Test cases scenarios are obtained from the state chart model for the system under test. Each test case has been represented using sequence diagram, which depicts the object interaction in time sequence. Unit Test cases and Test case scenarios are grouped for SPL and product variants and they are stored in a repository. Regression testing is performed by comparing the derived SPL test cases with its product variants. These test cases are categorized into re-testable, obsolete and reusable test cases. The obtained test suite for each product is then executed through the test run.

3.2 Outline

Figure 3-1 describes complete framework of the proposed approach for testing a Software Product Line. The procedure has been represented as Figure segregated into four different levels and four phases describing the complete framework. Level I explain how the Feature Model for a Product Line is created from the specification and each product variants are derived. Level II, describes the reusability of test models between SPL and product variants. Level III and IV deals with test case extraction and derivation from the test models. The four phases represent the functionality of the framework as follows.

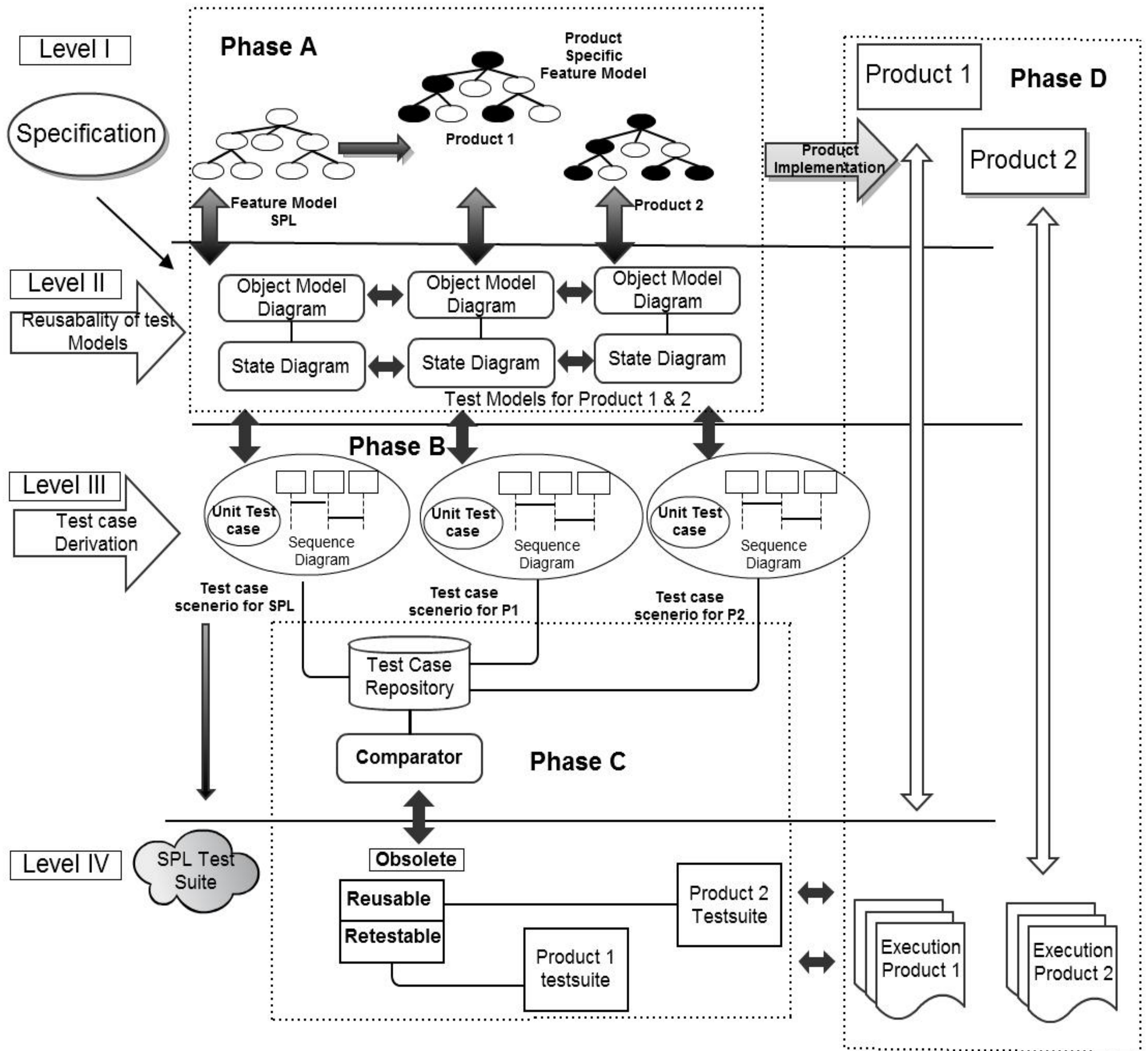


Figure 3-1 The proposed framework for SPL Testing

- In Phase A, a Feature Model is used to model the commonality and variability within the SPL and corresponding Object Model diagram and state diagrams are generated from Feature Model.
- In Phase B, specific product Feature Models are generated from the SPL and the Object Model and state diagram are also obtained for each product. Also, reusability of test models takes place between SPL and variants. Unit test cases are derived for

each component from the Object Model diagram for the SPL and product variants. Also, test case scenarios which are represented as sequence diagrams are derived from the State Diagram.

- In Phase C is the comparator. It compares and stores all the unit test cases and their scenarios derived from sequence diagram. Obsolete, Reusable and Re-testable test cases can be obtained from the comparator are mapped to suit variant products to construct their test suites.
- Finally in Phase D, test executions for the product are done using their corresponding test suites.

3.3 Requirement specification for a SPL

Requirement specification includes all the definitions of commonalities and identifying variations within the SPL. These requirements represent large reuse of components among product line and product variants. A requirement has the product line scope as one of its inputs–an artifact that does not exist outside the product line context. A Feature Model for a Software Product Line has been constructed using the specification requirements of a customer demand and corresponding Feature Models can also be derived for each product variant [35].

3.4 Level I: Modeling Software Product Line using Feature Model Diagram

The term Feature Model was first coined by Kang in 1990 and derived the method Feature-Oriented Domain Analysis (FODA) [36]. Then, Feature Modeling became the key for success in developing various SPLs and still various ideas are being proposed using this method. A Feature Model is the graphical representation of different combinations of features and they are frequently used to describe variable and common parts within the SPL [37]. Features are arranged in a directed cyclic graph which has parent and child node. Feature variability is represented by the arcs and groupings of features. There are four different types of feature groups: “mandatory", “optional", "alternative" and “or” as part of the Feature Model combining a hierarchical decomposition of features into sub features using different sorts of node notations [38].

The purposes of FMs are summarized as follows [36] [37].

- to describe feature commonalities and variabilities,
- to picture dependencies and constraints between features, and
- to specify permitted and forbidden combinations of features.

3.4.1 Concepts of Feature Model

The Basic concepts of Feature Model diagram have been explained using the example Weather sensor Feature Model Figure 3-2.

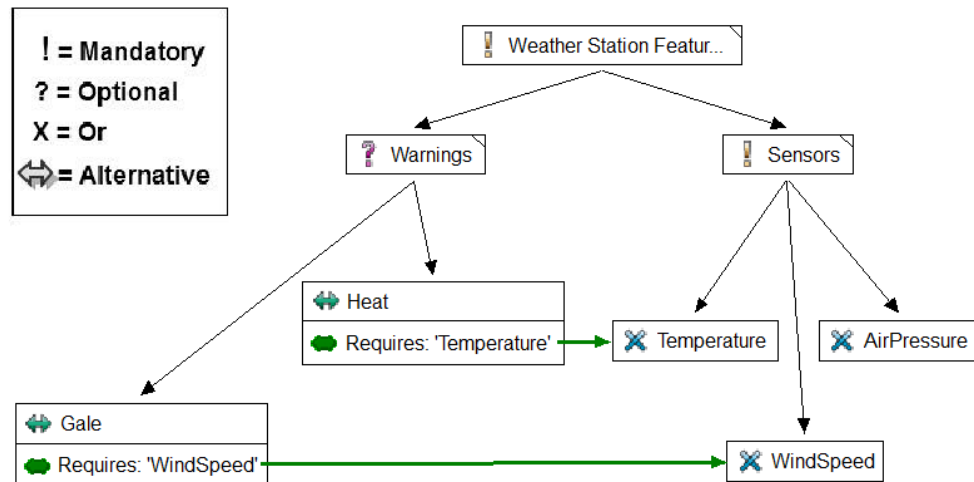


Figure 3-2 Weather Sensor Feature Model

- Mandatory: A mandatory feature, as the name suggests is a feature that should always be present in the product variants if its parent feature is included. A child feature has mandatory relationships with its parent. For example Figure 3-2, the feature named ‘Sensors’ is mandatory and it must be a part of all product variants.
- Optional: Optional feature may be included in the variant if its parent feature is added to a product. For example, feature ‘Warnings’ is optional for product instantiation to its sub features: ‘Gale’ and ‘Heat’.
- Alternative: Exactly one feature has to be chosen from many sub features if it is an alternative feature group. For example, among the alternative features ‘Gale’ and ‘Heat’, one of them has to be chosen if ‘Warnings’ is included.
- Or: A set of child features can be selected if they have an Or option when their parent node is selected in each product variant. For example, mandatory feature

in weather sensor has ‘Or’ sub features: ‘Temperature’, ‘WindSpeed’ and ‘AirPressure’. It prescribes that at least one element of the Or group has to be selected from the sub features and it is also possible to select all the three features within the same product.

- Require and Exclude constraints: Feature Model further describes cross-tree dependencies between features. Feature Models always support the mutual relationships, such as requires and mutually exclusive, in order to add more dependencies among the features that are described in the Feature Model [38]. So, in the above example model, a selection of the ‘Gale’ feature is only meaningful in connection with the wind speed measurement capability. This can be modeled by a ‘Gale’ requires ‘Wind Speed’ relationship.

3.4.2 Derivation of Specific Feature Models from the SPL

Each Feature Model for a specific product can be derived from the SPL. At Level I, Figure 3-1 depicts the derivation of Feature Models of products 1 & 2 from the Feature Model of the SPL. At the same level, implementation of the corresponding products has also been done.

3.5 Level II: Reusability of Test Models

Software Product Line offers a great advantage to build families of products with well defined and managed sets of reusable assets in different products. The test models have been derived using tools from the Feature Models and they are maintained at the Level II as shown in Figure 3-1. These test models are important, since they are the back bone for deriving test cases and scenarios for each variant. Object Models and state charts diagrams are used as test models to specify the structural and behavioral effects of the chosen feature combination in the SPL under consideration. Mapping of Object Model diagrams and state diagrams was also performed at the same level between SPL and product variants. So, reusability of the SPL test model is achieved by relating the Feature Models and test models of the product variants.

3.5.1 Object Model Diagram

The Object Model diagram has been chosen to create the test models for SPL and it variants. A code based creation of SPL and deriving its product variants is always a tedious task, a Model-based diagram is helpful to visualize their functional components. An Object Model diagram helps to show the partial and complete structural view of a SPL with its specific time and it clearly states different instances of the classifiers. Object diagrams are the same as class diagrams that they use notations throughout the model. But, the main difference between both diagrams is that the Object Model depicts the specific instances of the classifiers and it connects those instances whereas class diagrams make use of the actual classifiers. Object diagrams are created to identify the facts about specific model elements and their links. The main purpose is to capture the static view of a system at a particular moment [39]. Object Model diagrams can be created by instantiating the classifiers in class, use-case diagrams and components.

For better understanding, here is a concrete example for Object Model diagram shown below (Figure 3-3: weather station example). Object diagrams show instances instead of classes. They are useful for explaining small pieces with complicated relationships, especially recursive relationships. Each rectangle in the object diagram corresponds to a single instance. Instance names are underlined in UML diagrams. Class or instance names may be omitted from object diagrams as long as the diagram meaning is clear.

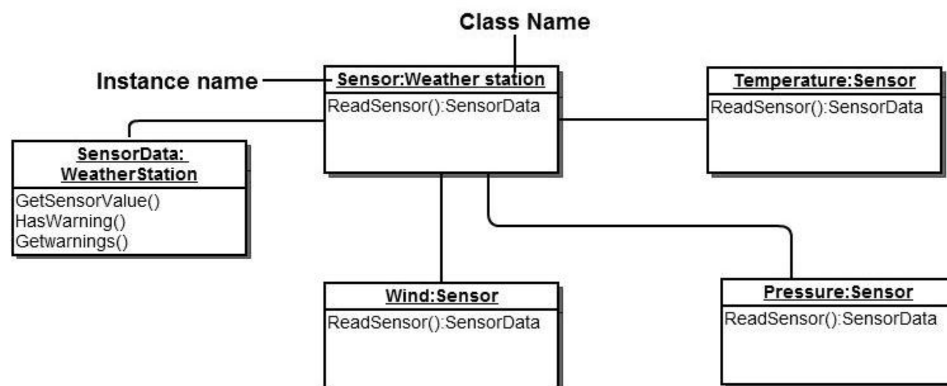


Figure 3-3 Object diagram for Weather Sensor

3.5.2 State Chart diagram

There are various merits in using State Chart diagrams in deriving test models for each component from Object Model within the SPL. This model represents the behavioral view of components through its graph of states and transitions. It also shows the response of an object to external stimuli for each class or a method [40] [41]. Each state satisfies a condition and also performs an action. They wait for each event actions and they cannot be interrupted but, they are concurrently nested to other states. The common model elements that state chart diagrams contain are: States, Start and end states, Transitions, Entry, Do, and Exit actions

State diagrams are represented in a rounded rectangle with at least one section of names which is mandatory. A list of internal actions can also be used with optional guards but, they are not mandatory. The start and end states are the special actions but they cannot have any arguments. A state transition is a relationship between two states that indicates when an object can move the focus of control on to another state once certain conditions are met. In a state chart diagram, a transition to self-element is similar to a state transition [41]. However, it does not move the focus of control. An attached state diagram for each participating object defines its state. Each combination of initial state configurations defines at least one test case in the set. Figure 3-4 describes state chart model for the weather sensor.

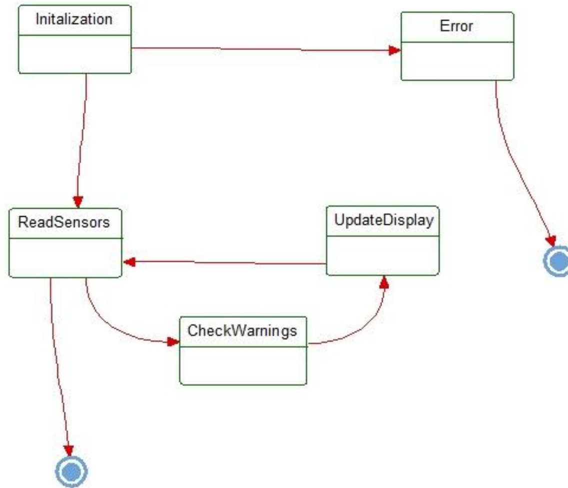


Figure 3-4 State Chart Model for Weather Sensor

3.6 Level III: Test case Derivation

Level III mainly focuses on generating Unit Test cases and test case scenarios from Object Model diagrams and from state diagrams respectively. Each test case is represented as a sequence diagram and describes how the main components of the system interact to fulfill the goal of each feature. A pragmatic approach is to concentrate on typical message sequences as modeled using the sequence diagram. Each sequence diagram specifies single test case or set of test cases. But normally, modeled sequences are incomplete and offer no information about time in the program's life cycle, when the modeled behavior will occur nor state information about participating objects. A test based on sequence diagrams must consider the aforesaid issues.

3.6.1 Sequence Diagram

During the testing process, sequence diagrams are used to capture the actual system trace and the possible interactions in a system. At the implementation part, they verify that all conditions are met between specification requirements. Also, sequence diagrams can be used to specify expected behavior (given a set of preconditions and an ordered set of stimuli) and it represents expected output. In this research, sequence diagrams are used to represent all test scenarios derived from different model diagrams [42] [43]. The test cases that are generated during the test consist of sequences of parameterized actions and events. The sequence diagrams attached to the features allow bridging a part of the gap

between the specification and the test cases, since they describe the expected exchanges of messages between the requirements and the system under test.

Each object in a sequence diagram is represented in a lifeline that describes all the points of interaction with other objects and its corresponding events [43]. The top of a sequence diagram is a lifeline that descends vertically to represent the passage of time. The actions and event interactions between objects and lifelines are represented in horizontal lines with an arrow head.

To explain other alternate actions and structures, boxes are used around set of arrows. The below Figure 3-5 describes the example sequence diagram for weather sensor.

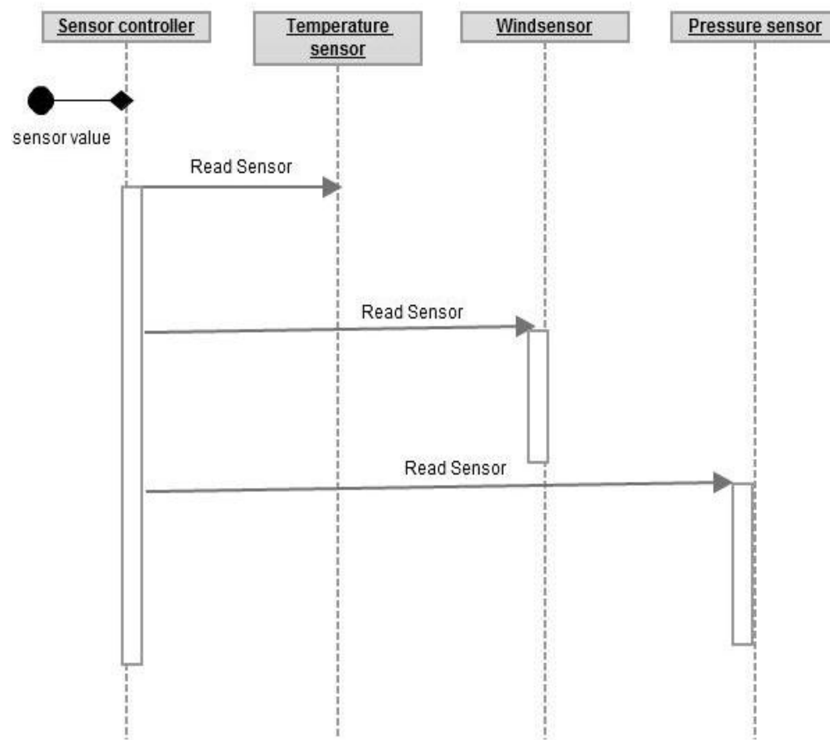


Figure 3-5 Sequence Diagram for Weather Sensor

Unit test cases and system test scenarios have been clustered for both SPL and its product variants [8] [9]. This allows one to easily recognize, locate and compare a specific product variant or the entire SPL from the repository. This clustering does not affect the individual functionality of the test cases since each of these test cases (either unit test case or system test case) can be individually verified and compared for a product variant

with its SPL. For example, any component change in Product 1 can be compared against the SPL using the unit test cases of Product 1.

3.7 Test Case Repository

An environment is required to maintain a large history of test cases. A software configuration tool is used for this purpose of storage and a comparison among test cases is also done here. All the test cases generated through unit and system testing are stored in this repository. The derived test cases that should be optimized are maintained as a set of sequence diagram.

3.8 Comparator

A regression test selection method is used to extract all the redundant, re-testable and reusable test cases by comparing the test cases using a comparator. The comparator finds the element-level differences, attribute-level differences, diagram differences and code-level differences. Both unit and system test cases that directly traverse the changed methods and their subsequent calling methods are selected. In this same level the test cases are sent through a comparator for detecting system level changes in the sequence diagram. Both identical and changed test cases will be reported. All the test cases in the repository are subject to comparison with the SPL test suite. The results are generated as a report and different classes can also be visually seen for each test case.

3.9 Level IV: Test Case Run and Execution

The obtained different classes of test cases such as Reusable, Re-testable or Obsolete are classified to their product variants to get corresponding test suites. These reduced sets of test cases are used for test runs and execution of the products is done.

CHAPTER 4: OPERATIONS

Each Phase in the Figure performs a set of operations. These operations have been discussed in this section. To perform these operations, various tools that are readily available in the market have been used. A detailed description of the operations and the tools used has been presented below.

4.1 PHASE A

The first phase begins with the Feature Model diagrams. Feature Model diagrams provide a means of describing the structure of the SPL based on the features that they perform. There exist many tools to provide a Feature Model diagram of the product line in question. However, pure::variants has been used in this research work.

4.1.1 Pure::variants

Pure variants provide a set of integrated tools to assist in the development of each step of a SPL. Pure::variants is the tool to outline and manage efficiently all parts of software products with their components, restrictions and terms of usage. With this set of information and with the continuous tool support throughout the entire software configuration process valid solutions are created automatically from the features [45]. Its open framework design provides the facility of integrating with other tools and types of data such as requirements management systems, object-oriented modeling tools, configuration management systems, bug tracking systems, code generators, compilers, UML descriptions, documentation, source code, etc. [45]. It facilitates in the creation of an infrastructure for a SPL by providing Feature Models. This serves as a representation of the problem domain. In addition to this, pure::variants also provides a representation for the solution domain through Family Models. To provide integral relationship between the models in a SPL, pure::variants offers the following four roles [45]:

Domain analyst: To build and maintain the Feature Model with commonalities and variations

Domain designer: To design a Family Model and connect it to the Feature Model

Application analyst: To explore the problem domain and provide features and additional configuration information for the problems

Application developer: To generate a solution family member using the transformation engine.

This research work only utilizes the Feature Model generation facility of pure::variants. Once the Feature Model for the given product line has been created using pure::variants, the next step essentially involves generating Object Models and State diagrams for the respective SPL. Pure::variants does not by itself generate Object Models and state models.

4.1.2 Rational Rhapsody

However, both Object Model diagram and State chart diagram can be generated using IBM Rational Rhapsody, a modeling environment based on UML. Rational Rhapsody can be used to create either embedded or real-time systems. It provides a method of implementing the solution from design diagrams. It also provides the ability to analyze and track the intended behavior of the application even during the early stages of a product's development cycle through UML and SysML diagrams [44]. In Rational Rhapsody, testing can be performed as the application is being created. One need not wait till the end of the development phase to test a product. Object diagrams, which are a structural overview of the building blocks of a system, and the State diagrams, which provide a visual representation of the control flow through different states of an object, can be produced using Rational Rhapsody. To do this, IBM provides a Rational Rhapsody plugin that can be used in pure::variants [44] [45].

To generate the respective Object Model and the State model for the given Feature Model, one needs to map each and every component of the Feature Model to a corresponding component in Rational Rhapsody environment. This procedure needs to be carried out manually to ensure proper mapping of the components to their respective objects and to visualize the control flow within various states of these objects.

4.1.3 Procedure for Phase A

The first step of Figure 3-1 essentially involves creation of Feature Models through the specifications and requirements provided by the user. These Feature Models are generated for a SPL using pure::variants. Once the Feature Model has been created, the

Rational Rhapsody plugin for pure::variants was installed. Then each and every component, represented by a feature in pure::variants is mapped to the corresponding component of Rational Rhapsody to provide a link based on which the Object Model and State model can be generated. Upon successful mapping, Object Models and State Models are created for the concerned SPL. Each Object Model contains at least one object. These objects in turn contain State Models. Once the three models (Feature Model, Object Model and State Model) have been created for the SPL, pure::variants optimizes and links these models to create corresponding models for any variations of the SPL that may result in an individual product. This method of linking Feature Models to Object Models and consecutively to State Models allows one to create these structures for n products based on the SPL, without having to individually link each and every component of the product variations [44] [46]. This methods inherits the reusability property of models and hence the term 'Reusable test models' in the Figure.

4.2 PHASE B

Object Models and State Chart diagrams that have been created and maintained at Phase A will be used for generation of test cases and test scenarios at phase B. As already discussed, these test models that are reusable between SPL and product variants, turn out to be more helpful for reuse of test cases. IBM Rational Rhapsody plays a major role in the creation of test case scenarios for both Unit and System Testing [46]. Though, it cannot perform independently in generation and automation of test cases, a test conductor add-on can be used for this purpose.

4.2.1 Test Conductor

Test Conductor is a Model-based testing environment used to debug and test object-oriented systems in Rhapsody. It mainly supports two main features: Automatic Test Architecture Generation and Automatic Test Case Execution. The Automatic Test Generation Add-on (ATG) supports the feature Automatic Test Case Generation, which is an optional one that has been integrated with test conductor. Figure 4-1 illustrates a layout for the features of test conductor.

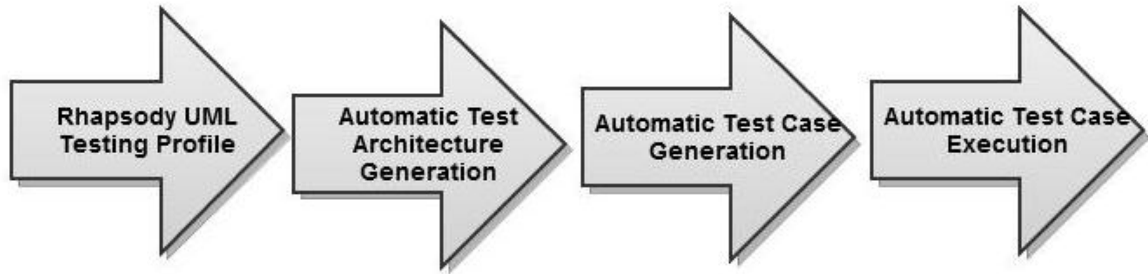


Figure 4-2 Test Conductor Features [46]

In this testing environment, test cases can be created in various forms such as sequence diagrams, state charts, flow charts and pure code-oriented test cases, but in this case, test cases are represented as sequence diagrams. The Rhapsody environment gives an advantage to test a system against its requirement. The merits of using sequence diagrams as test cases lie in the face that they provide graphical definitions, parameterized sequence diagrams and an advanced graphical failure analysis [46].

The main advantage in using this add-on is that it provides sequence diagrams for unit test cases and system test cases, wherein unit testing is based on graphical test definitions. Though, Rhapsody supports several languages such as C, C++, Java and Ada, this research makes use of C++. Also, it is easy to define and execute extensive test suites, as well as to create complex test drivers and test monitors using Rhapsody.

4.2.2 Procedure for Phase B

From the Object Model diagrams, test architectures are created using test conductor add-on for each class/component. Each component has a test architecture that can be used to derive unit test cases. Test cases can be either manually or automatically created. ATG helps to generate test cases from the test architecture that has already been created [47]. A test architecture assists in the creation of a new test package; new test context including System Under Test (SUT) and the test component. It also provides a link between SUT and test components. For better accuracy, partial manual creation of test cases from user requirements and specifications has also been done. Each test case has been represented as a sequence diagram. Similarly, system test cases were created using both manual and automatic test case generation. These test scenarios are simultaneously verified with their specification requirement to complete the model coverage. Finally,

both unit test cases and system test case scenarios have been clustered into a group for a SPL and its product variants. These grouped test cases are then sent to the repository.

4.3 PHASE C

The obtained test cases need to be maintained, stored and accessed anytime. This repository provides the ability to access and retrieve the required test case, be it for the entire SPL or for a specific product variant. Then, a comparator performs regression testing on this stored test suite to extract reusable, re-testable, and obsolete test cases. IBM Rational Clear Case, a software configuration management tool is used for workspace management and parallel development support for testing. Rational Clear Case is integrated with the IBM Diffmerge tool to perform the comparison between test cases.

4.3.1 IBM Clear Case

Software Configuration Management (SCM) is the software engineering discipline that deals with managing changes to software. IBM Rational Clear Case is a software configuration management system, which helps to maintain multiple variants of evolving software systems [48]. It keeps track of different versions of software that were built from individual programs or from multiple variant programs according to user defined specifications. This tool helps to manage test cases among SPL and product variants. It ensures that SPL test cases do not conflict with those of the product variants' through simultaneous updates. This also provides Safety, Stability, Control and Traceability among test cases derivations. IBM Rational Rhapsody and IBM Rational Clear Case were integrated to store all the test case that has been generated through software models [48]. This enables collaborative editing, maintaining and sharing test cases (simple and complex) of the SPL and its variants. The test cases are kept in a secure repository which offers features such as history, traceability, roll-back, metadata and much more.

IBM Diffmerge is already integrated with Clear case and has been used as a comparator to classify all the test cases that are stored in the repository [48] [49]. DiffMerge performs graphical comparison between sequence diagrams for each test case and also consecutive walk-through of all the differences in the test cases. It also aids the testing team in test

case collaboration by showing how a design has changed between model test cases and also how these diverse test cases can be merged. It also generates report on logical differences and graphical differences among each test case and a set of test cases.

Logical differences are model differences and source code differences. Graphical differences are differences that are identified by graphical comparison between sequence diagrams that do not affect the logical aspects of the test cases; for example, there might be a difference in a line color or font, or a changed position of an action that is supposed to be performed. IBM Diffmerge is a sub-component of Clear Case and hence can be accessed from within the Software Configuration Management tool, Clear Case [48].

4.3.2 Procedure for Phase C

The test cases have been derived both manually and automatically using Test Conductor and ATG add-on in Rational Rhapsody [46] [47]. Set of test cases that are derived from each components are considered as a unit and they are managed using Clear Case. A Rhapsody Unit is a file system representation of modeling elements such as projects, Object Model diagrams and State Chart diagrams and they can be saved as a separate file. These units can be generated for an entire SPL or for a product variant and they are stored in an archive. Many test cases are generated for each component and this process can be derivations of test cases from any of the models discussed in previous sections.

In this research work, these test cases represent derivations from sequence diagrams. Comparisons are performed for the entire unit which may represent either the tests cases for the entire SPL or a particular product variant or even a single unit/component of the product. To compare two units, which represent product variations, these units are chosen from the archive and the third unit would be the corresponding unit of the SPL. This is done on a Diffmerge window, which has been activated from within the Clear Case tool. The third unit of the SPL is optional, wherein two product variants can be compared against each other.

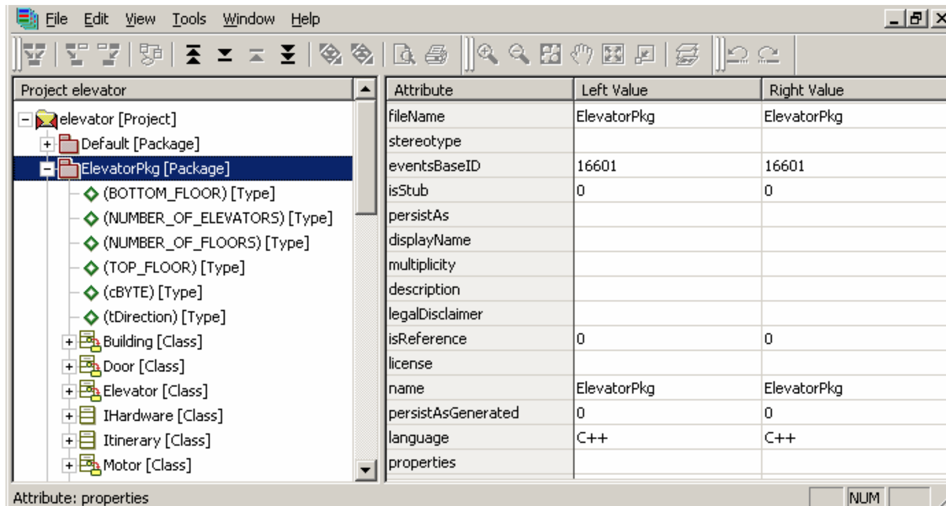


Figure 4-3 Comparison window from Diffmerge

Similarly, a product variant's unit can also be compared against the unit of a SPL. Once this is done, Diffmerge visualises the differences that exist in these units. A left against right kind of comparison is provided by Diffmerge [49]. For each and every test case of product variant 1, the differences between subsequent test cases of product variant 2 can be identified by report generated through each test case. The differences in each test cases have been report generated as follows:

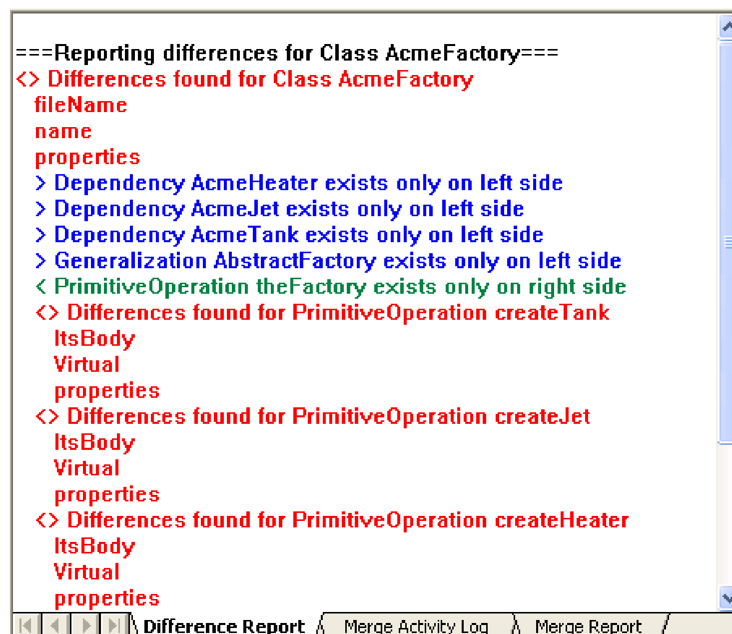


Figure 4-4 Diffmerge Report differences

The Output window uses the following colors to distinguish between the differences categories [48] [49]: red denotes differences between each element, blue represents the elements that exist only on the left hand side of comparison and gray depicts each nested difference. A nested difference is an element without any differences, but that contains an element with either a difference, left-side only, or right-side only element.

4.4 Phase D

Finally, the reusable, re-testable and obsolete test cases are identified through difference reports generated using Diffmerge. The obtained classified test cases will reduce the testing effort of a SPL and its product variants. These test suites for each product variant were used for test runs to execute their corresponding products through code based execution.

CHAPTER 5: IMPLEMENTATION

The feasibility of proposed framework has been demonstrated by implementing the framework on Vending Machine SPL as an example. Since acquiring and validating an industrial SPL is a laborious task, Vending Machine (in this case, a drink dispenser) has been assumed for better understanding. This section clearly defines each and every operation taking place at separate levels from specification till test cases are derived and classified.

The procedure in this implementation process can be described as constructing Feature Models from the specification requirements of a Vending Machine SPL. The test models such as the Object Model diagram and the State Chart diagram for the Vending Machine will be derived from a Feature Model. Corresponding test cases will be derived from those test models and will be stored in a repository for further classification of test cases into Obsolete, Reusable, and Re-testable.

The Vending Machine SPL has been depicted as three different product variants: Pro, Ultra and Simple. General functionality of the Vending Machine can be explained as follows: when a user inserts a coin and selects a drink from choice panel then the machine delivers water, tea or soft drinks. The Vending Machine has several other components and objects such as a Changer, a Choice Panel, a Drink Dispenser, and a Coin Validator. Each component is interrelated and triggers other operations when an event is performed.

The product variant Pro contains all the components that are derived from SPL. Pro can deliver water, soft drinks or tea as customer selects from the choice panel. The Coin Validator can accept either a $1 coin or a 50 cent coin. It also has other components such as a Drink Dispenser and a Changer. Product variant Ultimate can only deliver soft drinks or water from the choice panel. Similar to the Pro variant, Ultimate's coin validator can accept either $1 or 50 cent coins. The other components also are inherited from the SPL.

5.1 Modeling Vending Machine SPL using Feature Model diagram

The pure variants tool has been used to construct a Feature Model Diagram for Vending Machine SPL from its requirement specification. This Feature Model describes the feature commonalities and variabilities between SPL and product variants. Also, it pictures dependencies and constraints between features. This sample model has all three features: Changer, Choice Panel and Coin Validator. Drink Dispenser has been assumed to be an optional feature. Choice Panel has three alternative features: Water, Tea, Soft drinks. Also, Coin Validator has two events: ev100 and ev50 (100 and 50 cent coins). Similarly, Changer is a mandatory feature that has two OR events: ev100 and ev50. Figure5-2 is the Feature Model diagram for Vending Machine SPL.

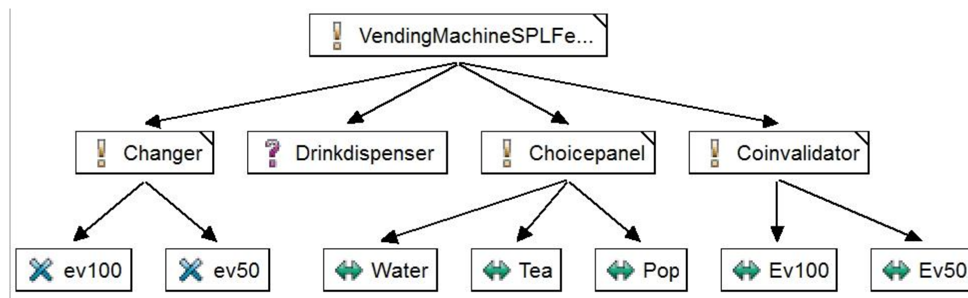


Figure 5-1 Feature Model Diagram for Vending Machine SPL.

From the Sample Feature Model of Vending Machine SPL, the product variants Pro, Ultra, and Simple has been derived. The whole Feature Models have been maintained in the pure variants tool and corresponding product variants are selected from the SPL. Figure5-3 is the snapshot from pure variants. It shows a certain instance in the operation of the software or the state of the system at one time. Once the Feature Model has been created for the SPL and its product variants, the Rational Rhapsody plugin that is installed in pure variants can be used to derive corresponding Object Model diagrams and state chart diagrams for the SPL and its product variants.

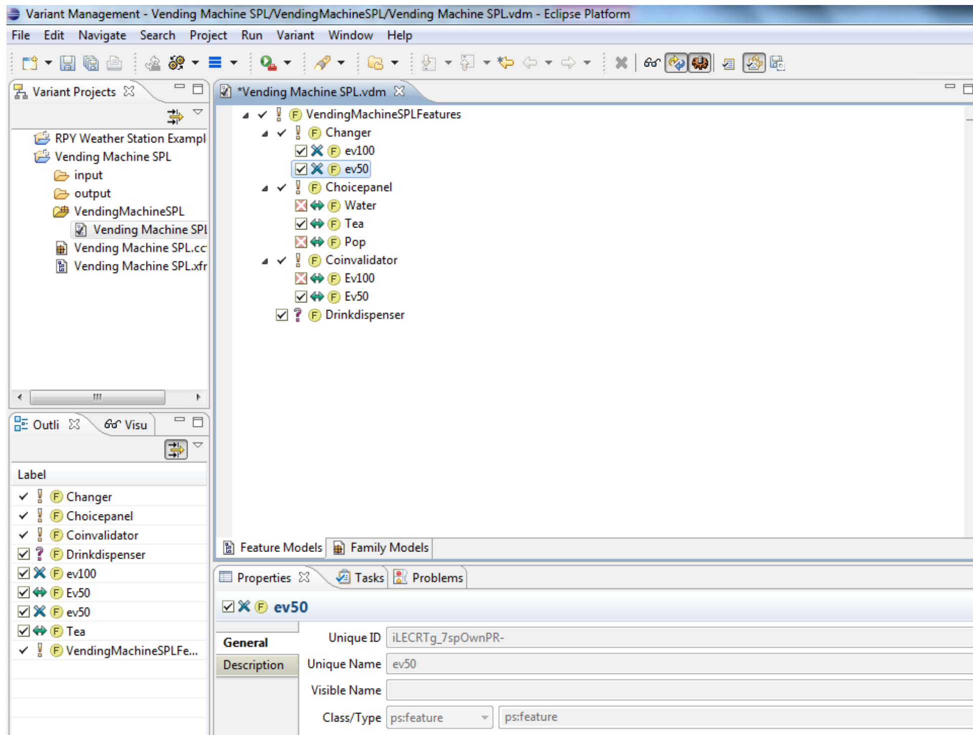


Figure 5-2 Deriving Product Variants

5.2 Generating Object Model for the Feature Model of Vending Machine

The Object Model diagrams are generated from Feature Model diagrams of the SPL. Rational Rhapsody is integrated with the pure variant tool plugin, the corresponding product Object Models are generated once product variants have been derived from the Feature Model in Pure Variant. Figure 5-4 describes the Object Model diagram showing the classes, objects, interfaces, and attributes in Vending Machine SPL and the static relationships that exist between them. Similarly, Object Model Diagram for different product variants can also be derived. These Object Models are considered to be test models since they are the backbone for deriving unit test cases for each component in the SPL.

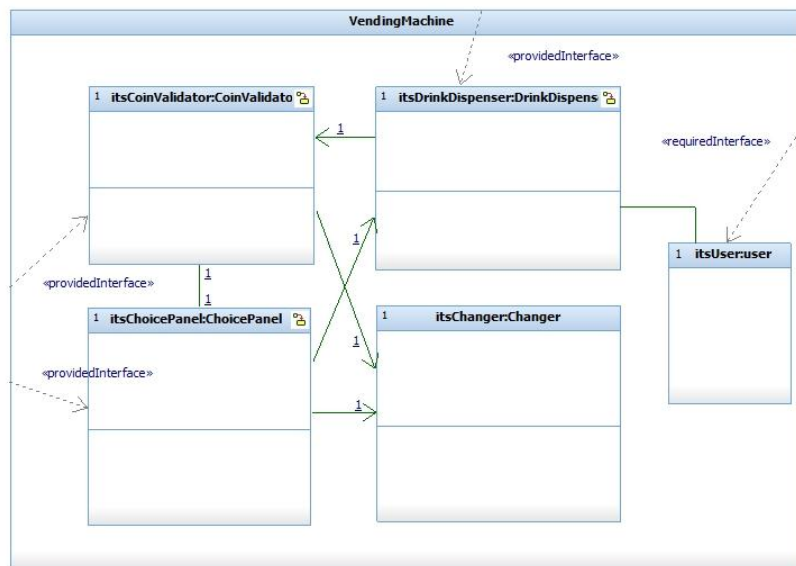


Figure 5-3 Object Model Diagram for Vending Machine SPL

Object Model Diagram for Vending Machine SPL has four different components: Coin Validator, Drink Dispenser, Choice panel and Changer, including one user class. Each component is interrelated and it holds its respective state diagrams for object. Also, these components are mapped to features of the Feature Model diagram for the SPL and to its corresponding product variants. Figure 5-4 has components: Coin Validator, Drink Dispenser, Choice panel and Changer which are already mapped with their features in Feature Model diagram maintained in pure variants tool. Whenever there is a selection

change in Feature Model diagram, it is reflected in these test models of Vending Machine SPL.

5.3 Generating State Chart Diagram for components of Object Model Diagram

State chart diagrams are constructed for each component from Object Model diagram of the Vending Machine. The behavior of components Coin Validator, Drink Dispenser, Choice panel and Changer was captured through its state and transition properties of the state diagram.

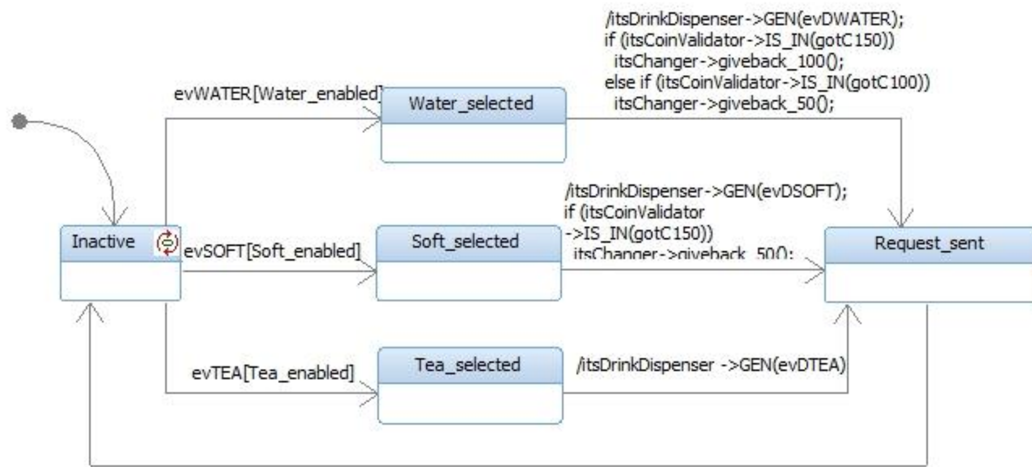


Figure 5-4 State Diagram for Choice panel of Vending Machine SPL

Figure 5-5 describes state chart diagram of the choice panel component. States represented by Water_selected, Soft_selected and Tea_selected in the diagram are objects that remain to satisfy a condition and also perform an action of selection from choice panel. These states are nested to other states. For example, from state of inactive, the choice panel gets active when user selects any one product from the Vending Machine. Similarly, each component from the Object Model diagram carries a state chart diagram with certain conditions that needs to be performed. Figure 5-6 is the state chart diagram of a component 'Choice panel' for the product variant 'Simple Vending Machine'. By comparing this component with state chart diagram for Choice panel of Vending Machine SPL, it can be observed that two states are missing in this diagram. These state chart

diagrams serve as the basis for deriving system test cases for the whole SPL and its corresponding product variants.

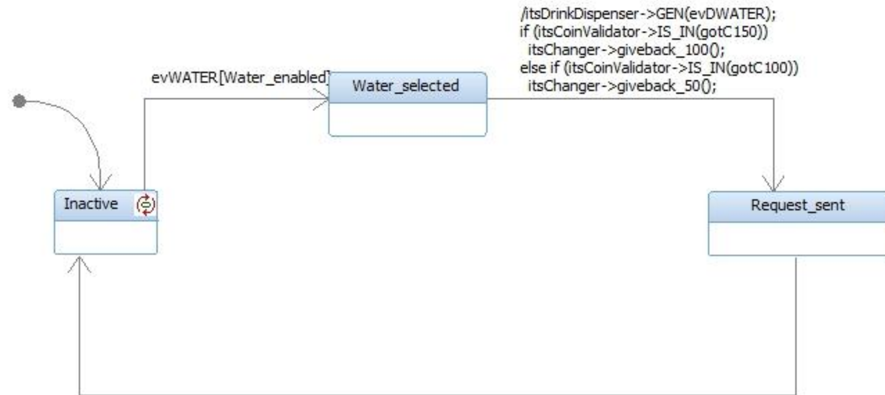


Figure 5-5 State chart diagram for Choice panel of 'Simple' Vending Machine

5.4 Unit testing for each components of Object Model Diagram

The obtained Object and State Chart test models of Vending Machine are the backbone for generation of test cases for the system under test. These test models are reusable between SPL of Vending Machine and its product variants Pro, Ultra, and Simple. IBM Rational Rhapsody maintains complete test models and creates test scenarios for unit components and for the whole system.

The components from Object Model Diagram of Vending Machine are used to create test architecture and thereby derive automated unit test cases from this architecture. The test architecture and test models have been maintained using Test conductor add-on for both Vending Machine SPL and for its product variants. Figure 5-7 describes the automated test architecture for Choice panel of Vending Machine SPL. This test architecture shows how the choice panel is interacting with other components. The test component Choice panel of Vending Machine is nested to other test components such as Coin Validator, Drink Dispenser, and Changer. These test architectures represent a single unit before respective test cases can be created.

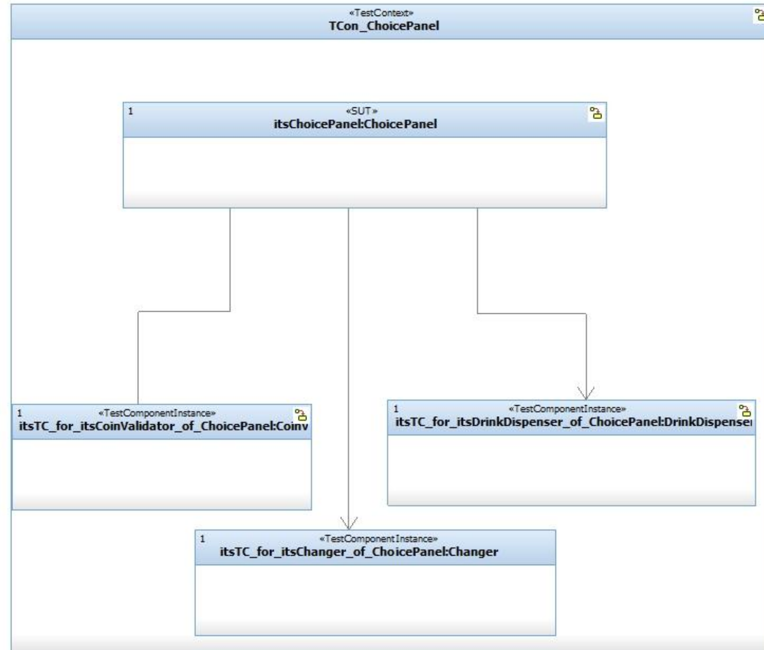
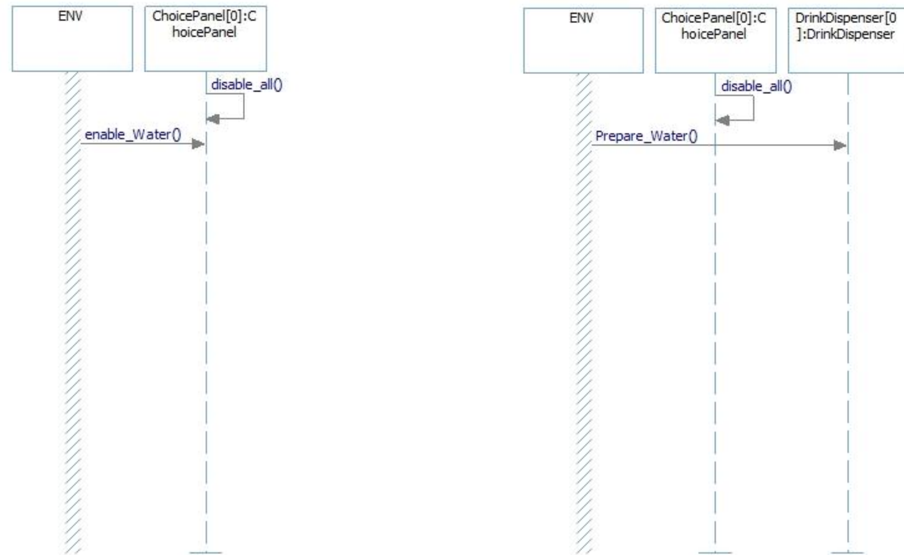


Figure 5-6 Test architecture of Vending Machine SPL 'Choice Panel' Component

From each unit component of test architecture, the test cases are generated manually and automatically using ATG (Automatic Test Case Generator). User requirements and specifications have been verified for better accuracy and also they are validated with the model coverage.

The Vending Machine SPL choice panel component has 24 test cases which covers all the possible chances of Unit Testing. Similarly, the test cases have also been derived for each component of choice panel product variants. There are 13 redundant test cases since the same choice panel component has been reused among different product variants. Figure 5-8 depicts the test scenarios for choice panel components of both Vending Machine SPL [Figure 5-8 (b)] and Simple Vending Machine [Figure 5-8 (a)]. The events enable_Water() and Prepare_Water() in Figure 5-8, represent the selection of water from the choice panel.



a. Simple Vending Machine Choice panel Panel **b. Vending Machine SPL Choice**

Figure 5-7 Test cases of selecting water from Choice panel Component

5.5 Generation of System test cases by ATG

Test cases have been automatically generated using ATG (Automatic Test Case Generator) for the whole Vending Machine SPL and also for specific products. State Diagrams have been constructed for each component in the Object Model diagram of the Vending Machine SPL. ATG generates its test cases from the architecture of state chart diagrams of the SPL and its corresponding product variants. These test scenarios were simultaneously verified with the specification requirements through traceability matrix using tools. Once these test cases have been generated, they are stored in a repository which can be accessed for retrieval of test cases anytime. For example, when one considers the water Vending Machine, the unit test cases of each component of the water Vending Machine has been clustered based on relationship and stored in the repository. When one needs to access the test case of the 'Changer' component of water Vending Machine, this can be easily accessed from the repository.

5.6 Comparator

The set of stored test cases in repository have been subjected to test case selection process of regression testing through comparator to classify the test cases into Obsolete, Re-testable, and Reusable. Unit test cases that are derived from each component of

Object Model diagram of Vending Machine SPL has been compared with the corresponding test cases of component of its product variants. Similarly, system test cases have also been subjected for comparison and test cases were derived through system testing.

Figure 5-9 is the snapshot of IBM Clear Case integrated with IBM Diffmerge tool that describes the comparison of system test cases between Vending Machine SPL and Water Vending Machine (a variant). This snapshot depicts the comparison of each attribute between the test cases of two product variants. Every test case represented in sequence diagram has subjected to regression testing during implementation at each level. Each test case of Simple Vending Machine has been compared with the test case of Vending Machine SPL and their corresponding test classifier reports have also been generated.

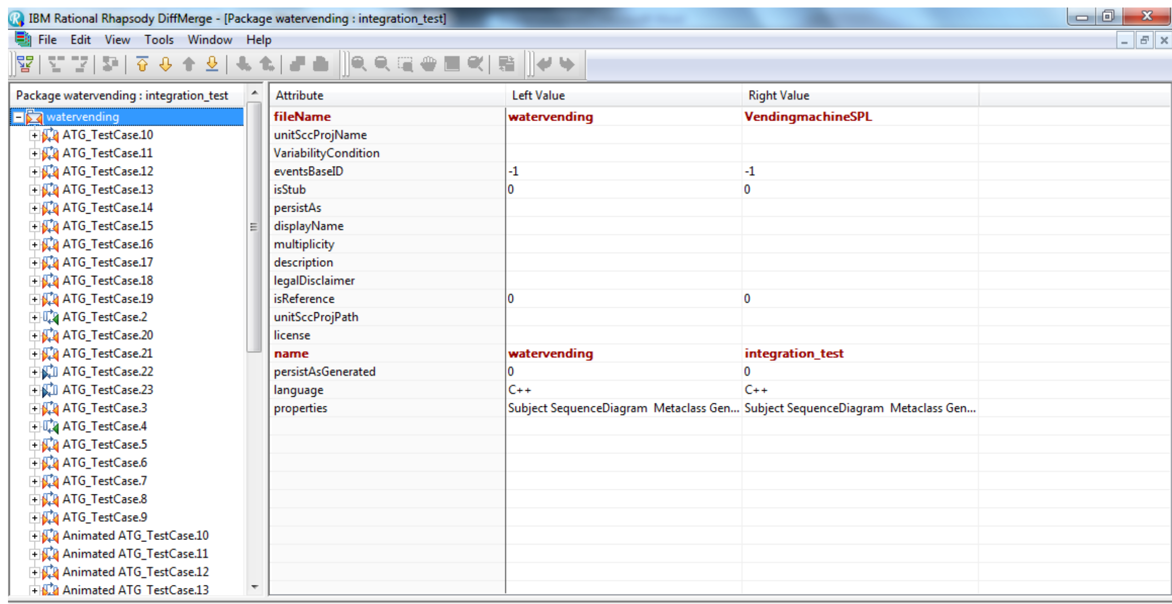


Figure 5-8 Snapshot of Comparator

5.6.1 Graphical Comparison

The attributes are also compared graphically between each test cases of Vending Machine SPL and Simple Vending Machine. This depiction provides a visual representation of the messages, transitions and signals of the sequence diagram respectively. As it can be seen from Figure 5-10, these three features have been distinctively color coded for easy

identification of test cases. This enables one to identify the test cases as redundant, reusable or obsolete through regression testing.

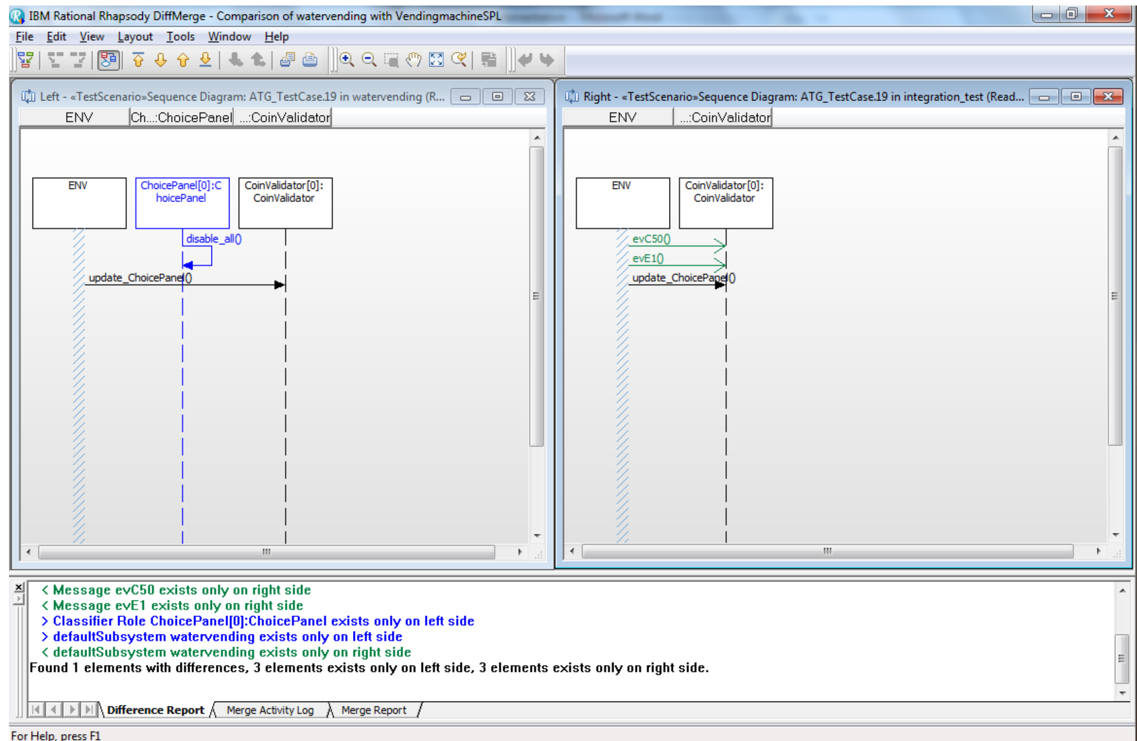


Figure 5-9: Graphical Comparison between SPL and a product variant

5.7 Output Report of Test Classifiers

IBM Clear Case uses the comparison tool Diffmerge to find differences in the attributes of test cases of Vending Machine SPL. It uses the test suite stored in repository for selecting the test cases for regression testing. Test Cases are classified using the 'change of attributes' information in the sequence diagram generated from state chart and Object Model diagram. It classifies the baseline test suite into Obsolete, reusable and re-testable test cases.

The comparator uses change of attributes information between test cases of Vending Machine SPL and Simple Vending Machine instances and classifies the test cases by using the set of added, deleted and modified transitions. The set of added, deleted and modified transitions are compared with each test transition in a test sequence to find the obsolete, reusable and re-testable sequences. The graphical differences among test cases have been simultaneously generated and the test cases are classified accordingly.

5.7.1 Obsolete Test Cases

The Figure 5-11 depicts one of the test cases of choice panel component of Simple Vending Machine (Water Vending Machine). This test case is considered to be obsolete since it contains an invalid execution sequence of messages on boundary objects. The choice panel component has been reused and also this product variant is expected to have only water selection. The test case checks for Soft Drink selection on a Water Vending Machine. An invalid sequence of message results from a change in the possible sequences according to which the comparator generates the report of differences between Vending Machine SPL and Simple Vending Machine.

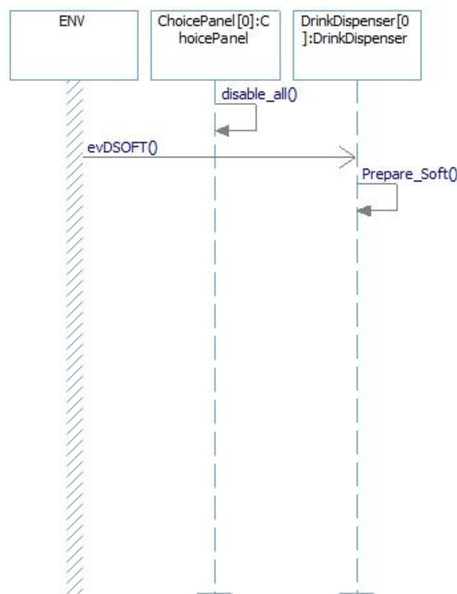


Figure 5-10 Obsolete test case of Water Vending Machine

These Obsolete test cases are identified and removed from the test suite of those corresponding product variants. This reduces the testing effort and time during the execution of these test cases.

5.7.2 Reusable Test Cases

Figure 5-12 depicts one of the reusable test cases of the Simple Vending Machine. A reusable test case consists of a valid sequence of messages to boundary objects that has remained unchanged in the newly derived product variant. The choice panel component of Vending Machine SPL has been compared with the Simple Vending Machine and has

been found that no differences exist between both these test cases. Since, these test cases derived from the component of Object Model diagram has been used in all the product variants there is no change in the sequences of internal messages triggered by the boundary messages. In other words, the test scenarios and the messages involved in the test cases of both the Vending Machines have not changed and these test cases can be used in all the derived product variants.

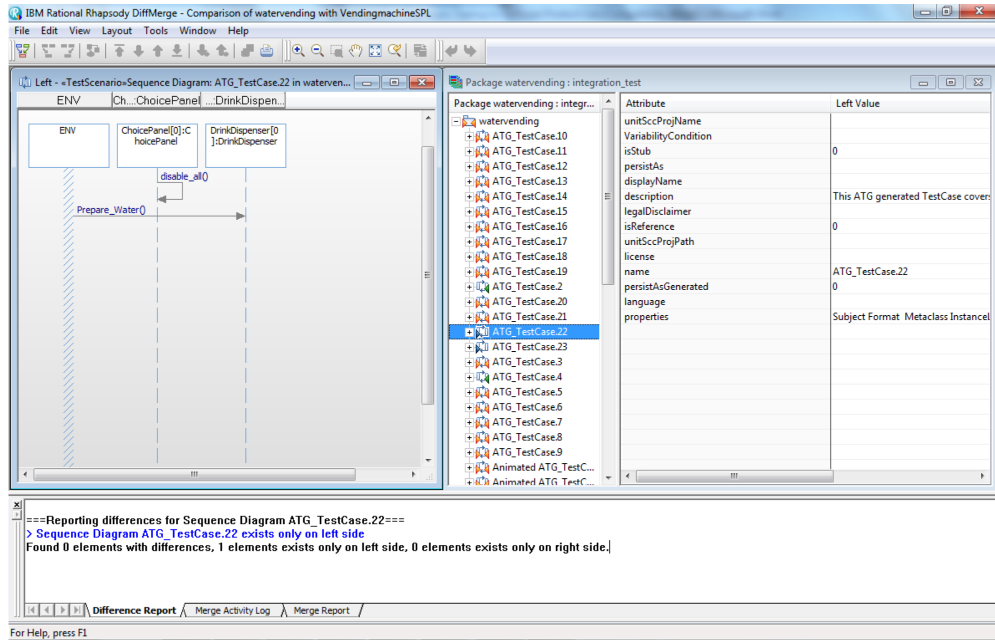


Figure 5-11 Reusable test case of Water Vending Machine

These reusable test cases are identified and they are reused in all the corresponding product variant test suite of those using the same component. This also helps in reduction of number of test cases from the Simple Vending Machine Test Suites and also reduces the testing effort and time during the execution of these test cases.

5.7.3 Re-testable Test Cases

Figure 5-13 depicts one of the re-testable test cases of the Simple Vending Machine. It describes that a sequence of re-testable test cases remained unchanged in the derived product variant. But one or more of these messages may have changed and also messages triggered by boundary class messages have also been changed. These changes of messages indicate that the test case is re-testable. During the comparison between 'Pro' and 'Simple' Vending Machines the re-testable test case are identified from the report

generated. The re-testable test cases are more important and they are always carefully executed in the end.

Also, they are identified and retested in the entire corresponding product variant test suite. This also helps in considerable reduction of the number of test cases from the Simple Vending Machine Test Suites. Once can observe a reduced testing effort and testing time during the execution of these test cases.

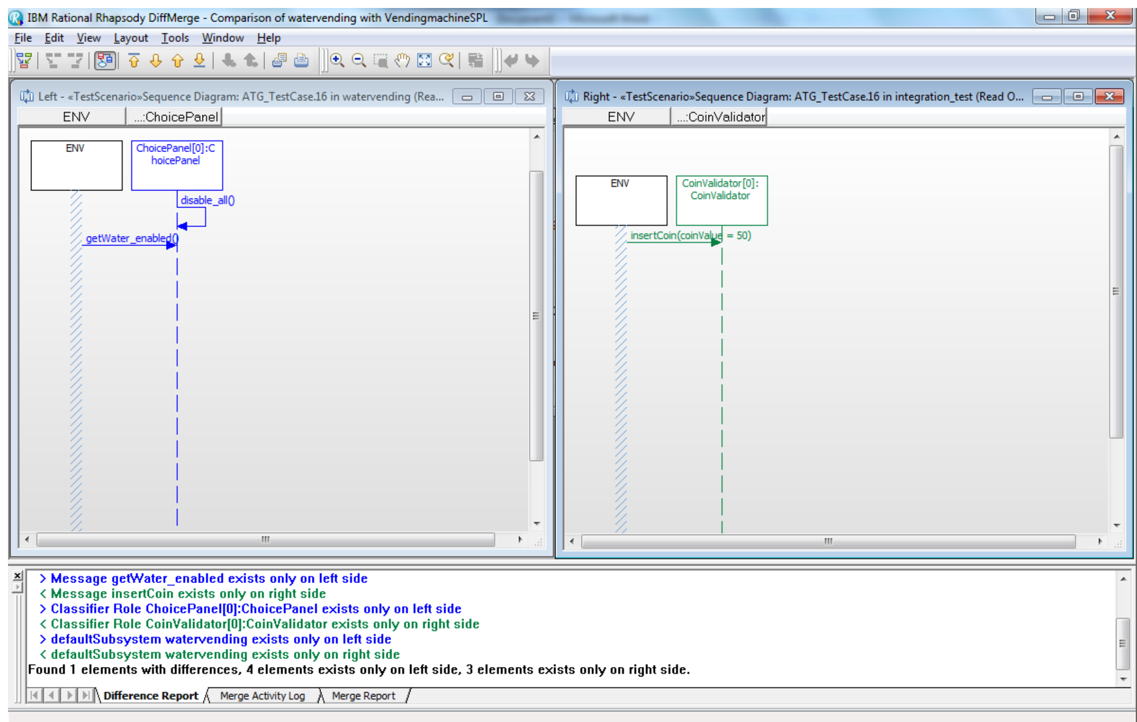


Figure 5-12 Re-testable Test case of Vending Machine SPL and Simple Vending Machine

Thus the test cases that have been classified by regression test selection strategy into Obsolete, Reusable and Re-testable confirms the successful performance of the proposed framework. The redundancy that existed in SPL testing has been identified through this approach and test models have also been reused. Previous research studies indicate that classification test cases will reduce the testing effort considerably [25, 26, 27].

CHAPTER 6: EVALUATION

6.1 Difficulty in specifications separation

The essence of the proposed framework is to separate the specifications into those common to all members of the family and those unique to individual variants, then to use Model-based Testing to generate appropriate tests for each specification set. If this separation is done with all specifications for all product variants, it is impossible to generate obsolete test cases. Accumulating all the specifications and composing relevant test cases is practically infeasible for medium-sized or large SPLs [25]. It would also be interesting to know when one could classify a test case as being obsolete. The whole concept of a test being obsolete only occurs in the context of specifications that are changing. If we consider an instance in time, the specifications are not changing and therefore tests could be created. This work has assumed such a SPL at one instance in time to generate obsolete test cases.

Another important question that needs to be addressed is how hard is it really to divide the specifications into those that exist in common to all product variants and those which are unique to a particular product perhaps because they are in conflict with a specification that would apply to a different product. In a real-time setting, this division of specifications is harder than what we have hinted at in this thesis work. The specifications may evolve over time and hence could be shared among the product variants. A Model-based Testing approach for the set of such time-dependent structure of shared specifications though may be aware of the test cases generated for each product variant, may generate obsolete test cases or premature test cases.

6.2 Reduced test cases

Model-based Testing has been used in this thesis work that has the potential of generating a very large number of tests. It was therefore essential to find out how good a reduction of tests could be expected from this process. Previous researches [22] [24] suggested that Model-based Testing has been effective in testing embedded systems and complex data systems. The Vending Machine SPL involves embedded system components and Model-based Testing generated only a limited number of test cases. The total number of test cases generated for the Vending Machine SPL was 78. The proposed framework helped

in separating the test cases and there were 24 obsolete test cases, 5 re-testable test cases and 49 reusable test cases in the pool of 78 test cases. This scheme resulted in 30% reduction of test cases for the assumed Vending Machine SPL. It would be interesting to find out what percentage of test case reduction may be expected in a real-time system.

There remain three other questions pertaining to Model-based Testing. Firstly, how effective is Model-based testing at actually finding bugs in programs? There exist so many bugs that have to do with actions that are not part of the model. Interface bugs and Performance bugs are two examples that are not found in specifications of the software and hence they do not appear in the model. Finding these bugs using Model-based Testing might be ineffective.

Secondly, does Model-based Testing consider the effort required to setup for tests? To perform a test, we may have to setup for the test, perform the test and then analyze what we observed. For example, while testing a database, the biggest amount of time consumed in testing a database is actually for setting up the database so that we could run the tests on it. The order of testing is very important because, if we setup the databases for a particular test, then we may have to setup the database differently in order to do the next test. This consumes a huge amount of time. In Model-based Testing, there is no suggestion of what order the tests should be done in.

Thirdly, while using Model-based testing, how hard is it to find an actual bug in the sense of what line of code needs to be changed to correct the bug when we start from a Model-based test? Model-based Testing only describes certain actions of the abstract model. In a piece of code which is 6000 lines, relating the 6000 lines to find where in the 6000 lines that particular abstract operation is actually performed might be hard.

It may also be worthwhile to investigate how hard it is going to be in implementing the proposed framework in some other kind of real-time system.

CHAPTER 7: DISCUSSION

The implementation of proposed framework has been performed on Vending Machine SPL scenario using various tools. But, some aspects need to be considered to avoid the limitations in the demonstration. Since obtaining a real industry oriented Software Product Lines is an impossible task, Vending Machine has been used and these derived test cases are restricted to only certain level of requirement specification. Also, the complete experiment was done based on Model-based testing and so source code may not be detectable from UML documents, e.g., a change in a method's body may not be visible from class, sequence or state chart diagrams. These issues can be avoided by parallel reference with the code generated from each test models at architecture level.

In software controlling physical devices (such as Vending Machine), which is defined as embedded systems, test cases are often chosen based on thinking about "What can go wrong here?". The more interesting tests are for failures not anticipated in the system specifications, such as the changing dispenser jamming because a foreign coin was inserted, or an overturned cup blocking the delivery of liquid as anticipated.

In a Software Product Line, these exceptional situations are most commonly different for different variants, not in the common core assets, because they are related to what is different among the variants. Unit testing against specifications is unlikely to find such things - exception handling often is outside the carefully controlled code flow that tools like Rhapsody are designed in terms of. In such cases exception handling should be specifically implemented. One might find lack of such exception handlers in this thesis work as the main goal was to reduce the test effort by classifying test cases. There is also provision to include these exception handlers in the future.

Also, specific code modifications in certain modules or class may not be reflected in the Object Model diagram or State chart diagram. Similarly, if a method is changed during implementation conditions may not be visible and its message or attribute used also may not change. In certain conditions, code has to be traversed to locate the modification and hence identifying those test conditions may be hectic, as a result UML designs will not be safe. One simple solution would be to ask the people changing the UML diagrams to

indicate if they expect such changes in operation implementations and make a note of it, in a predefined way.

This demonstration intends to illustrate that the proposed framework was adequate in reducing the test cases for the assumed simple Vending Machine SPL. It remains to be seen how this framework would function in a time dependent system. There were few other works done by other researchers in the same field of SPL testing. Noticeably, there were two significant contributions. Andreas Reuys et al [3] developed ScenTED (**Scen**ario based **TE**st case **D**erivation), a Model-based technique for system testing in product family engineering. This technique involved creation of reusable test case scenarios in domain engineering and reusing these test cases in application engineering. The method utilized use case diagrams for specification mapping and activity diagrams as test models to represent possible scenarios of use cases. Test cases were represented using sequence diagrams. This method was implemented in an industrial setting and reuse benefit was estimated at 57% compared to application of single system testing techniques. However, this method failed to address the issue of reducing testing effort or removing redundancy in testing SPLs. This technique also lacked integrated tool support.

Paulo Anselmo da Mota Silveira Neto et al [26] proposed a technique based on regression testing approach. This method utilized architecture specifications to reduce the testing effort. In addition to reusing test cases and execution results, the method intended to select and prioritize an effective set of test cases. These researchers faced a similar problem of not being able to experiment their method in the real SPL context. Synonymous to this thesis work, their method proved to be effective during its application in the experimental study.

Our demonstration helps in classifying test cases into Obsolete, Re-testable and Reusable to reduce testing redundancy for the assumed SPL. However, the lack of access to a real industrial SPL deemed it difficult to conduct a deeper evaluation in the form of an industrial case study. An alternative approach would be to map commonalities among product variants to derive a super-variant that represents a subset of variants. These super-variants may then be tested. This process would provide the means to reduce variants rather than reduce test cases. Separation of test cases into those that may be

reused, those that may need to be retested and those that may be obsolete turned out to be insightful in this demonstration. Further investigation is required to find out the possibility of duplicating the proposed framework in other systems.

CHAPTER 8: CONCLUSION

Software Product Line plays a huge impact in major software industry for creating large number of product variants from a common platform [15]. Creating a product from scratch is a time consuming process. So, analyzing the commonalities and variabilities between different products and their variants helps to reuse the core assets. Many researches have been conducted on SPL and large companies have been still investing to study the uses of Software Product Line in their software development cycle [14, 15, 35].

Software Product Line Testing is a laborious task since large number of product variants can be derived from a product line. Due to the enormous number of possible products, individual product testing becomes more and more infeasible. A framework has been proposed and it is supported by various testing tools to extract the different test cases through various testing processes from the UML based design models. The main objective is to classify test cases stored in a large repository derived from the Software Product Line. In the context of Model-based development, regression test selection method has been performed on different design models and their changes have been monitored at each level.

This demonstration of framework can be carried out in different systems that can fit in both industrial and academic settings. However, the proposed framework has been implemented in various tools and applied to a Vending Machine scenario for better understanding of entire procedure. The whole procedure has been broken down through top-down approach by constructing a Feature Model diagram from specification requirements. The test models such as Object Model diagram and State Chart diagram derived from Feature Models. By performing Unit testing and System testing on these test models, test cases have been obtained which are in turn stored in the repository. The extracted results through regression test selection method clearly indicate the classification of test cases into Obsolete, Reusable, and Re-testable. This confirms that the objective that was proposed in Section 1.2 has been successfully met. This approach of classifying test cases through regression testing has been proven to reduce the testing effort to a greater extent through a wide range of research studies. This work also explains that certain changes in operations may not be visible in the UML models and

additional code traversal or better way of documentation is required. Proposing a framework and identifying the corresponding testing tools that can perform regression test selection based on UML designs helps to reduce the testing effort in a SPL. When design changes have been determined, it gives good support to plan and adapt regression testing effort in the Software Product Line.

In the future, it is important to run additional case studies to assess the drawbacks and advantages of proposed framework by implementing in different systems. To get better results, combinatorial testing [25] can be performed while deriving the Feature Model diagram from specification requirements. By installing the Mosopolite plugin, the pure variant tool will generate a subset of configurations covering pairwise feature interaction with flattening algorithm for Feature Model diagram. This in turn reduces the number of product variants that have to be tested in the product line. Despite of not having been experimented in the real industry SPL context, it has been only tested in Vending Machine SPL. For future work, new experiments will be executed considering real components, modules and SPL architectures, in industrial projects. In addition, guidelines to help the test classification step and tool support to aid the approach execution are also identified as future work. Furthermore, the lessons learned in this experiment will provide important information to execute future evaluations.

REFERENCES

1. Isis Cabral, “Designing Software Product Lines for Testability”, A Thesis, Presented to the Faculty of the Graduate College at the University of Nebraska, 2010, http://digitalcommons.unl.edu/cgi/viewcontent.cgi?article=&context=computerscidis, [Accessed: 12 June. 2011].
2. Sungwon Kang, Jihyun Lee, Myungchul Kim and Woojin Lee “Towards a Formal Framework for Product Line Test Development”, Seventh International Conference on Computer and Information Technology, 2007. pp: 921 – 926.
3. Andreas Reuys, Erik Kamsties, Klaus Pohl, and Sacha Reis, “Model-based System Testing of Software Product Families”, O. Pastor and J. Falcão e Cunha (Eds.): CAiSE 2005, LNCS 3520, pp. 519 -534, Springer-Verlag Berlin Heidelberg, 2005.
4. Northorp, L., “A Framework for Software Product Line Practice Version 5.0”, Software Engineering Institute, Carnegie Mellon University,2001, http://www.sei.cmu.edu/productlines/frame_report/config.man.htm, [Accessed: 12 July. 2011].
5. K. Pohl, G. Böckle, and F. van der Linden. Software Product Line Engineering: Foundations, Principles, and Techniques, Springer, Heidelberg, August (2005).
6. Meyer, M. and Lehnerd, A “The Power of Product Platforms”. New York, New York: The Free Press (1997).
7. Tevanlinna, A.and J. Taina,, "Product Family Testing: A Survey." SIGSOFT Software Engineering Notes 29(2): 1-6, 2004.
8. McGregor, John, "Testing a Software Product Line" (2001), Software Engineering Institute. Paper 630. http://repository.cmu.edu/sei/630 [Accessed: 14 July. 2011].
9. McGregor, J. D and P. Sodhani, “Testing Variability in a Software Product Line“, Software Product Line Testing Workshop (SPLiT). Boston, MA, Avaya labs: 45-50, 2004.
10. Kolb, R., “A Risk-Driven Approach for Efficiently Testing Software Product Lines”, 2nd Int'l Conference on Generative Programming and Component Engineering. Erfurt, Germany, 2003. pp: 409-414.
11. Offutt, J.; Abdurazik, A.; “Generating Tests from UML Specifications”, 2nd Intl. Conference on UML’99, 1999.

12. Binder, R.; “Testing Object-Oriented Systems: Models, Patterns, and Tools”, Addison- Wesley, Reading, 2000.
13. Ibrahim K. El-Far; “Enjoying the Perks of Model-based Testing”, Software Testing, Analysis,and Review Conference, STARWEST 2001, pp. 9-15.
14. Andreas Reuys, Erik Kamsties, Klaus Pohl and Sacha Reis, “Model-based System Testing of Software Product Families”, O. Pastor and J. Falcão e Cunha (Eds.): CAiSE 2005, LNCS 3520, pp. 519 – 534, 2005.
15. McGregor, John D. “A Testing Perspective.” Journal of Software Testing Professionals 1, 4 (December 2000): 11-15.
16. J.Hartmann, M. Vieira, A. Ruder, “UML-based Approach for Validating Product Lines” , Intl. Workshop on Software Product Line Testing (SPLiT), Avaya Labs Technical Report, pp. 58-64, August (2004).
17. E.Kamsties , Klaus Pohl , Sacha Reis , Andreas Reuys ,“Testing variabilities in use case models”, Proceedings of the 5th International Workshop on Software Product-Family Engineering, PFE-5 (Siena, Italy, Nov. 2003), Springer, Heidelberg, 6--18. (2003).
18. H.Muccini and A. van der Hoek, “Towards Testing Product Line Architectures”, Electronic Notes in Theoretical Computer Science 82 No. 6, (2003).
19. C.Condron., “A Domain Approach to Test Automation of Product Lines”, International Workshop on Software Product Line Testing. (2004), pp. 27-35.
20. C.Nebut Pickin, S, Le Traon, Y, Jezequel, “Automated requirements-based generation of test cases for product families”, In Proceedings 18th IEEE International Conference on Automated Software Engineering (2003), pp.263.
21. C. Nebut, Pickin, S, Le Traon, Y, Jezequel, “Reusable Test Requirements for UML-Model Product Lines”, Inte rnational Workshop on Requirements Engineering for Product Lines (REPL) (2002), pp. 233-239.
22. Olimpiew, E. and Gomaa, H “Model-based testing for applications derived from Software Product Lines” Proceedings of the 2005 workshop on Advances in Model-Based Testing, pages 1–7 (2005).

23. S. Kang, J. Lee, “Towards a Formal Framework for Product Line Test Development”, In Proceedings of the 7th IEEE International Conference on Computer and Information Technology (October 16 - 19, 2007). CIT. IEEE Computer Society, Washington, DC, 921-926. (2007)
24. J.C.Dueñas, “Model driven testing in product family context”, First European Workshop on Model Driven Architecture with Emphasis on Industrial Application. (2004), pp. 91-96.
25. Emelie Engström, “Exploring Regression Testing and Software Product Line Testing - Research and State of Practice”, Lund University Faculty of Engineering, Department of Computer Science , Licentiate Thesis, 2010. pp. 120.
26. Paulo Anselmo da Mota Silveira Neto, Ivan do Carmo Machado, Yguarat˜a Cerqueira Cavalcanti, Eduardo Santana de Almeida, Vinicius Cardoso Garcia, and Silvio Romero de Lemos Meira, “A Regression Testing Approach for Software Product Lines Architectures”, Reuse in Software Engineering, (RiSE). VDM Publishing House, July 2010, Pages: 168.
27. Mary Jean Harrold, James A. Jones, Tongyu Li and Donglin Liang “Regression test selection for java software,” in OOPSLA ’01: Conference on Object-oriented programming, systems, languages, and applications. New York, USA: ACM, pp. 312–326, 2001.
28. J.Hartmann and D.J. Robson., “Techniques for selective revalidation”, IEEE Software 7, 1, (1990), 31–36.
29. Yih-farn Chen, David S. Rosenblum , and Kiem-phong Vo ,“Test tube: a system for selective regression testing”, In Proceedings of the International Conference on Software Engineering, (Los Alamitos, CA, USA), IEEE, 1994, pp. 211–220.
30. Rajiv Gupta, Mary Jean, Harrold Mary, and Lou Soffa, “Program slicing-based regression testing techniques”, Software Testing, Verification and Reliability 6, 2 (1996), 83–111.
31. Elbaum, A.G. “Test Case Prioritization: A Family of Empirical Studies”, IEEE Transactions on Software Engineering 28, 2 (Feb. 2002), 159-182.
32. Sebastian Oster, “MoSo-PoLiTe - Tool Suppor t for Pairwise and Model-based Software Product Line Testing”, VaMoS ’11, January 27-29, 2011.

33. Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Bjöorn Regnell, Anders Wesslé, “Experimentation in software engineering: an introduction”, Norwell, MA, USA: Kluwer Academic Publishers, 2000. Pp. 33-39.
34. H.K.N. Leung and L. White, “Insights into regression testing,” in ICSM ’89: International Conference on Software Maintenance, pp. 60–69, 1989.
35. Software Engineering Institute, “A Framework for Software Product Line Practice”, Carnegie Mellon, http://www.sei.cmu.edu/productlines/frame_report/req_eng.htm, [Accessed:11 July.2011].
36. Kang, K.C., S.G. Cohen, J.A. Hess, W.E. Novak and A.S. Peterson (1990), “Feature-Oriented Domain Analysis (FODA) Feasibility Study,” Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1990. pp.333.
37. Heymans, P, Schobbens, Trigaux, Bontemps, Y., Matulevicius, & Classen, A, “Evaluating formal properties of feature diagram languages”. Software, IET, 281–302, 2008.
38. Danilo Beuche, "Software Product Line Engineering with Feature Models", pure-systems, [Online] Available: http://www.pure-systems.com/, [Accessed: 13, May.2011].
39. IBM, “Object Diagram Help Contents”, publib.boulder.ibm.com/infocenter/rsmhelp/v7r5m0/index.jsp?topic=%2Fcom.ibm.xtools.modeler.doc%2Ftopics%2Fcobjdiags.html, [Accessed: 14 September, 2011].
40. Supaporn, “Automated-Generating Test Case Using UML Statechart Diagrams”, Proceedings of SAICSIT 2003, Pages 296 – 300
41. D. Seifert, S. Helke, and T. Santen, “Test Case Generation for UML Statecharts”, Perspectives of System Informatics (2003), Springer, Pages: 93–109
42. F. Fraikin and T. Leonhardt “SeDiTeC -Testing Based on Sequence Diagrams”, Proceedings of the 17th IEEE international conference on Automated software engineering (ASE) 2002.
43. Ashalatha Nayak, Debasis Samanta, "Automatic Test Data Synthesis using UML Sequence Diagrams", in Journal of Object Technology, vol. 09, April 2010, pp. 75-104.

44. IBM, "Rational Rhapsody User guide", March 2010, [online], Available: http://publib.boulder.ibm.com/infocenter/rsdp/v1r0m0/topic/com.ibm.help.download.rhapsody.doc/pdf75/UserGuide.pdf, [Accessed: 15 March 2011].
45. Pure Variants, Pure Variants User Manual, May 2010-2011, [online], Available: http://www.pure-systems.com/fileadmin/downloads/pure-variants/doc/pv-user-manual.pdf, [Accessed: 15 April. 2011].
46. IBM, IBM Rational Rhapsody Test Conductor Add On, May [online], Available: http://publib.boulder.ibm.com/infocenter/rhaphlp/v7r5/topic/com.ibm.rhapsody.oem.pdf.doc/pdf/RTC_User_Guide.pdf, [Accessed: 17 July. 2011].
47. IBM, IBM Rational Rhapsody Automatic Test Generation Add On User Guide, May 2011,Available:http://publib.boulder.ibm.com/infocenter/rhaphlp/v7r6/topic/com.ibm.rhp.oem.pdf.doc/pdf/ATG_User_Guide.pdf, [Accessed: 17 August. 2011].
48. IBM, IBM Rational Clear Case Guide to Building Software, Available:https://publib.boulder.ibm.com/infocenter/cchelp/v7r1m0/index.jsp?topic=/com.ibm.rational.clearcase.books.cc_build_windows.doc/cc_build.htm [Accessed: 17 September. 2011].
49. IBM, developing in parallel with Rational Rhapsody Diffmerge, Available:http://publib.boulder.ibm.com/infocenter/rhaphlp/v7r6/index.jsp?topic=%2Fcom.ibm.rhp.diffmerge.doc%2Ftopics%2Frhp_c_col_parallel_dev_with_diffmerge.html [Accessed: 17 September. 2011].