INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

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

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

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Université d'Ottawa · University of Ottawa

PERMISSION DE REPRODUIRE ET DE DISTRIBUER LA THÈSE

PERMISSION TO REPRODUCE AND DISTRIBUTE THE THESIS

NOM DE L'AUTEUR / NAME OF AUTHOR:	CHEN, Yanping
ADRESSE POSTALE / MAILING ADDRESS:	43 HARTIN STREET STITTSVILLE ON K2S1B9
GRADE / DEGREE:	**ANNÉE D'OBTENTION / YEAR GRANTED**
M.C.S.(Computer Science)	2003
TITRE DE LA THÈSE / TITLE OF THESIS: SPECIFICATION - BASED REGRESSION TESTING MEASUREMENT WITH RISK ANALYSIS	

L'auteur permet, par la présente, la consultation et le prêt de cette thèse en conformité avec les règlements établis par le bibliothécaire en chef de l'Université d'Ottawa. L'auteur autorise aussi l'Université d'Ottawa, ses successeurs et cessionnaires, à reproduire cet exemplaire par photographie ou photocopie pour fins de prêt ou de vente au prix coûtant aux bibliothèques ou aux chercheurs qui en feront la demande.

Les droits de publication par tout autre moyen et pour vente au public demeureront la propriété de l'auteur de la thèse sous réserve des règlements de l'Université d'Ottawa en matière de publication de thèses.

The author hereby permits the consultation and the lending of this thesis pursuant to the regulations established by the Chief Librarian of the University of Ottawa. The author also authorizes the University of Ottawa, its successors and assignees, to make reproductions of this copy by photographic means or by photocopying and to lend or sell such reproductions at cost to libraries and to scholars requesting them.

The right to publish the thesis by other means and to sell it to the public is reserved to the author, subject to the regulations of the University of Ottawa governing the publication of theses.

N.B. LE MASCULIN COMPREND ÉGALEMENT LE FÉMININ

Dec. 19, 2002.

DATE

Yanping Chen

(AUTEUR) SIGNATURE (AUTHOR)

Université d'Ottawa • University of Ottawa

Université d'Ottawa · University of Ottawa

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTDOCTORALES

FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

CHEN, Yanping

AUTEUR DE LA THÈSE - AUTHOR OF THESIS

M.C.S.

GRADE - DEGREE

School of Information Technology and Engineering

FACULTÉ, ÉCOLE, DÉPARTEMENT - FACULTY, SCHOOL, DEPARTMENT

TITRE DE LA THÈSE - TITLE OF THE THESIS

Specification-based Regression Testing Measurement with Risk Analysis

Robert L. Probert

DIRECTEUR DE LA THÈSE - THESIS SUPERVISOR

EXAMINATEURS DE LA THÈSE - THESIS EXAMINERS

M. Weiss

H. Ural

J.-M. De Koninck, Ph.D.

LE DOYEN DE LA FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTDOCTORALES

SIGNATURE

DEAN OF THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

Specification-based Regression Testing Measurement with Risk Analysis

by

Yanping Chen

A thesis submitted to the School of Graduate Studies and Research in partial fulfillment of the requirement for the degree of

Master of Computer Science

Ottawa-Carlton Institute for Computer Science

School of Information Technology and Engineering

University of Ottawa

Ottawa, Ontario, Canada

October 2002

ABSTRACT

Regression testing is essential to ensure software quality. A test team applies a regression test suite to ensure that new or modified features do not *regress* (make worse) existing features. Although existing research has addressed many related problems and put forward some solutions, most regression test techniques are code-based. Code-based regression test selection is good for unit testing, but it has a scalability problem. When the size of the object under test grows, it becomes hard to manage all relevant *Risk Exposure* information and to create corresponding traceability matrices for validation and coverage assessment.

We propose a scalable alternative, namely a *specification-based* method for regression test selection. There are three major parts of our work:

First, we propose and justify a new, *specification-based* regression test strategy built on two complementary test coverage goals, namely coverage of modified features, and "affected entities" (Targeted Coverage), and additional, complementary coverage of functional areas of risk (Safety Coverage). The resulting regression test suite will contain two corresponding sets of test cases: *Targeted Tests*, which ensure that important current customer features are still supported adequately in the new release and *Safety Tests*, which are risk-directed, and ensure that potential problem areas are properly handled.

Secondly, we provide systematic methods for selecting these two types of regression test cases. For the *Targeted Tests*, we apply regression analysis to requirement to check throughout consistency of "requirement followed by a blank", and design models. The basic model we use for describing requirements based on customer features or behaviors is the *activity diagram*, which is a notation of the Unified Modelling Language (UML). A process is presented for identifying the test cases affected by changes. For the *Safety Tests*, we use *risk analysis* and present a method of choosing *risk-based* test cases. Our *risk analysis* is based on a practical risk model, and is similar to that used by some

organizations.

Finally, to evaluate our approach, we use data from a real software product to do case studies. A common statistical analysis tool is used to analyze the results and generate statistical reports to help demonstrate the effectiveness and efficiency of our approach.

Acknowledgements

I wish to thank the many individuals who have provided comments and suggestions that helped improve this research. I am especially grateful to my supervisor Dr. R.L. Probert who helped me choose the thesis topic and provided the opportunity of doing research in IBM Canada Ltd.. He provided valuable support and information during this research.

This work has been supported by IBM Canada Ltd, NSERC (Natural Sciences and Engineering Research Council of Canada), and CITO (Communications and Information Technology Ontario). I would like to express my gratitude to Mr. D. Paul Sims for his coordination and support when I worked in IBM ECD (Electronic Commerce Development). I would like also to thank Heather Fry, Haiying Xu, Terry Chu and the IBM ECD staff for their help and valuable information for this research, especially during the case studies.

Finally, I wish to thank my parents, my brother, and especially my friend Yuming for his support and encouragement during the whole period that went into conducting this research and writing the thesis.

TABLE OF CONTENTS

List of Figures

List of Tables

Chapter 1. Introduction

1.1 Introduction and Definitions

Regression testing is well known to be an important software maintenance activity performed with the aim of gaining confidence in modified software [Rothermel+00]. It is an essential part of an effective testing process for ensuring software quality. Moreover, *regression testing* is also an important activity in object-oriented software development [Binder 00].

In this chapter, we briefly address problems in regression testing and motivation of our research. Also, we define terms that are used in this thesis. Finally, we describe the scope and contributions of our research, and give out the organization of this thesis.

1.1.1 Software maintenance and regression testing

Software maintenance is the most costly phase of the software life cycle, and has been estimated to comprise between 50 and 80 percent of the total software cost [Leung+91]. One software regression horror story was the network failure of AT&T on September 17th, 1991, which disrupted long-distance calls and air traffic control communications in the New York metropolitan area. Three airports had to be shut down for five hours. It cost AT&T millions of dollars to recover the network service and its company's reputation. Shortly before that disaster, in January 1990, AT&T's long-distance telephone network also had a nine-hour breakdown caused by a code patch, which was not regression tested.

Most software systems evolve to respond to changed requirements based on changing environments, changing needs, new concepts and new technologies. Software usually grows in the number of functions, components and interfaces. Existing systems may be expanded for uses beyond the scope of their original design. Thus, modification of software is inevitable [Leung+91].

Regression testing is one of the necessary maintenance tasks. It is defined as *the process of validating modified software to provide confidence that the changed parts of the software behave as intended and that the unchanged parts of the software have not been adversely affected by the modification* [Harrold+01]. This adverse impact of a change is often called the "ripple effect", and is known to be a serious cause of software regression (getting worse) as the result of a change [Probert 02]. Fortunately, it is widely accepted that efficient and effective regression testing can reduce the frequency and cost of software maintenance.

1.1.2 Software development process and regression testing

With widespread usage of object-oriented programming techniques, more and more projects follow an *evolutionary* process model, also called an *incremental model* [McGregor+01].

An *increment* is a deliverable that provides some of the functionality required for the system, including models, documentation and code. Under this process model, a system is developed as a sequence of *increments*. For each increment, the development process feeds new and/or revised designs and implementations into the testing process. The testing process feeds identified failures back into the development process. The development process and the testing process form a continual feedback loop as shown in Figure 1-1.

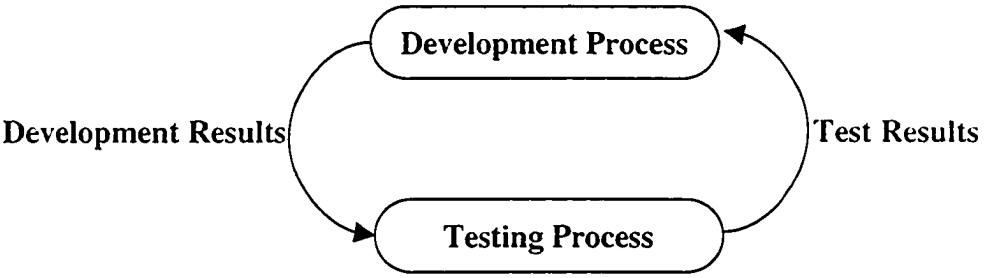


Figure 1-1: The feedback loop of development and testing processes in incremental model

In the *incremental model*, successive *increments* add and sometimes revise system functionality while keeping most of the functionality of previous *increments*. *Regression testing* is typically done between *increments* to ensure that changes do not adversely affect correctly working code of the previous versions.

Reuse has been proposed as a general solution to chronic software development problems: namely, lengthy development time and high cost, unacceptably frequent failures, low maintainability, and low adaptability [Binder 00]. A major precondition of reuse is to ensure that the reused components match the new requirement and do not conflict with new components. An example of disasters caused by incorrect reusing of components was the explosion of Ariane 5 on June 4, 1996. The rocket was on its first voyage, after a decade of development costing $7 billion. Estimates of the total cost of the destroyed rocket and its cargo vary from a low of $350 million in a European Space Agency press release to a high of $2.5 billion reported in Florida Today Space Online. The main reason of the explosion was that a time sequence based on the requirement of Ariane 4 was reused, but this did not match the requirement of Ariane 5 [SIAM 96]. Effective *regression testing* should have caught this problem before the disastrous launch.

With object-oriented techniques, skillful design and hard work, classes and

components can be produced that may be used in many applications. New projects are built by re-using components from legacy systems or third party. The obvious contribution of regression testing to reusability is to access and assure the quality of re-used components, and to gain confidence in their integration.

1.1.3 Definitions

Most of the following definitions are taken from [Binder 00], except those terms that are otherwise referenced.

A *component* is any software aggregate that has visibility in a development environment, for example, a method, a class, an object, a function, a module, an executable component, a task, a utility subsystem, or an application subsystem. This includes executable software entities supplied with an application programmer interface (API).

A *baseline* is a specification or product that has been formally reviewed and agreed upon, that thereafter serves as the basis for further development, and that can only be changed through formal change control procedures [IEEE 610]. A *baseline version* is a component (or system) that has passed a test suite.

A *delta* is a change to a component within the scope of an increment. The *delta version* of a component (or a system) is a changed version that has not been regression tested or has not passed all critical regression tests. A *delta build* is an executable configuration of the SUT that contains all the delta and baseline components.

Black box testing uses specified or expected responsibilities of a unit, subsystem, or system to design tests without reference to code. This term is synonymous with *specification-based testing*, *behavioral testing*, *functional testing*, or *responsibility-based testing*.

White box testing uses source code analysis to develop test cases. It is synonymous to *implementation-based testing*, *code-based testing*, *structural testing*, *clear box testing* and *glass box testing*.

Grey box testing is a combination of *black box* and *white box testing* to reveal omissions and surprises. Information from requirements, design, and source code are used to develop test cases [Probert 02].

A *regression test case* is a test case that the baseline has passed and which is expected to pass when rerun on a delta version [Binder 00]. A *regression test suite* is composed of regression test cases. In some organizations, the term "regression test bucket" is used instead of *regression test suite.* A *regression fault* is a fault induced by a delta build. It is revealed by a test case that the baseline version has previously passed but the delta version does not pass.

In [Leung+91], Leung and White defined *regression testing* as a testing process, which is applied after a program is modified. It involves testing the modified program with some test cases in order to reestablish our confidence that the program will perform according to the specification. In this thesis, we will use the definition in [IEEE 610], which is: *regression testing is defined as selective retesting of a system, subsystem, or component to verify that modifications have not caused unintended effects and that the system still complies with its specified requirements.*

Without running a modified program on all test cases of test suite T, *regression test selection (RTS)* attempts to select a cost-optimized subset of test cases to determine if the modified program has the same behavior as a previous, acceptable version of the program running on T.

1.2 Statement of the Research Problem and Motivation

1.2.1 Related work

Many researchers are investigating *regression testing* techniques. Their research spans a wide variety of topics. For example, Brown and Hoffman [Brown+90] work on test environments and automation of the regression testing process. Harrold, Gupta, and Soffa [Harrold+93], and Wong et al. [Wong+95] address test suite management. Rothermel and Harrold [Rothermel+96] present a framework to evaluate regression test selection techniques.

In recent years, much attention has been put into the *regression test selection* area. Most techniques are *white-box* (*code-based*), that is, they select tests based on information about the *delta* between original code and the modified version [Kung+95, Binkley 97, Rothermel+00, Harrold+01]. Only a few techniques are *black-box* (*specification-based*) methods, that is, they select tests based on information obtained from program specifications [Leung+90, Mayrhauser+94, Stocks+96, Vieira+00].

Many researchers, especially those from industry, are very interested in the idea of *risk-based* testing. Pfleeger [Pfleeger 00] discusses risk management. Amland [Amland 00] presents fundamentals and metrics of *risk-based* testing. Bach works on methodologies for *risk analysis* [Bach 99]. Some researchers have considered applying *risk analysis* in testing, but only that in an informal way. Their approaches have applicability problems and issues with automation.

1.2.2 Problems with existing approaches

Currently, most *regression test selection* techniques are *code-based. Code-based* regression techniques can be applied effectively to *regression testing* at the unit level. But when we try to test a larger or more complex component, for example, a utility or a subsystem, it becomes very difficult to manage all the information obtained from code and to create corresponding traceability management matrices (for example for coverage statistics). Therefore, *code-based* regression techniques are not suitable for testing larger components at more abstract levels, such as subsystem testing.

Secondly, by *code-based* regression techniques require testers to access and understand the code to some degree [Chen+02]. This requirement causes many practical problems. Testers have to spend time reading the code written by others and trying to understand how it works. This is very time-consuming, costly, and error-prone.

Finally, *code-based* regression techniques are language-dependent. In some software systems, more than one programming language may be used, that is, an Internet application may use Java, JSP and HTML. More than one *code-based* regression technique will then be necessary for *regression testing*. This makes the situation more complex.

Little research has been done on *specification-based* regression testing, since *specification-based* methods are somehow informal. Thus, it is difficult to measure coverage and evaluate the completeness of the selected test suites. But recently in practice, some companies have developed *specification-based* regression test strategies, such as IBM [Chen+02]. The major problem with these *regression testing* methods is that the selection criteria are somewhat subjective, that is, there are too many personal decisions involved. Thus, for a manager, it is very hard to measure whether a regression test suite is good enough to ensure a customer's business will not be at risk. Also, there is no way to automate the selection process.

The main motivating factors behind this research can be summarized as follows:

- The absence of effective and efficient *specification-based* regression testing methods

- The absence of objective *specification-based* regression test selection criteria
- The weakness in test suite selection and evolution methods and tools
- The need for systematic, *risk-based* regression test selection methods.

1.3 Scope of the Thesis and Overview of Contributions

In this research, we propose a *specification-based* regression method as an efficient and effective solution for addressing the problems stated above. The underlying thrusts of our method are:

- Traceability: The user requirements, analysis, design, implementation, and test cases are tightly linked. These links are used to manage regression analysis. We define a traceability matrix to trace and identify the test cases affected by changes.
- *Regression analysis*: We model the desired system behavior and changes using "*grey box*" *activity diagrams*. The *activity diagram* is an UML notation. A detailed description is provided in Chapter 4 (see section 4.2.2). We also introduce a regression algorithm to solve the reselection problem of regression test cases based on a core *activity diagram* and traceability matrix.
- *Risk analysis*: *Risk Exposure* (RE) is introduced to measure how important a test case is with respect to risk coverage. It is used to measure the quality of test suites. We also present a systematic and objective method to select test cases and end-to-end scenarios for *regression testing*.

The contributions of this thesis can be summarized as follows:

- We demonstrate that traceability is essential for conducting and managing regression analysis and testing. We define useful requirement traceability links for regression analysis between *requirement attributes* and test cases.

- We provide a new strategy for selecting regression test cases. Our strategy is totally *specification-based.* Unlike previous *specification-based* strategies, it has less subjective selection criteria, is systematic, and can be implemented in test tools. In addition, it is directed toward coverage of the modification ripple effect from both a customer requirements perspective, and toward coverage of *risk* from both customer and developer perspectives.
- We present techniques for implementing this strategy to yield regression tests consisting of both *Targeted Tests* and *Safety Tests.* These two terms are defined in Chapter 2 (see section 2.5). This provides a clean and useful focus for the definition of *specification-based* regression tests.
- We define and illustrate the use of *risk analysis*, a customer-oriented concept, in *regression testing*, and show how the *Risk Exposure* metric can be used to measure the quality of regression test suites.

1.4 Organization of the Thesis

The remainder of this thesis is organized as follows:

Chapter 2: Describes background information on regression testing, and the basic terminology used throughout this document.

Chapter 3: Introduces *risk analysis* in general, and gives a detailed definition of a common, practical risk model to use.

Chapters 4: Presents a method to build requirement traceability links based on the concept of *activity diagram* model, and a regression analysis and test selection method based on requirement traceability links.

Chapter 5: Presents and gives examples of a complete *risk-based* regression test

selection strategy.

Chapter 6: Presents and analyses the results of three detailed industrial case studies to evaluate our technique.

Chapter 7: Gives conclusions and suggestions for future work.

Chapter 2. Background: Regression Testing Concepts

2.1 Introduction

Myers observed that modifying an existing program was a more error-prone process than writing a new program in terms of errors per statement written [Myers 79]. Jones reported that high-technology companies like IBM had "a bad fix (error) injection rate which ranges from less than 2% to more than 20% with a modern average in the 1990s running to about 7%" [Jones 97]. Note that this observation does not apply directly to any one company, such as IBM, but reflects rates at a number of similar companies.

Regression testing is used to confirm that fixed bugs have been truly fixed, previously that new bugs have not been introduced (ripple effect) in the process, and that features that were proven correctly functional are intact [Nguyen 01]. Regression testing is effective for revealing regression faults caused by delta side effects, delta/baseline incompatibilities, and undesirable feature interactions between a baseline and a delta. It is also effective for revealing bugs caused by bad fixes.

2.2 Regression Analysis and Testing

Regression analysis and testing is a process applied after a change is made to the implementation of a software system [Luiu+93]. Regression testing usually takes place after the tested software system is modified. One major difference between *regression*

testing and *development testing* is that during *regression testing* an established suite of tests will typically be available for reuse. We denote this established suite of tests as the *full test suite*. With *selective retest (regression test)* techniques, we only rerun a reduced regression test, which involves test cases that test the *affected entities* of the modified components or subsystems. These test cases make up our *regression test suite*. If the cost of selecting a reduced subset of tests to run is less than the cost of running the tests that we omit, the *selective retest* technique is more economical than the *retest-all* technique [Leung+91].

2.2.1 Regression testing techniques

In [Rothermel+96], Rothermel and Harrold described regression testing techniques as follows:

Given a program P, a modified version P', and a set T of test cases used previously to test P, regression analysis and testing techniques attempt to make use of a subset of T to gain sufficient confidence in the correctness of P' with respect to behaviors from P retained in P'.

Regression testing techniques basically consists of the following steps:

1. Identify the modifications that were made to P to yield P'.
2. Select T' $\subseteq$ T, a set of tests to execute on P', based on these modifications.
3. Test P' with T', to establish the correctness of P' with respect to T'.
4. If necessary, create T", a set of new functional or structural tests for P'.
5. Test P' with T", to establish the correctness of P' with respect to T", and gather test information for T".
6. Create T"', the regression test suite for P', by combining T' and T".

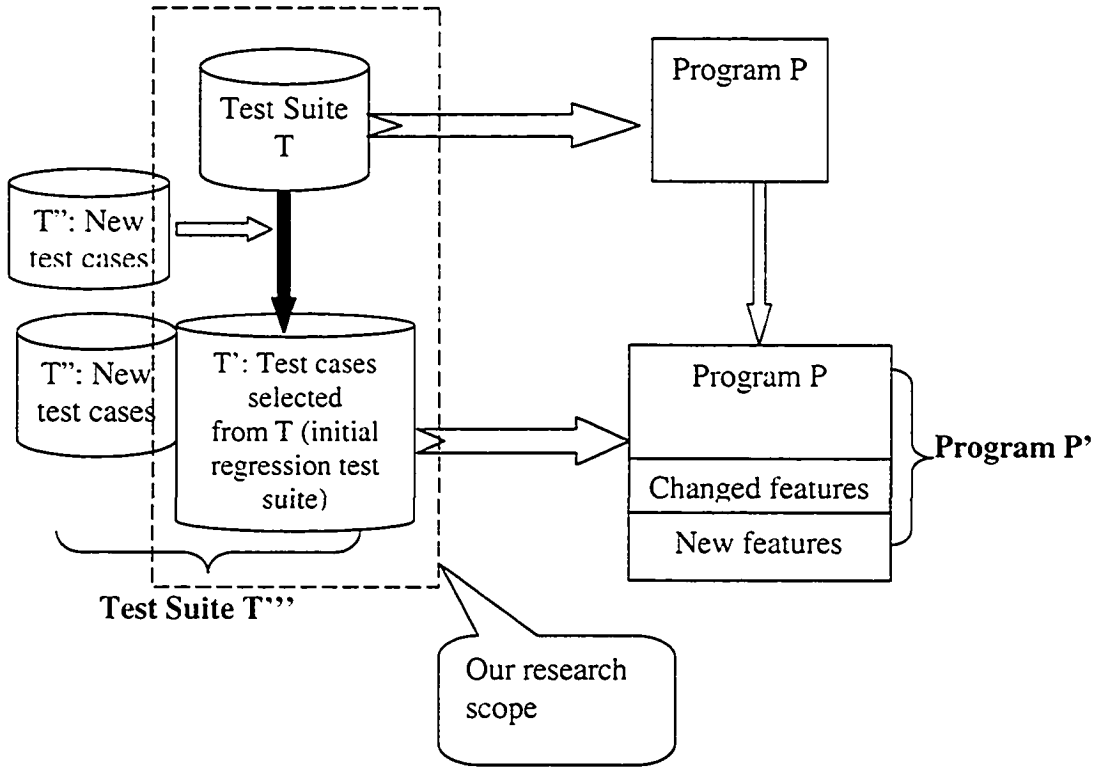


Figure 2-1: Regression test selection

This process of regression testing is illustrated in Figure 2-1. Regression test selection is applied to select T' from T, which is indicated by a solid arrow in Figure 2-1. In this research, we do not focus on creating new test cases for new or changed features

2.2.2 Discussion of regression analysis

Retesting an entire software system that has received few changes is very expensive for large systems. With regression analysis and testing, we retest only the parts of the software system affected by modifications.

Regression analysis and testing has to address the following fundamental problems [Kung+95]:

- How can all components affected by changes in some components be identified?
- What strategy should be used to retest these affected components?
- What are the coverage criteria for retesting these components?
- How should regression test cases be selected by reusing and/or modifying existing test cases?

To address the above issues, the following activities are essential for any regression analysis and testing strategy.

A Pragmatic Regression Testing Strategy:

1. Identify the affected components and select a regression test coverage criteria (discuss later).
2. Select test cases to test the affected components.
3. Execute the modified program on the selected test cases.
4. Gather and evaluate test results, including evaluating the behavior of the modified software system and reporting the coverage of the regression test suite.
5. Revise test plan as required for the next regression analysis and testing session

Activity 2 is concerned with the selection of test cases to rerun. It may involve generating new test cases and selecting appropriate old ones. In our research, we only focus on technical solutions for selecting appropriate old test cases from the *full test suite*. This is a common industrial approach.

In [Luiu+93], Luiu divided test cases of the *full test suite* into two classes:

- Reusable test case: the test cases that tested unmodified parts of the specification and their corresponding unmodified parts of the implementation.

They remain valid after the modification to the specification and implementation, and need not be rerun.

- Affected test cases: the test cases relevant to the modified parts of the specification and the implementation. There are two categories:
  - Re-testable test cases: the test cases that are still valid and should be rerun
  - Obsolete test cases: the test cases that have become irrelevant or out of date due to the changed specification or implementation.

This process of test case "categorization" or "classification" is shown in Figure 2-2. H. Ben Hajla generalized it to Object-Oriented regression test selection in [Hajla 97].

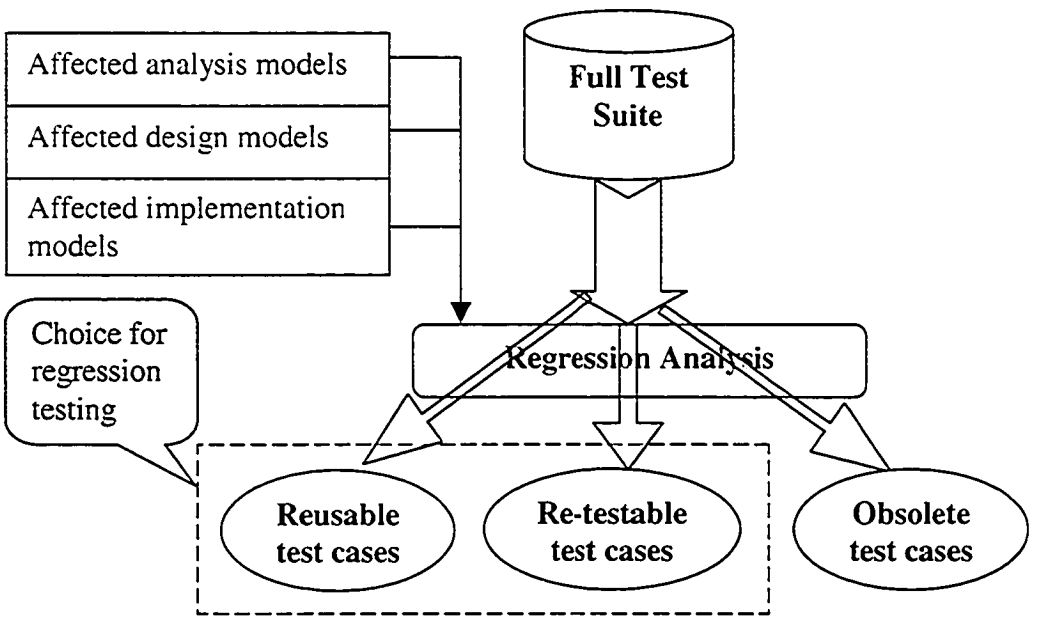


Figure 2-2: Test cases selection

2.3 Regression Test Patterns

Robert Binder abstracted current regression test selection strategies into five patterns [Binder 00]:

- *Retest all*: Rerun all baseline tests.
- *Retest risky use cases*: Choose baseline tests to rerun by applying risk-based heuristics.
- *Retest by profile*: Choose baseline tests to rerun by allocating time in proportion to the operational profile to obtain more customer-representation tests.

  An operational profile is a hierarchic model of system usage used for system testing. Each unique leaf-level usage is assigned a probability and facilitates design of a functional test case [Binder 00].

- *Retest changed segments*: Choose baseline tests to rerun by comparing code changes.
- *Retest within firewall*: Choose baseline tests to rerun by analyzing dependencies.

  Dependencies are the relationships between components. Components typically depend on each other in many ways and dependencies are necessary to implement collaborations and achieve separation of concerns. In regression testing, some components need to be retested because of the dependencies though they are not changed. To set up a firewall is to select a set of components whose test cases will be included in a regression test [Binder 00].

Retest all is the simplest and easiest strategy with the least risk of missing a regression bug. But, if, with respect to time and cost a full regression test run is prohibitive, a reduced regression test suite must be selected. *Retest risky use cases* and

retest by profile are *black box* strategies for selective regression testing because selection is made by analyzing implementation-independent relationships, requirements, and capabilities. *Retest changed segments* and *retest within firewall* are *white box* strategies for partial regression that require analysis of implementation dependencies with respect to the structure of the code.

Each regression test pattern has advantages and disadvantages. One pattern may be better than other in a specific situation, but that is not always the truth. There is widely used framework for evaluating regression test selection strategies introduced by Rothermel and Harrold in [Rothermel+96], which considers four aspects: inclusiveness, precision, efficiency, and generality. We will discuss this framework and apply it to evaluate our approach in Chapter 6.

2.4 Classification of Modifications and Types of Regression Testing

During software development and maintenance, many modifications may occur when the software system is corrected, updated, or fine-tuned. According to the maintenance activities performed on the software system, White classified the modifications into three categories [White 91]:

- *Corrective maintenance*: Involves correcting software failures in order to keep the software working properly. The specification is not likely to be changed and no new modules are likely to be introduced. This type of maintenance is usually referred to as "fixes".

  During the development phase, the specification usually will not be changed based on identified code errors. Therefore, many modifications happening during development are similar to *corrective maintenance* type

modifications.

- *Adaptive maintenance*: Involves adapting the system in response to new user requirements or change in the operational environments.
- *Perfective maintenance*: Involves any enhancement to the system. The objective may be to provide additional functionality, tune system performance, or improve system maintainability by restructuring the software architecture.

In *adaptive maintenance* and *perfective maintenance*, changes occur in user requirements and/or the system specification. New functionality is often introduced. Modifications occurring in the development phase to implement new or modified specifications are similar to these two types of modifications. Leung and White referred both *adaptive maintenance* and *perfective maintenance* as *progressive maintenance* [White 91].

Based on the classification of modifications addressed above, we can classify *regression testing* into two categories:

- *Corrective regression testing*: To test *corrective maintenance*, that is, test a modification in which the specification has not changed.
- *Progressive regression testing*: To test *progressive maintenance*, that is, test a modification to reflect a modified specification.

In this thesis, our discussions about regression testing will be based on this classification.

2.5 Our Strategy: Combine Targeted Tests and Safety Tests

Since the major goal of *regression testing* is to assure system stability, we have to rerun test cases for *requirement attributes* that have been modified or may be affected by

modifications. Through *regression testing*, we want to achieve adequate confidence in the quality of the modified or possibly affected software. As a measure of quality, we would like to have our regression test suite achieve some coverage target as well. In this research, coverage of the regression test suite refers to the percentage of test cases that have been selected as regression test cases out of the *full test suite*.

To differentiate between the two purposes (the confidence purpose and the coverage purpose) and the corresponding methods, we separate regression test cases into two categories:

- *Targeted Tests*: test cases that exercise important affected *requirement attributes*, services, or features and
- *Safety Tests*: test cases selected to reach a pre-defined coverage target.

Intuitively, *Targeted Tests* exercise system behaviors that are key functional features for the customer, while *Safety Tests* try to ensure that potential problem behaviors are handled (that is, properly managed by the system) so as to not cause undesirable consequences for either the customer or the vendor or both. *Targeted Tests* are selected for both corrective regression testing and progressive regression testing, and *Safety Tests* are risk-based test cases, i.e., are independent of type of change. The purpose of *Safety Tests* is to cover defects that missed by *Targeted Tests* because of incomplete or out-of-date documentations. We will discuss this issue more in Chapters 4 and 5.

2.6 Controlled Regression Testing Assumption (CRTA)

In *regression testing* we try to determine if a modified program has the same behavior as a previous version of the program by running all or some of the same tests. To be safe without being too inefficient and too conservative, our approach is based on an assumption called the *Controlled Regression Testing Assumption* (CRTA). In

[Rothermel+96] CRTA is described as follows:

When P' is tested with T, we hold all factors that might influence the output of P', except for the code in P', constant with respect to their states when we tested P with T.

Ideally, regression tests should be run while CRTA holds. When CRTA holds, we can identify whether the behaviors of a system have been changed after modification. This can be quite difficult in a complex, multi-platform test environment. Testers may have to identify and document uncontrollable factors and the test cases that are potentially affected by them in practice.

2.7 Chapter Summary

In this chapter, we provided background information about regression analysis and testing in general and listed common regression test patterns. Also, we categorized modifications, regression testing, and test cases according to the published literature. Then we introduced our own classification of regression test cases used in our regression testing approach, namely *Targeted Tests* and *Safety Tests*. Finally, we restated the CRTA assumption used in our research.

In the next chapter, we will introduce *risk-base testing* and discuss related basic concepts.

Chapter 3. Background: Risk and Risk Analysis

3.1 Introduction

Risk concerns anything that threatens the successful achievement of a project's goals. Specifically, a *risk* concerns an event that has some probability of happening, and that if it occurs, will result in some loss [McGregor+01]. Software systems may contain bugs when delivered. Since the loss for some *risks* is more serious, these *risks* are rated higher than others. The fundamental principle of *risk-based testing* is to do more thorough testing to those parts of the software system that may bring higher *risk* to the project to ensure that the most potentially harmful defects are revealed.

3.2 Risk-based Testing

The tester's mission is to reveal high-priority problems in the product. *Risk* is a measure of concern that a problem might happen. The more likely the problem is to happen, and the more impact it will have if it happens, the higher the *risk* associated with that problem. All testing is motivated by *risk*. In some sense, it seems odd to use the term "*risk-based* testing", since this is like referring to "air-based living" although we all have to breathe to live. We may notice the importance of air to our lives and plan the usage of it carefully when we are in danger of running out of air. This is somewhat similar to risk-directed testing.

The truth that testing is motivated by *risk* does not mean that accounting for *risks* is explicitly required in managing a test process. In fact, *risks* are usually addressed

implicitly in standard approaches. Traditional testers have always conducted *risk-based testing*, but in an ad hoc fashion based on their personal judgment [Amland 99b]. In the cases that the *risks* are already well understood or the total *risk* of failure is not too high, the ad hoc fashion may be good enough. When the cost of failure in the system under test is extremely high, or the testers have to plan to test the right components at the right time because of limited time or resources, *risk-based testing* can help.

In this research, we provide a method for *risk-based testing*. Specifically, we use *risk* metrics to quantitatively measure the quality of a test suite in our approach. The details of our approach are presented in Chapter 5.

3.3 Risk Analysis

In [McGregor+01] *risk analysis* is defined as a procedure for identifying *risks* and for identifying ways to prevent potential problems from being realized. The output of *risk analysis* is a list of identified *risks* in the order of the level of *risk* that can be used to allocate limited resources and to prioritize decisions.

The *risk analysis* approach includes three tasks:

- Identify the *risk* corresponding to each function or feature.
- Quantify the *risk*.
- Produce a *risk-based*, ranked list of functions or features.

The objective of *risk analysis* is to identify potential problems that could affect the cost or outcome of the project. To attain this goal, statistical models, possibly with failure mode analysis (the most common analytical technique for estimating *risk* [Amland 00]) may be derived and applied.

3.4 Risk Analysis Activities

Karolak defined a model of *risk analysis* activity in his book "Software Engineering Risk Management" in 1996 [Karolak 96]. Amland added the corresponding elements relevant to *risk-based testing* in a test process into Karolak's model in [Amland 99b]. Figure 3-1 illustrates the enhanced *risk analysis* activity model. The oval boxes are Amland's additions. Each activity is discussed below.

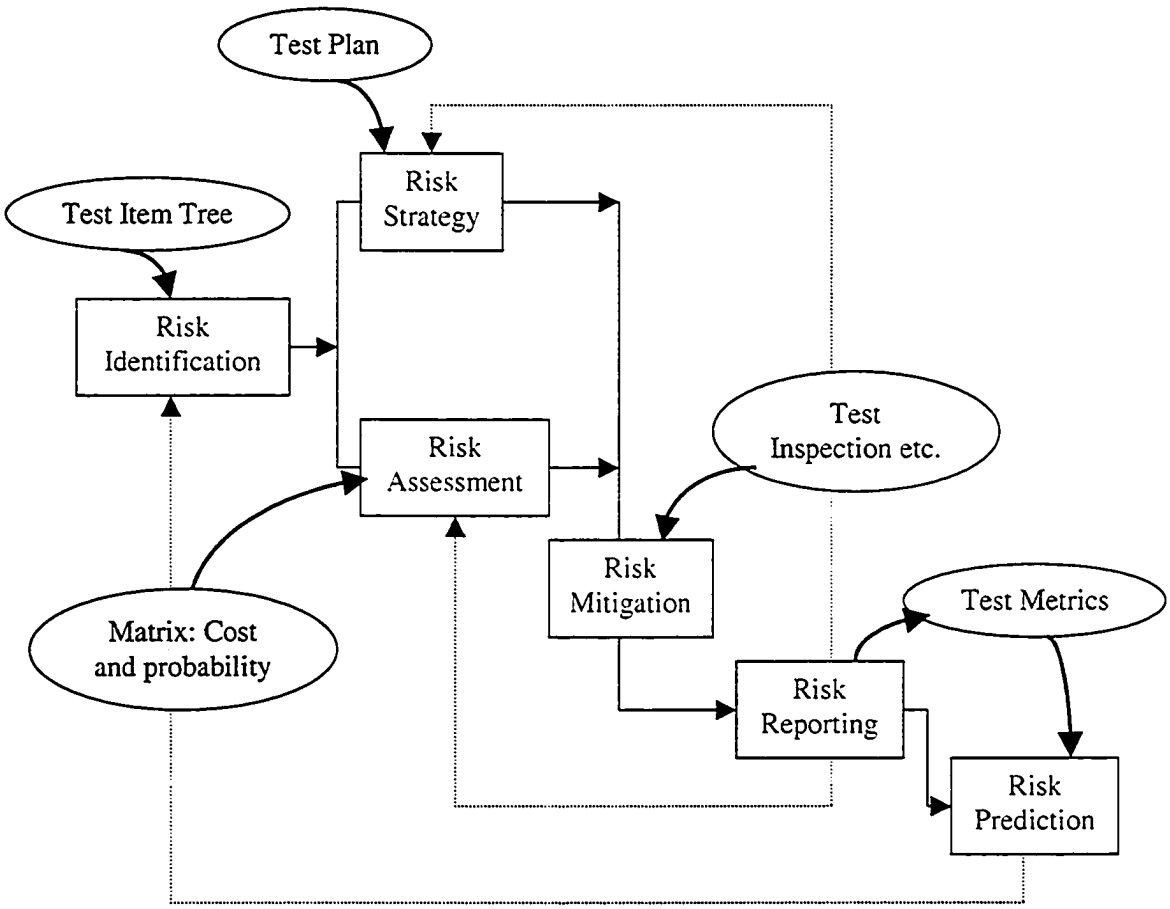


Figure 3-1: Risk analysis activity model

- *Risk object identification*: Collect information about the project and classifying it to determine the amount of potential *risk* in the test phase and in production in the future. The test item tree is the hierarchical breakdown of functions and features in the system under test as preparation for *risk* assessment.
- *Risk strategy*: Identify and assess *risks*. Define an appropriate level of testing per function based on the *risk* assessment of the function. This activity takes a test plan as an input. The risk strategy varies from one project to another. It is dependent on the type of system to be developed and tested, the development and user environment, and other quality and commercial requirements.
- *Risk assessment*: Determine the effects of potential *risks* on functions and features in the test item tree. The determination includes predicting the probability of a failure and identifying the consequences of the failure for each function.
- *Risk mitigation*: Mitigate and avoid *risks* or minimize their impact based on information gained from the previous activities. The idea of mitigation is to use inspection or to focus testing on the critical functions.
- *Risk reporting*: Use test metrics to monitor information obtained from previous testing activities. Typical reports record information such as the number of errors found, number of errors per function, classification of errors, number of hours testing per error, number of hours to fix per error, etc.
- *Risk prediction*: Forecast *risks* using the history and knowledge of previously identified *risks*. Using results from previous activities and test metrics are essential to making correct forecasts. In addition, feedback loops are used to improve the accuracy of elements that are used to forecast *risks*.

3.5 A Practical Risk Model

The definition of *risk* varies from one project to another. It may also be changed over time even within the same project because priorities and development strategies change.

In object-oriented projects, *risks* are specific and typically unique to the architectural features, the areas of complex interactions among objects, the complex behaviors associated with a class specification, and the changing or evolving project requirements. Other definitions of *risk* might be the complexity of the class as measured by the size of its specification, or the number of relationships it has with other classes.

Our research is *specification-based.* We do not consider the relationship between *risk* and code. In *black box testing*, the various uses of the system under test are prioritized based on the importance to the user and the proper operation of the system. *Risk* may also be evaluated based on the complexities of the concepts that must be implemented in different components, the volatility of the requirements in a particular component, or the maturity of domain knowledge within a particular component.

Amland introduced the term *Risk Exposure* and presented a simple risk model with only two elements of *Risk Exposure* in [Amland 00]. We use this model in our research. It takes into account:

- The *probability* of a fault being present.

  Myers [Myers 79] reports that, as the number of detected errors increases, the probability that more undetected errors exist also increases. Thus, if one component has defects that are detected by *full testing* (executing the *full test suite*, which includes all test cases designed to test this component), it is very likely that we can find more defects in this component by *regression testing* because of the increased probability of undetected defects. The more defects detected, the more defects we can expect usually by defining more test cases. Therefore, the more detected defects that a modified component had in *full testing*, the more carefully the component should be tested in *regression testing*.

  Selecting tests that are more likely to discover design omissions is part of an a priori "high-yield" strategy [Saleh+02]. Here, we use *risk analysis* of actual test results to select risk-directed regression test cases.

- The *cost* (consequence or impact) of a fault in the corresponding function if it occurs in the field.

  It is a well-known, observed fact that most commercial software contains bugs at delivery time. Companies begin a project with the knowledge that they will choose, because of schedule pressures, to ship software that contains known bugs [Bach 98]. Some defects cause more serious consequences than others. In our definition, the *cost* is the consequence or impact of a fault. The higher the *cost* associated with a defect, the more critical the defect is. To lower the *risk* of a project when time or resources are limited, we strive to detect and correct the most critical defects first. In other words, we focus on detecting (and repairing) defects with the highest *cost* first.

The formula to calculate *Risk Exposure* is:

$$RE_f = P_f \times C_f$$

where RE_f is the *Risk Exposure* of function *f*, P_f is the *probability* of a fault occurring in function *f* and C_f is the *cost* if a fault is executed in function *f* in operational mode.

3.6 Chapter Summary

In this chapter, we provided background information about *risk-based testing* and *risk analysis*. Also, we discussed the activities that should be contained in *risk-based* testing. Finally, we introduced a simple risk model and provided a formula to calculate *Risk Exposure*. This risk model and formula are used in our research, to guide the selection of *Safety Tests* and Safety Scenarios.

Chapter 4. Approach 1: Specification Modification-Based Regression Test Selection

4.1 Introduction and Motivation

We have separated regression test cases into two categories, namely *Targeted Tests* and *Safety Tests* introduced in Chapter 2. Hence, our regression test technique is divided into two parts respectively: *Targeted Tests* selection and *Safety Tests* selection. In this chapter, we provide a method to select *Targeted Tests* for components that are affected by modifications. The purpose of the *Targeted Tests* is to cover the immediate ripple effect of a code change. In Chapter 6, we will give evidence that the *Targeted Tests* are effective for *regression testing*.

In our research, we use *activity diagrams* as a notation for requirements analysis and design, especially for workflow modelling. Test cases are designed based on an *activity diagram*, which comes from the design document. Our approach includes the process of building traceability links between requirements, specifications, implementation and test cases using the *activity diagram*, and the process of selecting *Targeted Tests* based on these links.

Our method is based on documentation including design documents, change histories, and test execution logs. If documentation is incomplete or incorrect, our method does not necessarily cover all possible changes. We assume that our strategy can be used in conjunction with other regression analysis and testing strategies to cover the entire test suite.

4.2 Requirements Traceability and Activity Diagrams

In Chapter 2, we have pointed out that the first activity of any regression analysis and testing strategy is the identification of components that are affected by the modification (see section 2.2.2). In the case of *corrective maintenance*, we can identify the affected components by telling which components contain the changed piece of code. For *progressive maintenance*, since changes happen in requirements or specification, we need links among requirements, specification and implementation to identify the affected components. Moreover, after all affected components have been identified in the first activity, we select test cases for testing these components as regression test cases in the second activity. A link between affected components and test cases is also required for the selection.

Requirements traceability is a simple, common-sense bookkeeping property that can prevent a wide range of problems [Binder 00], and supports crosschecking by linking requirements, analysis, design, implementation, and test cases. Documenting *requirements traceability* is a reasonable solution for identifying both affected components and test cases that test these components.

4.2.1 Requirement traceability

The term "*requirements traceability*" was first introduced by the US Government's Department of Defense. A commonly accepted definition of *requirement traceability* is the ability to describe and follow the life of a requirement, in both a forward and backward direction [Gotel+94]. That is, traceability is the ability to follow the requirements from their origins, through their specification and development, to the product's subsequent deployment and use, and through periods of on-going refinement

and iteration in any of these phases.

Requirements traceability has been widely recognized as a significant factor for efficient software project management and software systems quality. In the past years, the area of *requirement traceability* has been very active. Several traceability models have been created [Toranzo+99, Harmelen+97]. According to Spanoudakis [Spanoudakis 02], the links we build can be used to:

- Assist the process of verifying that a system meets its requirements.
- Establish the impact of changes in the requirements specification of a system upon other artifacts in its documentation (e.g. design, testing and implementation artifacts) and vice versa.
- Document and understand the evolution of artifacts

The second usage of *requirement traceability* matches our situation perfectly. In our proposal, we use it to address the first problem of regression analysis and testing (see section 2.2.2), namely identifying components affected by changes.

From user requirements to the first product release, a software development process usually passes four major phases: requirement analysis, design, implementation, and testing. As a result, information about the system is transformed into different forms and into different languages in different phases. We choose the UML *activity diagram* to describe the analysis and design models in our research. *Requirements traceability* is derived based on the *activity diagram*.

4.2.2 Activity diagram

In the Unified Modelling Language (UML), the *activity diagram* is the notation for an activity graph, used to model threads of computations and workflows of a system [Rumbaugh+99]. It combines concepts from several techniques: flow graphs, state

transition diagrams, event diagrams (a notation from J. Odell), SDL modelling techniques, workflow modelling, and Petri nets [Fowler+00]. Compared with the sequence diagram [Fowler+00], the *activity diagram* provides a more comprehensive representation. Compared to the use case diagram, the *activity diagram* presents the order in which the behaviors of the system might occur. A use case diagram does not specify precisely to the order of system behaviors. *Activity diagrams* can cut across the logic of several use cases.

Activity diagrams can be used to describe system behavior either in requirements capture phase, or design and analysis phase. It documents the logic of a single operation or method, a single use case, or the flow of logic of a business process. In many ways, the *activity diagram* is the object-oriented equivalent of flow graphs and data-flow diagrams (DFDs) from structured development [Ambler 00]. This technique can be very helpful in the following situations:

- Analyzing a use case: In this case, we need to understand what actions need to take place, in what order, and what the behavioral dependencies are.
- Understanding workflow: Even before we get into use cases, we can easily draw an activity diagram with business experts to understand how a business operates and how it may change.
- Describing a complicated sequential algorithm: In this situation, an activity diagram is an UML-compliant flow graph. The usual pros and cons of flow graphs apply.
- Dealing with multithreaded applications: The *activity diagram* has a set of elements to describe the system behaviors of multithreaded applications.

Some companies such as IBM have begun to use the *activity diagram* in system design. It is a powerful tool for describing system behavior and workflows. It is the latest notation accepted by UML, and is still one of the least understood modelling methods of UML. Even though it is used in industry, academic researchers have seldom paid attention to this technique [Fowler+00]. Some researchers have misunderstood the concept of the *activity diagram* and thought that it is only a control flow graph (CFG). But indeed, the *activity diagram* is much more than a CFG. The two major reasons are as

follows:

- The *activity diagram* can be used to model both control flow and data flow, while CFG is only used for control flow.

  Gunnar pointed out that the *activity diagram* "is intended for applications that need control flow or object/data flow models ..." [Gunna+01].

- The *activity diagram* can be used to model large-scale systems, while CFG represents code and cannot be applied to large systems.

  The *activity diagram* can be nested. We can apply "nested" structure to large systems. In high-level design, we draw an overall diagram. This *activity diagram* is not in detail and not so big. Then, for each subsystem or component, nested *activity diagrams* give out more detailed description. The nested *activity diagrams* can be also used for activities of original activity diagram, and can be invoked by *subactivity* symbol.

In our research, we use the *activity diagram* to specify system requirements for *regression analysis* purposes. Since requirement analysis and design sometimes merge with each other in an object-oriented development process, the *activity diagram*, as an output of the analysis and design phase, can contain more information than the original requirements. In this situation, our approach belongs rather to *grey box testing* than to *black box testing* [Probert 02]. Thus, we choose the term *specification-based testing*.

4.2.2.1 Elements of the activity diagram

The core symbol of the *activity diagram* is the *activity state*, or simply *activity*. An activity is a state of doing something. It can be either a real-word process, such as typing

a letter, or the execution of a software routine, such as a method of a class [Fowler+00].

The definitions of the major notations in the *activity diagram* are as follows:

- *Activity*: A task that needs to be done, either by a human or a software system. Processes and threads are extensions to activities of UML *activity diagram*. In activity diagram, synchronized thread can be also called as *synch activity*.
- *Subactivity*: Holds the place of an activity and invokes a nested *activity diagram*.
- *Start marker*: Establishes the entrance (initial state) of an *activity diagram*. Each *activity diagram* only has one start marker.
- *Stop marker*: Signifies the exit (final state) of an *activity diagram*. Each *activity diagram* can have several stop markers.
- *Decision point*: As in a flow graph, a conditional flow of control is modelled using the decision element. A decision is shown by labelling each output transition of a decision with a different guard condition.
- *Synchronization bar*: Identifies the start and end of potentially parallel processes. With synchronization bars, concurrent transitions are merged into a single target. Also, a single transition can be split into parallel transitions.
- *Signal sender*: The designated signal is emitted when the predecessor activity terminates.
- *Signal receiver*: The successor activity is activated only when the designated signal is received.
- *Transition*: A relationship between two activities. The transition indicates that control is passed from the first activity to the second activity once the work of the source activity is complete. In another word, transitions are triggered by the completion of the source activity. In the *activity diagram* transitions are used to model the flow of order between activities. To emphasize functional flow of control, transitions can be labelled, and contain parameters, guard conditions and action expressions.
- *Guard condition*: a condition that must be fulfilled to proceed along the

transition. Guard conditions are described by text in the format of "[condition]" associated with transitions (on the arrows which are used for transitions).

- *Object*: An object that participates in an interaction. The object may be a component, an entire subsystem, or an interface to an external system. In an activity diagram, object refers to something out of the scope of this diagram, for example, a TCP/IP Network component.
- *Message*: A piece of information sent to or from an object. It can be a mechanism for invocation of a method or process.
- *Swimlane*: A "package" for organizing activities within a group. Each zone or lane represents the responsibilities of a particular group. Swimlanes depict both the activity diagram's logic along with the allocation of responsibility. To use swimlanes, the activity diagram must be arranged into vertical zones separated by dashed lines. Transitions may cross swimlane boundaries. Because swimlanes are intended to represent responsibilities rather behaviors, we do not discuss them further in this thesis.

Symbols used for major *activity diagram* elements are listed in Table 4-1.

The *activity diagram* describes the sequences of activities. This technique is very similar to flow graph modelling familiar to experienced software developers and testers. It can be used to capture the *basic flow*, *alternative flow*, and *exceptional flow* of system execution. The *activity diagram* supports both conditional and parallel behaviors:

- Conditional behavior is delineated by branches and merges
    - *Branch*: A branch has a single incoming transition and several guarded outgoing transitions. Only one of the outgoing transitions can be taken, so the guards should be mutually exclusive. Using [else] as a guard indicates that the "else" transition should be used if all the other guards on the branch are false.

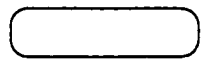
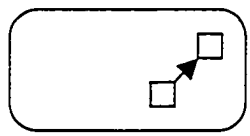
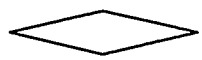
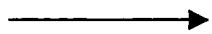
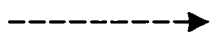
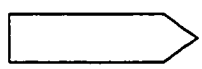
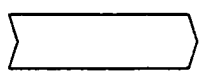
Concept	Symbol
Start marker	Filled circle: 
Stop marker	Filled circle with a border: 
Activity (Include process and thread)	Rounded rectangle: 
Subactivity	
Decision point	Diamond: 
Synchronization bar	Thick bar: 
Transition	Solid arrow: 
Object	Rectangle: 
Message	Dashed arrow: 
Signal sender	A rectangle with a triangular point in either side 
Signal receiver	A rectangle with a triangular notch in either side 

Table 4-1: Symbols of the activity diagram

Related notations of the *activity diagram*: Decision points, transitions (outgoing), guard conditions.

- *Merge*: A merge has multiple input transitions and a single output. It marks the end of conditional behavior started by a branch. A diamond can be used to make the merge explicit in the diagram.

  Related notations of the *activity diagram*: Transitions (incoming).

- Parallel behavior is indicated by a fork or a join
  - *Fork*: A fork has one incoming transition and several outgoing transitions. When the incoming transition is triggered, all of the outgoing transitions are taken in parallel.

    Related notations of the *activity diagram*: Synchronization bars, transitions outgoing).

  - *Join:* A join has multiple incoming transitions and one outgoing transitions. With a join, the outgoing transition is taken only when all the states on the incoming transitions have completed their activities.

    Related notations of the *activity diagram*: Synchronization bars, transitions (incoming).

Figure 4-1 is one simple example of such an *activity diagram.* It represents the system behaviors of an order process feature for an online airline reservation system. We identify notations of the *activity diagram* in Bolded Italics.

In the example system, a customer can place orders using an existing system account, named *subscription*, or place a single order without an account. As shown in Figure 4-1, after an order is received, there is a **branch**. If this is a subscription, the money for this order will be charged to the customer's bank account offer by his system account profile, and award points are added to the customer's system account. If it is a single order, the money is charged to account information associated with this order. When processing a

subscription, the system deals with payment and award in parallel. Thus, there are a **fork** and a **join** in Figure 4-1. Mailing products is a common activity for both subscription and single order after they have been filled, where the **merge** is located. Because the system needs to keep a trace for every order, a **message** is sent to the **object** Log File.

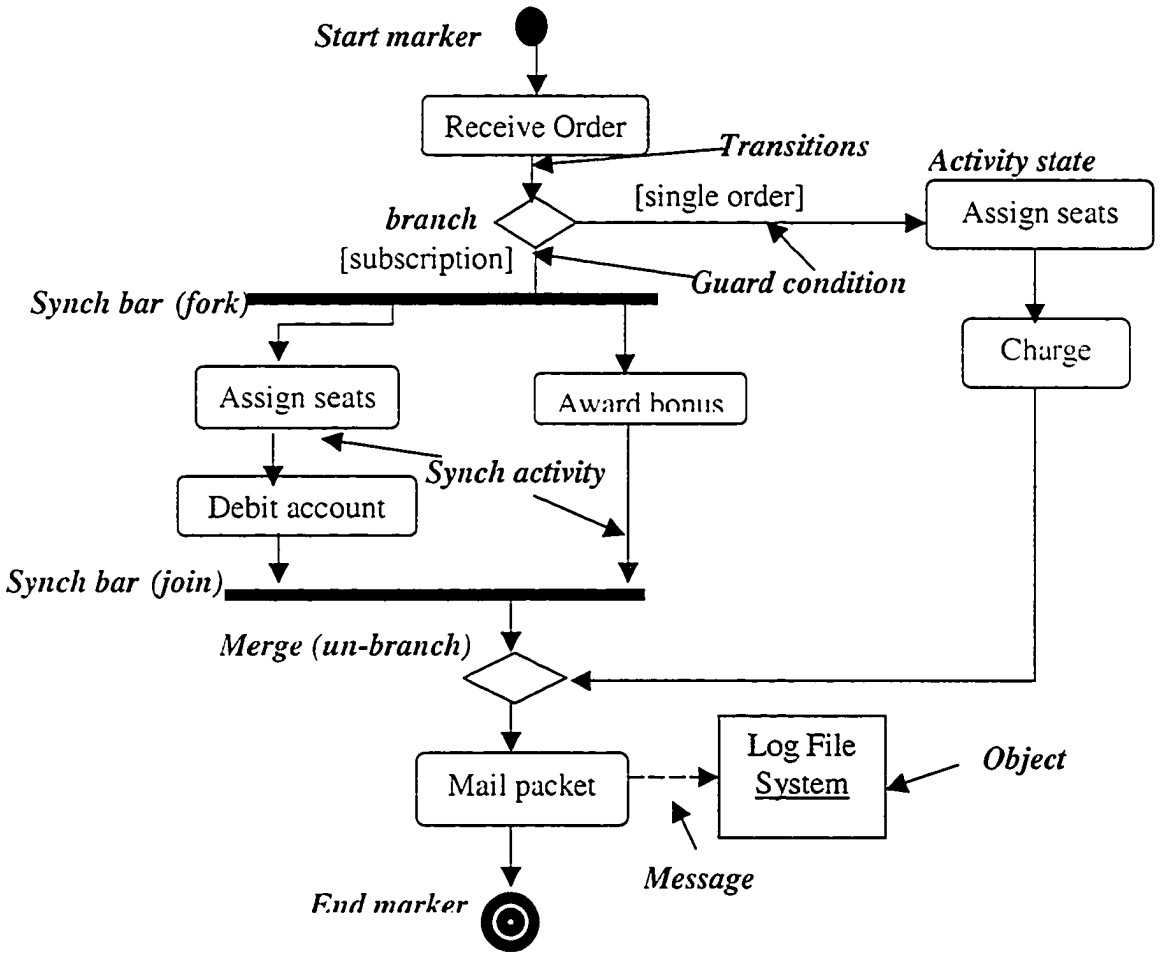


Figure 4-1: A simple activity diagram example

4.2.2.2 Activity diagram and functional test case design

In *specification-based* testing, test cases are designed based on the requirements specification, which is the output of the requirements analysis and design phase. As part of the documentation of requirements capture and design phase, the *activity diagram* is given to developers for detailed design and implementation, and also to testers for *specification-based* test case preparation. In our research, we focus on the relation between the *activity diagram* and the test cases, and we apply this relationship to regression test selection. We discuss issues related to *regression testing*, such as obtaining *requirements traceability*, but not out of scope topics such as how to generate imputs and outputs for a specific test case.

A graph represents relations among objects by using two simple constructs: nodes and edges. Nodes are connected to other nodes by edges. The nodes and edges of a graph represent a relation, which is another simple but powerful mathematical concept. The *activity diagram* can be viewed as a graph that represents many significant relationships, especially control-flow relationships, and therefore provides a rich source of information for test design. Elements of the *activity diagram* related to our research can be categorized into nodes and edges.

- *Nodes*: In an *activity diagram*, a node is shown as a box with a dog-eared corner. It may be connected to another diagram element or it may be free-standing [Binder 00].

  Notations of the *activity diagram*: start and stop markers, activities, decision points, synchronization bars, objects, signal senders, and signal receivers.

- *Edges*: An abstract connection between abstract nodes. A line refers to one or more contiguous line segments rendered on a graph [Binder 00].

  Notations of the *activity diagram*: transitions, messages.

When preparing *specification-based* test cases, we can either design them based on the *activity diagram*, or on other documentation that represents the *requirement attributes*. In this chapter, we suppose that test cases are prepared based on the *activity diagram*, which is the case in some organizations, such as the electronic commerce department (ECD) in IBM. In Chapter 6, we generalize our approach to some other notations.

Test cases are assumed to relate to certain *test purposes*. A *test purpose* may relate to coverage of control flow, data flow, or signal flow. Based on the *activity* diagram, such a test purpose can be oriented towards an specific activity or a path (a sequence of activities) that covers a single system functionality. Here the graph-theoretic sense of path is used: a set of nodes and edges for which an unambiguous property holds [Binder 00]. In a graph, the number of paths is unlimited. Therefore, there may be an unbounded number of paths in an *activity diagram*. Each test purpose is associated with one specific path. Test cases based on a specific *test purpose* may cover the same path, but can carry different test data or have different configurations.

When designing test cases, we always apply one or more test strategies and define appropriate test coverage criteria. A test coverage criterion is a measure of the completeness of software testing. Binder summarized testing strategies for each UML diagram, and provided complete cross-references of UML diagrams and test design patterns [Binder 00]. We do not discuss the process of designing test cases based on the *activity diagram* in this thesis, except for some points about concurrent threads, or *synch activities*.

Synch activities are used for synchronizing concurrent regions to insure that one region leaves a particular activity or activities before another region can enter the particular activity or activities [UML 01]. *Synch activities* exist between synchronization pairs, one for the *fork*, and the other for the *join*. A dashed box in Figure 4-2 marks the part of *synch activities* in our simple activity diagram in Figure 4-1. To identify paths, we label all nodes in the diagram. The edge of the diagram can be defined as a pair:

Edge = (S, D)

where S is the node where this edge comes from, and D is the node of destination.

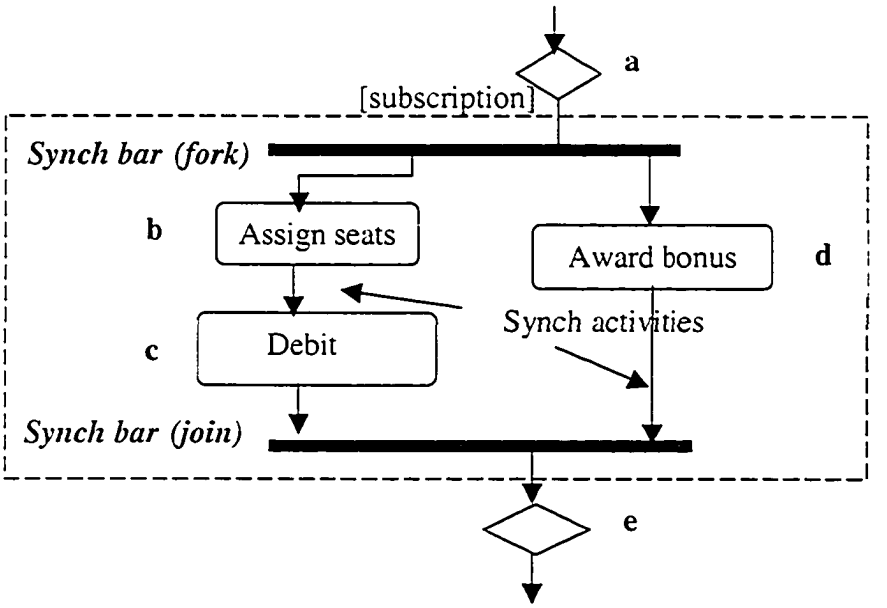


Figure 4-2: *Synch activities* with synchronization *fork* and *join*

For *synch activities*, once the system reaches a synchronization *fork*, all activities between this fork and its respective *join* have to be finished in parallel before the join's outgoing transition is triggered. The execution order of these parallel threads is controlled by the operating system instead of by the system under test. Test cases run through the "box" (the *synch activities*) **always** execute **all** activities in the box with uncontrolled execution order. One choice for testers to describe the execution is using "/" instead of "," to separate parallel threads. For example, we use a, bc/d, e as the path. Because the execution in the "box" is uncontrolled, a better way to describe the execution is to view the "box" of the *synch activities* itself as a node. We name the "box" *synchronization box*. Then we can label the *box* as for other nodes, and use it for the path. We choose **this better way** that uses a *synchronization box* in this thesis. Our construct of *synchronization box* appears to be unique.

To represent the cases when some *synch activities* are not executed, we will not use a synchronization *fork* but only a synchronization *join*. A small example is shown in Figure 4-3. In this case, we look at these *synch activities* as regular *activities* and do nothing to simplify the *activity diagram*.

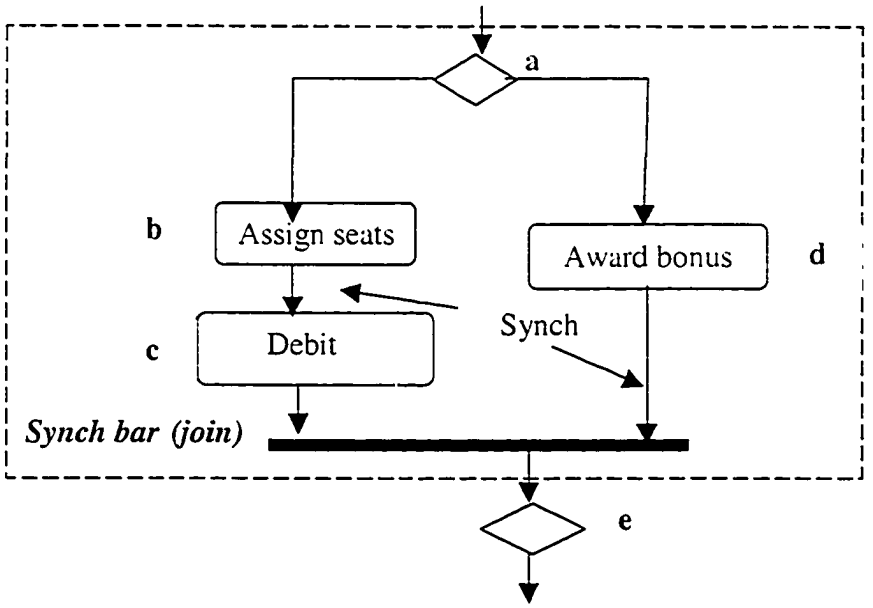


Figure 4-3: *Synch activities* without synchronization *fork*

Figure 4-4 is the example we use to describe *activity diagram*-based test case design, and regression analysis and testing. This is an *activity diagram* for the Get_Quote feature of the EX system, which is a CORBA-based e-commerce system developed for research and teaching purposes by SITE's e-commerce testing research group at the University of Ottawa. It is an online currency exchange system. This system helps users to conduct currency exchange from Canadian dollars into US dollars. This *activity diagram* represents the specification of how the EX system gets a quote of the current exchange rate from banks. The EX system is able to get quotes from three banks. It sends requests

to the banks in parallel. In the diagram, these processes are *synch activities* including nodes *d1*, *d2* and *d3*.

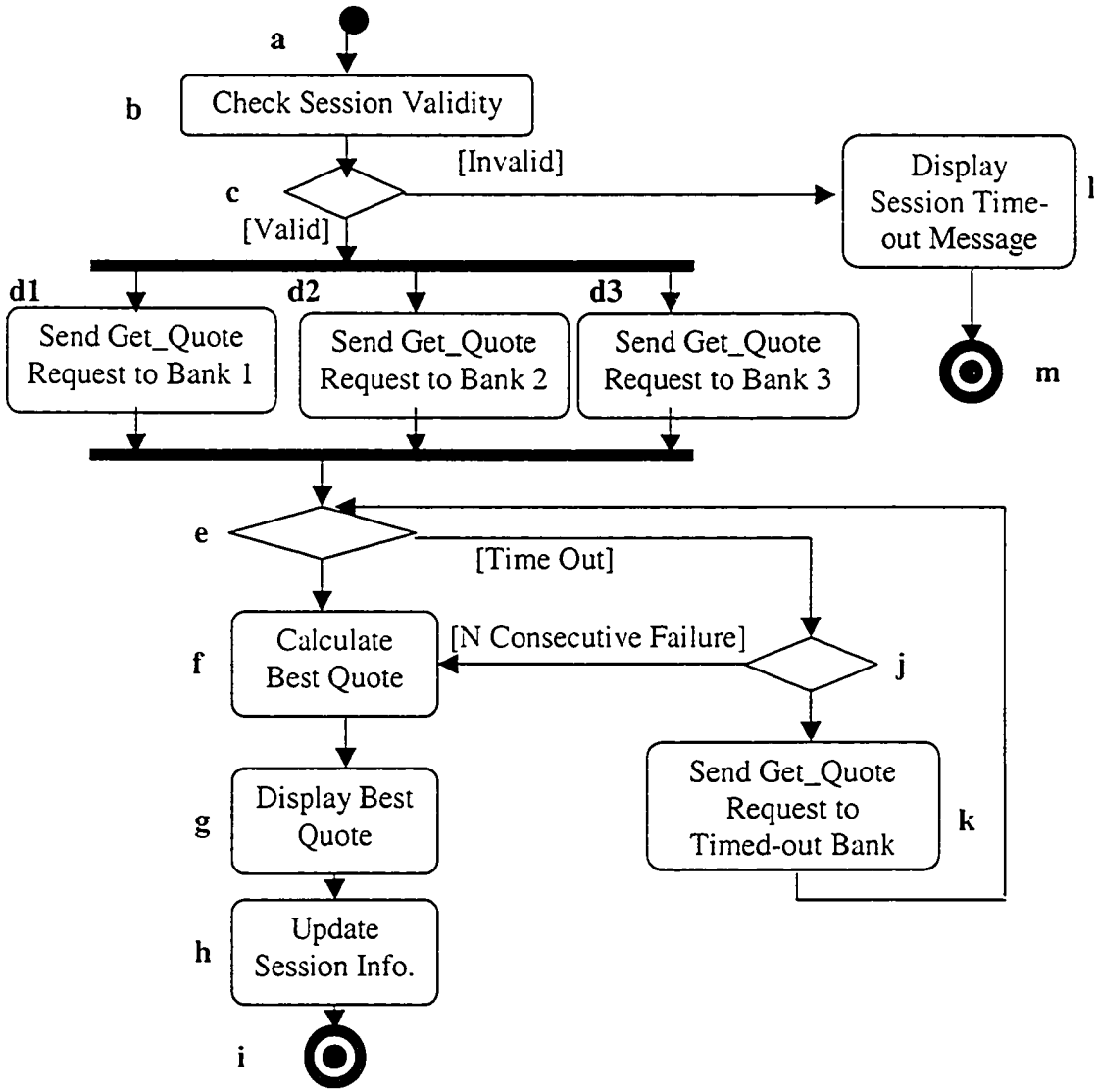


Figure 4-4: The activity diagram for the Get_Quote feature

When we use a synchronization box d to represent the current threads (d1, d2 and d3), the activity diagram in Figure 4-4 is simplified into that of Figure 4-5.

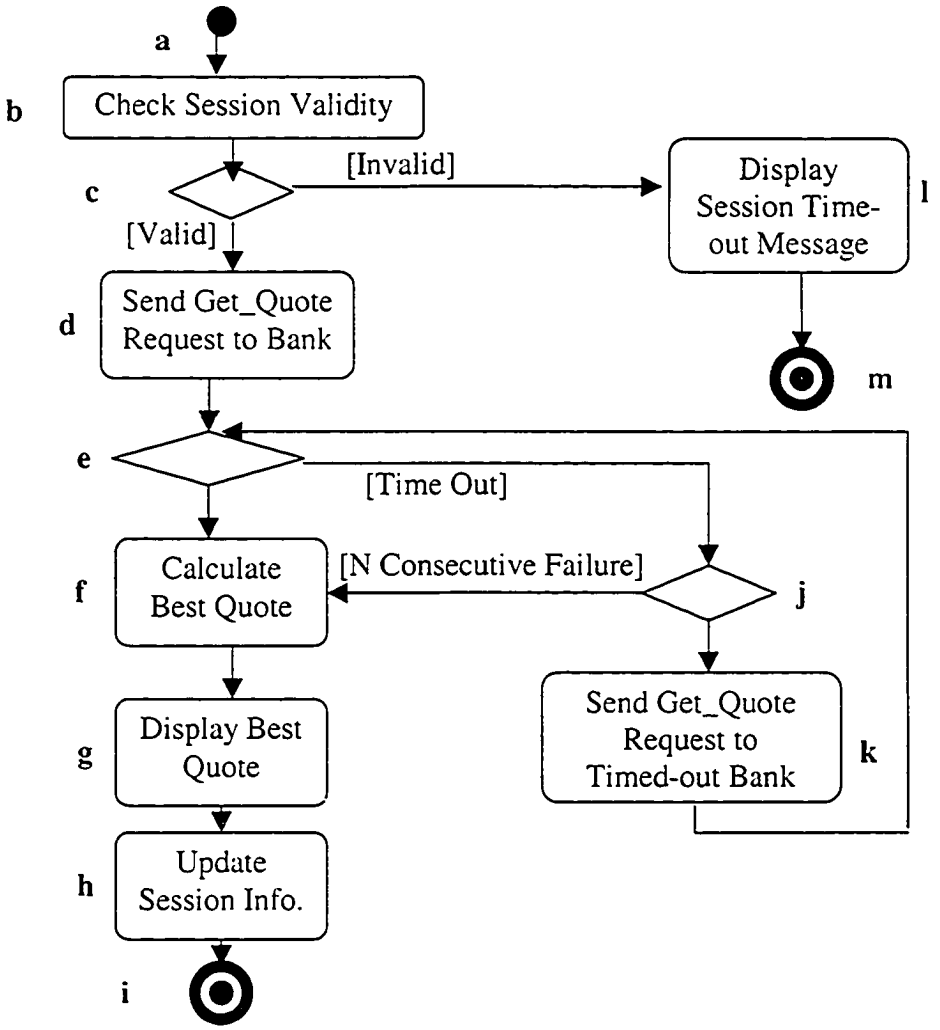


Figure 4-5: The activity diagram for the Get_Quote feature

In our example, we design five test cases for the diagram in Figure 4-5, which are listed in Table 4-2. In this test suite, each test case takes a path from the start marker to the stop marker, and tests a specific system function. t3 and t4 cover the same path but have different system configurations. The test suite exercises all nodes and edges in the diagram, and satisfies our *specification-based* test coverage criteria, namely *node and edge coverage criterion*. Since test case design is out of the scope of this thesis, we do not discuss how to derive these test cases.

Test Case	Path
t1	a, b, c, d, e, f, g, h, i
t2	a, b, c, d, e, j, k, e, f, g, h, i
t3	a, b, c, d, e, j, k, e, j, f, g, h, i
t4	a, b, c, d, e, j, k, e, j, e, k, e, f, g, h, i
t5	a, b, c, l, m

Table 4-2: Test suite for the Get_Quote feature in Figure 4-5 satisfying node and edge coverage

4.3 Building Requirements Traceability Links Based on Activity Diagrams

We have given a detailed discussion of the role of *requirements traceability* in regression analysis and testing, and the *activity diagram* in previous section. In this section, we provide the method of obtaining requirements traceability links based on the *activity diagram.*

4.3.1 Requirements testing, design testing and activity diagrams

In different development phases, we can use different notations to model different views of the system. In our approach, we use *activity diagrams* to represent the desired

system behavior. If the models are complete, consistent, and correct, they provide a solid basis for test design. Testing of the *activity diagram* against the requirements should be an essential part of requirements testing and design testing described below. Table 4-3 is a summary of issues that need to be tested in requirements testing and design testing. A detailed discussion for each test issue is given in Sections 4.3.1.1 and 4.3.1.2.

	Requirements Testing	Design Testing
Correctness	√	√
Completeness	√	√
Consistency	√	√
Feasibility		√
Correctness		√
Traceability		√

Table 4-3: Summary of test issues in requirement testing and design testing

4.3.1.1 Requirements testing

Requirements analysis aims at precisely and clearly defining a problem to be solved. Therefore, the objective of requirements testing is to validate each of the outputs of the analysis phase. Testing of requirements involves three basic issues [Bashir+99]:

- Correctness: Requirements must clearly state the customers' true wishes. Analysts must ensure that there are no vague words in the requirement

statements. The correctness of requirements is a fundamental assumption for designers.

- Completeness: The problem must be completely and clearly specified. Our focus, the functional requirements, must specify all the desired behavior of the system under specified input conditions.
- Consistency: It is very common for customers to give contradictory requirements, or redundant requirements. These incongruent requirements and redundant requirements must be caught and eliminated during requirements testing, to avoid later design problems.

The major acceptance criterion of requirements testing is user satisfaction. However, the requirements have to be feasible within the project time and budget framework. Once the customer and the vendor agree with the requirements, it is time to move onto design phase.

4.3.1.2 Design testing

In the design phase, designers intend to find a solution for defined problems. Their objective is to generate complete specifications for implementing a system using a set of tools and languages. The design stage inherits requirements from the requirements analysis stage, and transforms them into a complete plan for implementing the system. Two primary products of an object-oriented design are a description of the architecture and descriptions of common tactical policies [Booch 94]. Based on the products of the design phase, there are five primary objectives for testing an object-oriented design [Bashir+99]:

- *Consistency*: Inconsistent design is a source of major errors that are not discovered until the later stages and that cause nightmares for the maintenance teams. Thus, the first objective of testing designs is to eliminate

inconsistencies.

- *Completeness*: An important attribute of a good design is that it provides a complete solution to all the problems it is supposed to solve. In design testing, testers need to ensure that the design meets not only its functional requirements, but also other requirements, such as performance requirements.
- *Feasibility*: The time and resources allocated to a project is always limited. A good design has to be able to implement within the specified time and budget.
- *Correctness*: A design must solve the problem at hand. It is correct if its input and output relation can be proved true or false.
- *Traceability*: We have discussed how important traceability is to ensure software quality. The traceability of a design is part of the traceability link of the project. To test traceability, testers need to find antecedents in earlier specifications for the terms in the design.

Design of an object-oriented software system is never finished until the system is delivered [Booch 96]. Two aspects of the acceptance criteria have to be met before moving into the next major phase of the project.

- The development team must have enough confidence in the framework of the design, as well as in the design decisions and its associated *risks*. Testing enhances this level of comfort.
- The testing team must have enough confidence that the test suites are complete, consistent, and correct.
    - Completeness: The test suites cover all the requirements of the system.
    - Consistency: For the same requirements, no two test cases contradict each other.
    - Correctness: The oracle for the test suites has been correctly specified.

Since we focus on *regression testing*, we will not explain here the traceability links between requirement/design testing and the *activity diagrams*.

4.3.1.3 Activity diagram assumption: Correct, completed, thorough tested

The *activity diagram* is the basic for both implementation and test case design. It is part of the output of the requirements analysis and design phases. We suppose that both requirements testing and design testing have been done properly and pass all the acceptance criteria. However, to be safe, we state the following assumption:

Assumption: Our approach assumes that the *activity diagrams* have been thoroughly tested. They represent system specification correctly, completely, consistently, and feasibly. Moreover, they are traceable. Elements of the *activity diagram* can be mapped to *requirement attributes* (requirement statements).

4.3.2 Tracing test cases to activity diagram elements

The purpose of creating linkages between requirements and test cases is to capture the relationship between *requirement attributes* and test cases. In other words, we need to document which test cases are testing each specific requirement attribute. In our *activity diagram* assumption (see section 4.3.1.3), we have assumed that elements of the *activity diagram* can be mapped to *requirement attributes*. If we can find a way of tracing test cases to the elements of the *activity diagram*, by the *activity diagram* assumption, we are able to create links between *requirement attributes* and test cases (see Figure 4-6).

When designing test cases, we list the *path* for each test case. A *path* is a sequence of labelled *activity diagram* elements. Therefore, we can tell which *activity diagram* elements are tested by a specific test case. Actually this relationship is like a two-way street. When we come from the other direction, we are able to tell which test cases are associated with one specific *activity diagram* element. In our approach, we create a traceability matrix to observe the relations between test cases and *activity diagram*

elements based on test case design document.

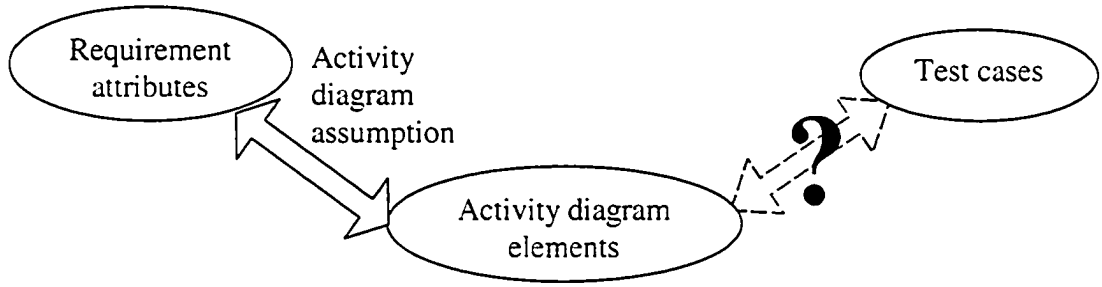


Figure 4-6: Creating traceability links between *requirement attributes* and test cases

We have listed the path from the *activity diagram* pf Figure 4-5 for each test case in Table 4-2. For each test case, we can identify all nodes and edges covered. This yields Table 4-4, which is more useful than Table 4-2 for identifying uncovered nodes and edges.

Test Case	Nodes Covered	Edges Covered
t1	a, b, c, d, e, f, g, h, i	(a, b), (b, c), (c, d), (d, e), (e, f), (f, g), (g, h), (h, i)
t2	a, b, c, d, e, j, k, e, f, g, h, i	(a, b), (b, c), (c, d), (d, e),(e, j), (j, k), (k, e), (e, f), (f, g), (g, h), (h, i)
t3	a, b, c, d, e, j, k, e, j, f, g, h, i	(a, b), (b, c), (c, d), (d, e),(e, j), (j, k), (k, e), (j, f), (f, g), (g, h), (h, i)
t4	a, b, c, d, e, j, k, e, j, f, g, h, i	(a, b), (b, c), (c, d), (d, e),(e, j), (j, k), (k, e), (j, f), (f, g), (g, h), (h, i)
t5	a, b, c, l, m	(a, b), (b, c), (c, l), (l, m)

Table 4-4: Test cases and activity diagram elements

By simply re-organizing the information in Table 4-4, we create a test case traceability matrix (Table 4-5) to display and crosscheck the links between test cases and *activity diagram* elements. In this matrix, test cases are used as the dependent variables. For each node or edge of the *activity diagram*, we list the test cases that pass through it. For instance, for node *j*, t2, t3, and t4 are listed in the tenth row of the traceability matrix. For edge (e, f), t1 and t2 are listed in the sixth row.

Node	Test Case	Edge	Test Case
a	t1, t2, t3, t4, t5	(a, b)	t1, t2, t3, t4, t5
b	t1, t2, t3, t4, t5	(b, c)	t1, t2, t3, t4, t5
c	t1, t2, t3, t4, t5	(c, d)	t1, t2, t3, t4
d	t1, t2, t3, t4	(c, l)	t5
e	t1, t2, t3, t4	(d, e)	t1, t2, t3, t4
f	t1, t2, t3, t4	(e, f)	t1, t2
g	t1, t2, t3, t4	(e, j)	t2, t3, t4
h	t1, t2, t3, t4	(f, g)	t1, t2, t3, t4
i	t1, t2, t3, t4	(g, h)	t1, t2, t3, t4
j	t2, t3, t4	(h, i)	t1, t2, t3, t4
k	t2, t3, t4	(j, f)	t3, t4
l	t5	(j, k)	t2, t3, t4
m	t5	(k, e)	t2, t3, t4
		(l, m)	t5

Table 4-5: Traceability matrix of activity diagram versus test cases

The traceability matrix provides a link between test cases and elements (including nodes and edges) of the *activity diagram*, and provides the traceability chain shown in

Figure 4-7. In this thesis, we apply this matrix in regression analysis.

The process of extending the traceability matrix to obtain full *requirements traceability* is shown in Figure 4-7.

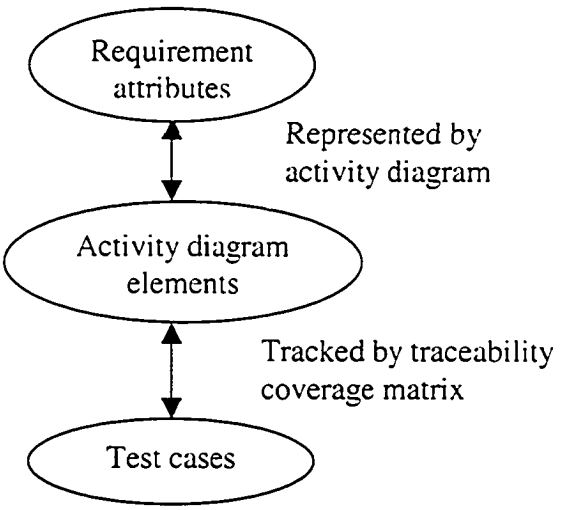


Figure 4-7: A traceability chain of links between requirement attributes and test cases

This type of traceability also provides a check of the testability of all *requirement attributes*, along with the ability to check the completeness of traceability. In the last few years, the term *testability* of requirements has been defined, and has been generally accepted, as a prediction of the probability of software failure occurring due to the existence of faults [Bashir+99]. Requirement testability checks that every requirement attribute is covered by at least one test case. The traceability chain we created between requirement attributes and test cases maps requirement attributes to test cases, and therefore, provides a way to check requirement testability. Viewed from the other side, we can also check if every test case covers at least one requirement attribute.

4.4 Regression Analysis and Targeted Tests Selection

In this section, we discuss selecting test cases, namely *Targeted Tests*, to cover immediate ripple effects of modifications. As we mentioned in Section 2.4, by analyzing changes in existing systems, Leung and White identified that there are really only two types of changes to be considered for practical purposes [White 91]: *corrective maintenance* and *progressive maintenance*. In *corrective maintenance*, changes only happen in the implementation without affecting the specification. In the case of *progressive maintenance* where changes happen in requirements or design, the *activity diagram* needs to be modified accordingly. For changes in the code only, we assume the *activity diagram* will stay the same. We justify and illustrate this assumption in the next section.

4.4.1 Corrective maintenance

As we described in Section 2.4, this type of maintenance does not affect specifications. It occurs quite often during the development process, that is, when developers fix defects, they frequently change only the code. In this case, the prescribed system behaviors (requirements and specifications) are not changed. Therefore, it appears we do not need to change the *activity diagram*. However, sometimes code changes result in significant changes in system behavior, and therefore, such changes should be documented.

For most developers, any changes in code will be documented in a code change history. In our approach, this document additionally needs to identify all nodes and edges in the *activity diagram* whose corresponding implementation has been changed.

For a tester, traceability needs to be established between the test cases and the log of

defects. We (and some organizations) refer to this as a *test profile*. When opening a defect, the tester is required to associate the defect with the specification. In our approach, defects are assigned against the *activity diagram*, which means the elements (nodes and edges) of the *activity diagram* where the defects occurred have to be identified.

In practice, developers may miss some defects, and the code change history documentation may not be complete. With our approach, testers can select regression tests relying not only on documents from the developers, but also on test profiles owned by testers.

After identifying all elements whose implementation has changed based on the code change history and test profile, the tester can choose test cases that need to be included in the regression test suite using the traceability matrix.

In our example, as shown in Figure 4-8, a defect is opened for node *k*, and some modifications are made in the implementation for edge (*c*, *l*). From Table 4-4 we know that test cases t2, t3, and t4 pass through node *k*, and test case t5 is associated to edge (*c*, *l*). Thus, test cases t2, t3, t4, and t5 have to be included in the regression test suite.

4.4.2 Progressive maintenance

As addressed in Section 2.4, both *adaptive maintenance* and *perfective maintenance* are type of *progressive maintenance*. They are maintenance activities that cause not only code changes, but also requirements or specifications changes. Therefore, it appears we need to change the *activity diagram*.

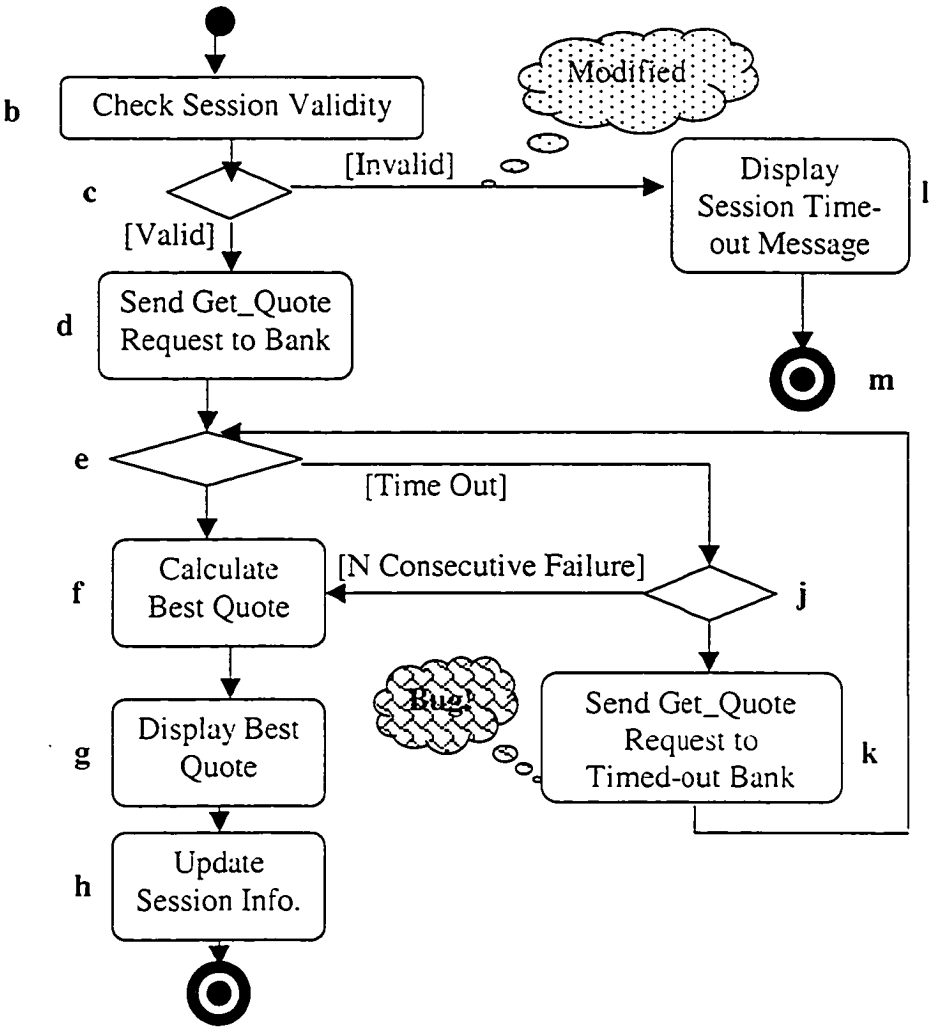


Figure 4-8: Bugs and modifications in the implementation of the Get_Quote feature

In Section 4.2.2, we mentioned that the *activity diagram* borrows concepts from flow graphs. It supports all elements of basic flow graphs. In a control flow graph (CFG), nodes are used to represent statements, and edges represent the control flow between the statements within a procedure (a CFG can also be used to represent inter-process control flow) [Rothermel+00]. We show a sample CFG *C* in Figure 4-9. Comparing Figure 4-5 with Figure 4-9, we find that the *activity diagram* is quite similar to the CFG except that the nodes of the *activity diagram* describe activities instead of code statements.

4.4.2.1 An existing CFG-based regression test selection technique

Rothermel, Harrold, and Dedhia present a control-flow based regression test selection algorithm. They use CFGs to represent the implementation of procedure P and its revision P' and use edges in the CFGs as potential *affected entities* [Rothermel+00]. An entity is denoted as *affected entity* if it is affected (changes its behavior) by the modification. By traversing the CFG for P and the CFG for P' in parallel, *affected entities* are selected. Whenever the targets (target labels) of like-labelled CFG edges in P and P' differ, this edge is added to the set of *affected entities*.

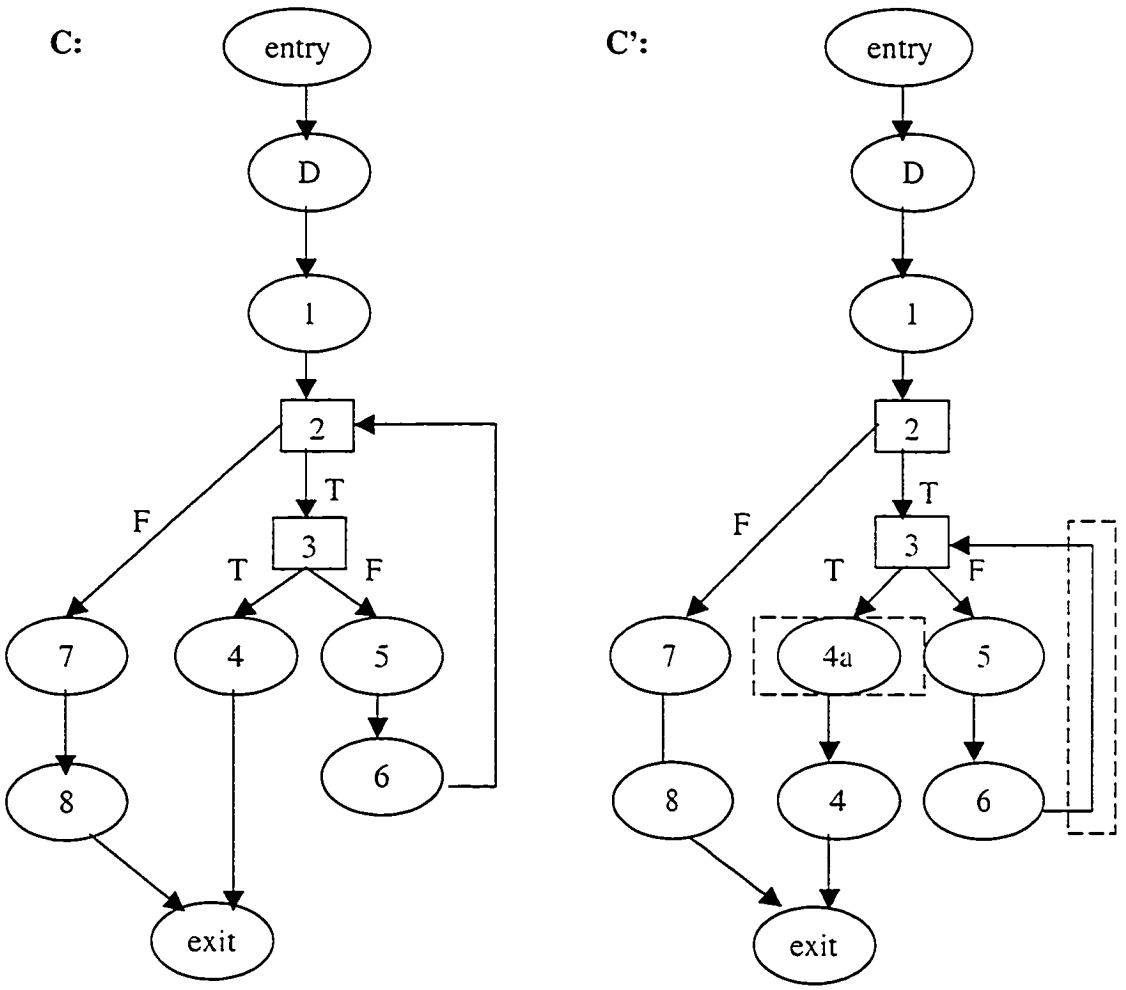


Figure 4-9: Example control-flow graph *C* and its modified version *C'*

In Figure 4-9, we have a sample CFG *C* on the left with its modified version *C'* on the right. In *C'*, a node 4a has been inserted and edge (6, 2) has been changed to edge (6, 3). The *affected entity* algorithm begins the traversal at entry nodes in *C* and *C'*, and traverses like paths in the two graphs by traversing like-labelled edges until detecting a difference in the target nodes of these edges. When the algorithm reaches node 3 in *C* and *C'*, it finds that the targets of the branches labelled "T" differ. It adds edge (3, 4) to the set of affected entities and stops its traversal along this path. The algorithm then considers the edges labelled "F" from node 3. When it reaches node 6 in *C* and *C'*, it discovers that the targets of the "out" edges differ; therefore, it adds edge (6, 2) to the set of affected entities, and stops its traversal along this path. Note that there might be additional changes that occur later in the same path, but we do not need to consider these now, since before it reaches these changes, a test case will need to pass the first change. Identifying the first change is enough for identifying test cases for later changes. Let us assume there were no additional affected edges found in subsequent traversals.

After all affected edges have been identified, they are used with the edge-coverage matrix to guide the selection of test cases.

For example, suppose for *C* in Figure 4-9 we have a test suite T consisting of test cases t1, t2, and t3. Suppose the edge-coverage matrix for this test suite is as shown in Table 4-6. We can see that this edge-coverage matrix is similar to the edge/test case part of our traceability matrix (Table 4-5).

Edge	Test Case
(entry, 1) (1, 2)	t1, t2, t3
(2, 3) (3, 4) (4, exit)	t1, t2
(3, 5) (5, 6) (6, 2)	t2
(2, 7) (7, 8) (8, exit)	t3

Table 4-6: Edge-coverage matrix for test suite T on CFG *C*

Using the edge-coverage matrix and the set of affected entities, we can search the matrix for affected entities and select corresponding regression test cases easily. In our example, with the affected entities, edges (3, 4) and (6, 2), test cases t1 and t2 should be added to the regression test suite.

We now adapt this approach to *specification-based* regression coverage analysis based on the *activity diagram*.

4.4.2.2 Regression coverage analysis with respect to the activity diagram

As shown in previous sections, UML *activity diagrams* are used to capture the "roadmap" of the processing flow as a series of steps. This technique is very similar to flow graph modelling familiar to experienced software developers and testers. The primary difference here is the level of abstraction used to focus attention on critical processing details while hiding much of the underlying code complexity. These diagrams quickly capture and summarize the system dependencies at a processing level leading to a better architectural definition of the system. The first category of regression test cases in our approach is *Targeted Tests*, which test customer-visible *affected entities*. To select this type of test case we first have to identify *affected entities*.

The regression test selection technique that we present is based on the *activity diagram*. Since the *activity diagram* is quite similar to the CFG, we apply the CFG-based algorithm to analyze the *activity diagram* for regression test selection. Like the CFG-based technique described previously, our technique also has two main steps. First we traverse the *activity diagram* to identify affected edges. Then we select test cases that execute the affected edges based on the traceability matrix to create our *Targeted Tests*. Given an *activity diagram A* and its modified version *A'*, the complete method for deriving the *Targeted Tests* is as follows:

Activity diagram-based Targeted Tests Selection Method:

Step 1. Derive set of *affected entities*.

Begin at *start markers* in *A* and *A'*, and traverse like paths in the two diagrams by traversing like-labelled edges. When the algorithm reaches like nodes that have different targets of outgoing edges, put the outgoing edge of the original diagram *A* into the set of affected entities, and stop traversing along this path. Keep on running the algorithm until all paths in *A* have been either completely traversed (have reached *end markers*), or the traversing of the paths was terminated because of detecting *affected entities*.

Step 2. Select *Targeted Tests*.

For each affected entity, search the traceability matrix and select corresponding test cases as the *Targeted Tests*.

Figure 4-10 illustrates this with simple example. Specification changes cause modifications to the *activity diagram* in Figure 4-5; for example, edge (*j*, *f*) has been changed to edge (*j*, *n*). In Figure 4-10, the old edge (*j*, *f*) is crossed out by "X", and the new edge (*j*, *n*) is indicated by a bolded arrow. Let us denote the *activity diagram* in Figure 4-5 by *A*, and the *activity diagram* in Figure 4-10 by *A'*. The algorithm begins the traversal at *start markers* in *A* and *A'*, and traverses like paths in the two diagrams by traversing like-labelled edges in parallel. When the algorithm reaches node *k* in *A* and *A'*, it discovers that the targets of the "out" edges differ; therefore, it puts edge (*j*, *f*) into the set of *affected entities*, and stops its traversal along this path. There are no additional affected edges found in subsequent traversals. From Table 4-5 we know that test cases t3 and t4 cover edge (*j*, *f*). Thus, test cases t3 and t4 have to be included in the regression test suite.

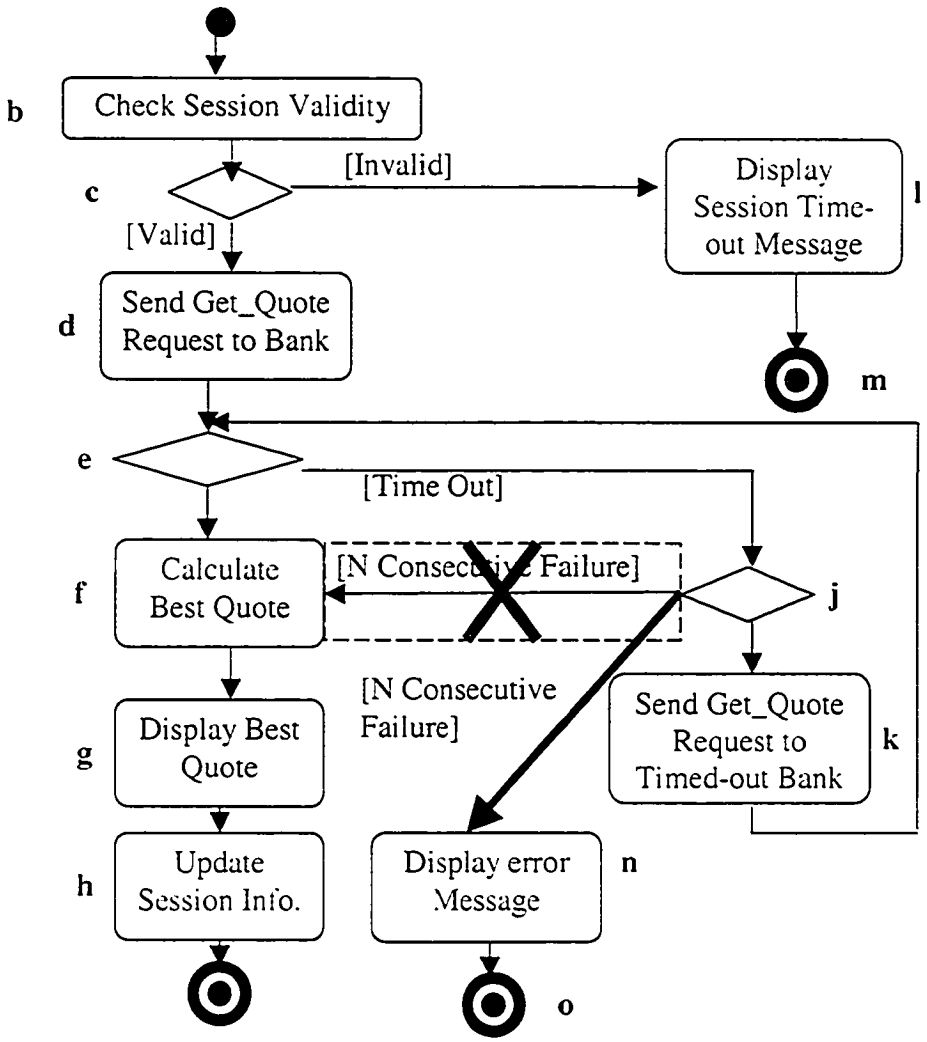


Figure 4-10: Modification to the activity diagram of the Get_Quote feature in Figure 4-5

As a result, we derive a list of test cases that are chosen for *Targeted Tests* to test components affected by modifications. For the remaining test cases, we define them as *non-Targeted Tests*, which will be considered during *Safety Tests* selection, described in the next chapter.

4.4.3 Combination of corrective maintenance and progressive maintenance

In practice, corrective maintenance and progressive maintenance may sometimes happen together. That is, both specifications and implementations for a requirement attribute are changed. In this case, the corresponding *activity diagrams* will be modified, and the implementation of some nodes or edges of the *activity diagrams* is changed as well.

To deal with this situation, we separate the corrective maintenance and progressive maintenance of a requirement attribute, and select test cases for these two types of changes individually. The process order does not make any different to the result since the *Targeted Tests* are the union of test cases for both types of changes.

For example, in Figure 4-10, besides the change from edge (j, f) to edge (j, n), the implementation of node k is also changed. First, we choose test cases for changes in node k, which are t2, t3, and t4. Secondly, we select test cases t3 and t4 for affected edge (j, f) (see Section 4.4.2.2). Finally, the *Targeted Tests* include t2, t3, and t4, which is the union of test cases for node k and edge (j, f).

4.5 Chapter Summary

This chapter focused on the first part of our regression test technique, which is called *Targeted Tests* regression test selection, to select test cases to cover the ripple effects of code changes. In this chapter, we first discussed requirement traceability and its role in regression analysis. Then we introduced the *activity diagram* as the basic of our research and provided a method to assess and document *requirement traceability* based on the *activity diagram*. Also, we defined and illustrated our approach for *Targeted Tests*

selection. This technique is different than Rothermel, even though the algorithms are similar because *activity diagram* is a grey-box technique that can be applied to large applications.

In the next chapter, we proceed to present a complementary strategy to select *Safety Tests* out of the *non-Targeted Tests.*

Chapter 5. Approach 2: Risk-based Regression Test Selection

5.1 Introduction

In Chapter 4, we presented the first part of our regression test selection approach, namely selecting *Targeted Tests* for components that are affected by modifications. The purpose of *Targeted Tests* is to cover immediate ripple effects of changes. However, as we mentioned in section 2.5, we would also like to choose some of the *non-Targeted* test cases to increase confidence in the software quality. This is the second part of our approach, namely *Safety Tests* selection. This is complementary to the *Targeted Tests* selection, and we recommend that both parts of the approach be used.

Risk analysis is the basis for *Safety Tests* selection. In our research, we consider both *regular test cases* and *end-to-end scenario test cases* for *Safety Tests*. Definitions and examples of these two types of test cases will be given in section 5.3 and 5.4. Methods of choosing these two types of *Safety Tests* are presented separately.

5.2 Motivation for Risk-based Safety Tests Selection

The *Targeted Tests* selection method we presented in Chapter 4 is intended to select only test cases that test changed components. Because our regression testing technique is *specification-based*, some test cases that will reveal defects of the modified program might be omitted. There are two major reasons for such an omission.

- Complexity: The relationship between components of current software systems is very complicated and difficult to capture precisely. Once the code is changed, the developer will have difficulty in identifying all affected components, especially when changes were made to commonly referenced functions. For example, a change in a string processing function will cause all components that use this function directly to be retested, such as input/output functions. Besides these, many components that call the string processing function indirectly may be affected as well, e.g., those that use the input/output functions.
- Incomplete or obsolete documentation: Documentation is the basis of *specification-based* testing. To select *Targeted Tests* correctly, we need correct and complete documentation that records all *corrective maintenance* and *progressive maintenance* modifications in the *activity diagram*. This is essential to compute the *affected entities* as shown in Section 4.4. However, in practice, it is very common that a document is out-of-date or incomplete. If a developer modifies code without documenting the change, and this change cannot be identified by the *test profile* either (the changed code was without defects in the previous tests), the tester may miss some test cases that might reveal defects.

To ensure software quality and improve our test confidence, we would like to include some test cases other than the *Targeted Tests* for our regression test suite. This is the second category of regression test cases in our approach, the *Safety Tests*. This composition of regression test selection techniques is unique to our approach.

Rothermel et al [Rothermel+00] separate regression testing into two phases: the *preliminary phase* and the *critical phase*. The *preliminary phase* begins after the release of some version of the software. In software development, under the incremental model, the development of a new increment also belongs to this phase [Rothermel+00]. In the *preliminary phase*, developers are still enhancing and correcting software. When corrections are completed, the *critical phase* of regression testing begins. During the *critical phase*, regression testing is the controlling activity. *Safety Tests* are useful in two

distinct situations especially in this critical phase:

- After running the *Targeted Tests*, if time and resources are available, testers may want to rerun some test cases other than the *Targeted Tests* to gain more confidence in the modified system before general release.
- The time to delivery may be extremely short and there may be no time left for the *critical phase* due to development schedule overrun in the *preliminary phase*. *Safety Tests* provide some assurance that the remaining defects in the release will not bring about serious failures. In this case, we select *Safety Tests* from the *full test suite* instead of the *non-Targeted Tests*.

In both situations, the *Safety Tests* are good choices.

In the *critical phase*, the time is limited by product release deadlines. Thus, using time and resources efficiently is very important. An efficient and effective regression testing technique can be essential for obtaining cost-effectiveness in the *critical phase*. *Risk-based testing* focuses testing on key functions and critical functions. The more important a component is, the higher priority it has. In *risk-based testing*, un-tested components have been judged (subjectively) to be at lower *risk* than components that have been tested. In a limited time, we can achieve high confidence in the product by using risk coverage to select *Safety Tests*. Therefore, we recommend a *risk-based* method to select *Safety Tests* in our approach.

In the following two sections, we present our method for selecting *Safety Tests*. First, we talk about test case selection from regular test cases. Secondly, we discuss a method for selecting regression tests from end-to-end scenario test cases. Both types of test cases need to be discussed because they are applied in different test phases.

5.3 Risk-based Test Case Selection of Safety Tests

In this section, we provide a method to select test cases for *Safety Tests*. The objective is to choose test cases with the highest *risk* coverage. Our approach uses the risk model presented in Chapter 3 for *risk analysis*. In this model, *risk* is quantified and measured by *Risk Exposure* defined in Section 5.3.3. First, we calculate the *Risk Exposure* for each test case (RE_t), the subscript stands for test case. Then, we choose test cases with the highest RE_t for our regression suite of *Safety Tests*. There are four main steps in our approach:

Model-based Safety Tests Selection Method:

Step 1. Assess the *cost* (impact of potential failure covered by this test case) C_t for each test case.

Step 2. Derive *severity probability* P_t for each test case.

Step 3. Calculate *Risk Exposure* RE_t for each test case.

Step 4. Select test cases with top RE_t as *Safety Tests*.

We define and illustrate these terms and our approach with example data, which comes from a real product.

5.3.1 Assess the cost C_t of failure (impact of potential failure covered by this test case) for a given test case t (Step 1)

In the risk model, *cost* is defined as the consequence or impact of a fault in the corresponding function if it occurs in operational model at a customer site. We believe

cost is associated with the function where the fault happens. The more important the function is, the higher *cost* is a failure caused by a fault in this function. In this definition, the term "function" does not refer only to a system feature. It can be generalized to various components of the system, such as a requirement attribute, a system function, a component, or a subsystem. A big component may consist of several small components, that is, a system function may cover several *requirement attributes*, and a sub-system may include several components.

Throughout this thesis, we use the term "cost of a fault" for the meaning of "cost of a failure caused by a fault".

To make cost assessment more reasonable, two kinds of costs will be taken into consideration:

- The consequences CC of a failure as seen by the customer, e.g., losing market share, not fulfilling government regulations, being assigned legal liability, etc., because of faults.
- The consequences CV of a failure as seen by the vendor, e.g., higher software maintenance costs, increased probability of negative publicity, and/or damage to reputation and market share, etc., because of faults.

In our case studies, we are not sure which should be weighted heavier than the other. However, our approach is sufficiently flexible to allow weights to favour either customer or vendor interests. In our case studies, and in this thesis, however, we take the average of CC and CV as our *cost* C. The formula for calculating C is therefore:

$$C = (CC + CV) / 2.$$

In the last chapter, we described how traces (links) are derived between test cases and *activity diagrams* modeling requirements. Since nodes and edges of the *activity diagram* can be mapped to *requirement attributes*, we are able to link test cases to *requirement attributes* (see sections 4.2 and 4.3). Thus, we can associate the failure *cost* of *requirement attributes* with the failure *cost* of test cases. In this thesis, we define failure

cost (or simple *cost*) *of test case* C_t as the *cost* of the *requirement attributes* tested by this test case. Therefore, we have the following formula to calculate C_t:

$$C_t = (CC_t + CV_t) / 2.$$

Of course, both CC_t and CV_t have to be given values to calculate C_t. We now describe a useful means of estimating CC_t and CV_t as a value on a one to five scale (1 – Low, 5 – High).

5.3.1.1 Customer fault cost (CC)

When testers test a software system, they should simulate the behaviors of customers running the operational system. Testers are therefore highly encouraged to understand how customers use the system. Based on their experience and knowledge of the customers, testers are often able to estimate CC_t.

Depending on the *activity diagram*, each test case takes one, specific control flow and includes some specific data. To estimate CC_t of a set of test cases, we created a questionnaire for testers with questions related to both the control flow and the data. Some examples are:

1. Does the test case test a new feature?
2. Does the test case test a changed feature?
3. How often will the customers use the feature?

If several features are tested by one test case, often the *test purpose* is only really focused on one of these features. In the case of more than one features under test, we need to consider these feature together. For each question in the questionnaire, multiple choices with associated points are provided. The choices with points for the three above example questions are as follows:

1. Does the test case test a new feature?

   Yes – 2 points No – 0 points

2. Does the test case test a changed feature?

   Yes – 2 points No – 0 points

3. How often will the customers use the feature(s)?

   Very frequently – 2 points Frequently – 1 point Seldom – 0 points

When answering these questions, testers score each test case based on their answers. For example, for test case t0010, if a tester chose "Yes" for Question One, "No" for Question Two, and "Very frequently" for Question Three, test case t0010 obtains two points from Question One, zero points from Question Two, and two points from Question Three accordingly. After the tester gives answers for all test cases, he or she calculates the total score for every test case. For each test case, its total score is the sum of the points that this test case receives for all questions. In our example, the total score for test case t0010 was 17. Scores for some test cases in our case studies are shown in Table 5-1.

Once scores for all test cases have been tabulated, we assess CC_t in this way:

- For test cases in the top 20% of scores, $CC_t = 5$
- For test cases scoring between the top 20% and 40%, $CC_t = 4$
- For test cases scoring between the top 40% and 60%, $CC_t = 3$
- For test cases scoring between the top 60% and 80%, $CC_t = 2$
- For test cases in the bottom 20% of scores, $CC_t = 1$.

The last column of Table 5-1 shows CC_t for some test cases in our case studies. For example, test case t0010 has score 17, which belongs to the top 20% of course. Thus, its CC_t is assigned 5.

Test Case	Q1	Q2	Q3	...	Total	CC_t
t0010	2	0	2	...	17	5
t0020	0	2	2	...	8	3
t0030	2	0	2	...	6	2
t0040	0	2	1	...	12	4
t0050	0	0	1	...	3	1
t0060	2	0	0	...	5	2
t0070	0	0	1	...	3	1
.....	...	...	...	...	...	.....

Table 5-1: CC of some actual test cases

Different companies may have different focuses on testing, and the focus can vary considerably among different testing phases even within the same company. Moreover, different components may have different characteristics. Hence, the questionnaires are company-dependant, phase-dependant and component-dependant. A sample questionnaire appears in Appendix A. This questionnaire consists of some general questions. We call it our *basic questionnaire*. In our case studies, testers created their own questionnaire by adding new questions to the *basic questionnaire* depending on the test purposes and the characteristics of components under test.

5.3.1.2 Vendor fault cost (CV)

For a vendor, the maintenance cost is the major part of the *cost* of a fault. In this research, we consider the *cost* rather at the component-to-component level than at the

individual LOC level. The more complicated a system is, the higher the cost is needed to fix bugs in it. Therefore, we measure the cost to the vendor CV_t by measuring the complexity of the system.

As we mentioned before, test cases consist of specific flows of control and specific test data. Obviously the complexity of the control flow affects the complexity of a test case. But it is easy to miss some control effects that are caused by the test data. For example, a payment process deals with several payment methods. Here, the payment process belongs to control flow, while the type of payment method is test data. Since some payment methods are more complicated than others, test cases that test these payment methods are more complicated. Hence, we consider complexity of both control flows and test data in our complexity measurement. In this thesis, we use $CompCF_t$ for the complexity of control flows, $CompD_t$ for the complexity of test data, and $Comp_t$ for the complexity of test cases. The formula we use to calculate the complexity of a test case is:

$$Comp_t = CompCF_t \times CompD_t.$$

Since developers are more knowledgeable about control flows, we obtain our system complexity estimates by giving complexity questionnaires to the developers.

After testers have finished their test plan document, developers are asked to estimate the complexity of the control flows and test data on a one to five scale, where one is for the simplest ones and five is for the most complicated ones. The estimation can either be done formally based on complexity theory, or be done in an informal way based on questionnaire. In this thesis, we employ a questionnaire instead of applying metrics from complexity theory, such as McCaler's Complicity Cycle metrics, for the estimation. Since we use a formula $Comp_t = CompCF_t \times CompD_t$ to estimate $Comp_t$, we create two questionnaires, one for $CompCF_t$ (Control flow), the other one for $CompD_t$ (test data). Some simple questions in the questionnaire we provide to developers are as follows:

For $CompCF_t$ (Control flow):

1. How many lines of code are involved?
2. Are there many programming tips used?
3. Is the code straight-forward (seldom use statements such as if-then-else, for loop, etc.)?

For $CompD_t$ (test data):

4. Does the test data require extra processing?

Similar to the previous questionnaire, for each question in the questionnaire, we provide multiple choices with associated points. The choices with points for the example questions are as follows:

1. How many lines of code are involved?

   >1000 – 2 points 300~1000 – 1 points <300 – 0 points

2. Are there many programming tips used?

   Many – 2 points Some – 1 points Seldom – 0 points

3. Is the code straight-forward (seldom use statements such as if-then-else, for loop, etc.)?

   No – 2 points Somehow – 1 points Yes – 0 points

4. Does the test data require extra processing?

   Yes – 2 points No – 0 points

These are only example questions. Developers can always modify or add questions to form their own questionnaires. When answering the two questionnaires, developers score each test case based on their answers similarly to what testers do. Once all test cases have been scored based on both questionnaires, we assess $CompCF_t$ and $CompD_t$ as follows:

- For test cases in the top 20% of scores, $CompCF_t/CompD_t = 5$
- For test cases scoring between the top 20% and 40%, $CompCF_t/CompD_t = 4$
- For test cases scoring between the top 40% and 60%, $CompCF_t/CompD_t = 3$
- For test cases scoring between the top 60% and 80%, $CompCF_t/CompD_t = 2$
- For test cases in the bottom 20% of scores, $CompCF_t/CompD_t = 1$.

For each test case, we calculate its $Comp_t$ using the formula presented above. CV_t is established based on $Comp_t$ in the following way:

- For test cases in the top 20% of scores, $CV_t = 5$
- For test cases scoring between the top 20% and 40%, $CV_t = 4$
- For test cases scoring between the top 40% and 60%, $CV_t = 3$
- For test cases scoring between the top 60% and 80%, $CV_t = 2$
- For test cases in the bottom 20% of scores, $CV_t = 1$.

Table 5-2 is part of the data from our case studies, e.g., the $Comp_t$ for test case t0010 is 12, which falls between the top 20% and 40% of scores. Thus, its CV_t is assigned to be 4.

Test Case	$CompCF_t$	$CompD_t(d)$	$Comp_t = CompCF_t \times CompD_t$	CV_t
t0010	3	4	12	4
t0020	5	3	15	5
t0030	2	5	10	4
t0040	2	1	2	1
t0050	4	2	8	3
t0060	4	5	20	5
t0070	1	4	4	2
.....	...	...	...	...

Table 5-2: CV of some actual test cases

5.3.1.3 Calculate test case cost estimates (C_t)

Having assessed both CC_t and CV_t, we take the average of CC_t and CV_t as C_t. Since both CC_t and CV_t are between one and five, the *cost* C_t is also in a one to five range. Again, other organizations may choose to weigh customer cost or vendor cost differently than use depending on organizational priorities.

The *cost* estimates for some of the test cases in our case studies are shown in Table 5-3.

Now that a cost estimate has been derived for each test case, we can derive a failure probability enhanced by severity weights, called *severity probability*.

Test Case	CC_t	CV_t	C_t
t0010	5	4	4.5
t0020	3	5	4
t0030	2	4	3
t0040	4	1	2.5
t0050	1	3	2
t0060	2	5	3.5
t0070	1	2	1.5
...	.....	...	.....

Table 5-3: Cost of a few real test cases

In our method, *cost* of each test case is estimated only once. But testers can always change its value if it is necessary to revise the assessment.

5.3.2 Derive severity probability for each test case (Step 2)

The *Fault Probability* of the risk model that we presented in Chapter 3 is the probability that a fault will occur. As we discussed in section 3.5, the more defects detected, the more defects we can expect [Myers79]. To estimate *probability*, we can make good use of the number of defects detected by each test case. This information exists in the test profile of the *full testing* data. After running the *full test suite*, we can sum up the number of defects uncovered by each test case to estimate *probability*.

Based on our experience with real, in-house testing, we found that the severity of defects is just as important as the number of defects for measuring software quality. *Severity* is commonly used in industry to describe how important or serious a defect is. When a defect is opened (observed and documented), the tester assigns a *severity* estimate of the problem from both the customer perspective and the vendor perspective. In our case studies, we used four levels of the *severity*. The definitions are as follows, with Severity four defects considered the highest priority:

- Severity Four: 1) Defects that cause wrong results or failures that are critical to the system under test. 2) Defects that prevent a major portion of test or development from proceeding. These defects need to be fixed immediately.
- Severity Three: 1) Defects that cause wrong results, though the results are not critical to the system under test. 2) Defects that restrict test or development processes, and there is no acceptable circumvention. They need to be fixed in a short time.
- Severity Two: 1) Defects that cause unexpected behaviors in the system under test. 2) In spite of the defects, test or development is still able to proceed, but only with limited functions that are not crucial, possessing and acceptable circumvention (workarounds). These defects should be fixed, but the targeted fix date is negotiable.
- Severity One: 1) Defects that do not really cause a failure, but are related to

enhancement request. 2) Defects that do no have immediate impact to test or development processes. When time is available, these defects should be fixed to improve quality of the system under test.

To take severity into account, we modify the simple risk model that we proposed in Chapter 3. The *probability* element in the original risk model is changed to *severity probability*, which combines the average severity of defects with the *probability* of execution. Based on multiplying the Number of Defects N_t by the Average Severity of Defects S_t ($N_t \times S_t$), we can estimate the *severity probability* P_t for each test case t.

Severity probability again is normalized to fall into a one to five scale, where one is low and five is high. Since it is still possible for any test case without any defect to fail in the future, we set its P_t to one instead of zero. The assessment of *severity probability* obeys the following rules:

- For test cases in the top 25% of scores, $P_t = 5$
- For test cases scoring between the top 25% and 50%, $P_t = 4$
- For test cases scoring between the top 50% and 75%, $P_t = 3$
- For test cases in the bottom 25% of scores, $P_t = 2$
- Test cases without any defect in the *full testing*, $P_t = 1$.

Table 5-4 displays P_t for some test cases in our case studies.

We believe that using *severity probability* instead of *probability* in our risk mode is important, because *severity probability* weighs the test cases that had serious defects in previous test execution. The areas covered by these test cases probably contain more defects based on clustering of defects.

Severity probability is calculated based on *test profiles* [Section 4.4.1]. The value of *severity probability* will change after new test execution. But testers are not allowed to change the value manually.

Test Case	Number of Defects (N_t)	Average Severity of Defects (S_t)	$N_t \times S_t$	P_t
t0010	1	2	2	2
t0020	1	3	3	3
t0030	1	1	1	2
t0040	1	4	4	3
t0050	2	3	6	4
t0060	3	3.5	10.5	5
t0070	0	0	0	1
...	...	...	...	...

Table 5-4: Severity probability of test cases

5.3.3 Calculate Risk Exposure for each test case (Step 3)

Now we can compute the degree of *Risk Exposure* RE_t for each test case. Using column C_t in Table 5-3 and column P_t in Table 5-4, we calculate *Risk Exposure* RE_t for every test case to yield Table 5-5.

The *Risk Exposure* RE_t is our basis for choosing test cases as *Safety Tests*. To enhance or focus the results, we can also add weights to test cases that we must give preference to. For example, we might like to choose more test cases for some corporate flagship features, such as database features, if we are a database systems provider.

Test Case	C_t	P_t	$RE_t = C_t \times P_t$
t0010	4.5	2	**9**
t0020	4	3	**12**
t0030	3	2	**6**
t0040	2.5	3	**7.5**
t0050	2	4	**8**
t0060	3.5	5	**17.5**
t0070	1.5	1	**1.5**
...	.....	...	...

Table 5-5: Risk Exposure RE_t of test cases

From the above discussion, we can see that the *Risk Exposure* is affected by both *cost* and *severity probability*. We assess *cost* when preparing test plan and design test cases (minimizes overhead). *Severity probability* is estimated from *test profiles* (records of previous executions).

In the next section, we select *Safety Tests* to cover the overall *Risk Exposure* for our product consisting of components and features.

5.3.4 Select regular test cases as Safety Tests (Step 4)

In this section, we provide our approach of selecting feature-oriented, regular test cases as *Safety Tests*. A regular test case is often designed to test a specific local feature.

To get "good enough quality" [Bach 98], our regression test suite should achieve

some coverage targets. These coverage targets have to be set up while respecting the available test time and budget. Intuitively, we do not want to choose low *risk* test cases, such as t0070, because the resources used to run t0070 could be used more profitably to run other test cases. Therefore, we choose test cases that have the highest value of RE_t from the *non-Targeted Tests* to reach our coverage targets (*Risk Exposure* and modified functionality) by supplementing *Targeted Tests* with these *Safety Tests*.

In our case studies, we selected 30% of the *full test suite* to form regression test suite. 30% was selected to make our regression test suite comparable to the manually selected regression test suite of historical data that had 30% coverage. Table 5-6 is part of the test suite. One means the test cases is selected and zero means the test case is omitted.

Test Case	Full Test Suite	Regression Test Suite (30%)
t0010	1	1
t0020	1	1
t0030	1	0
t0040	1	0
t0050	1	1
t0060	1	1
t0070	1	0
...	...	...

Table 5-6: Test case selected

At this stage, we could stop since, as Chapter 6 will show, we have systematically achieved a good level (better than the manual level) of regression coverage with the combined *Targeted Tests* and *Safety Tests*. However, if we also have end-to-end scenario tests in our initial test suite, we show in the next section how to enhance *risk* coverage by

selecting some scenario test cases as well.

5.4 Risk-based End-to-end Test Scenario Selection

Industrial regression testers also utilize user-oriented, end-to-end scenarios. End-to-end scenarios simulate common user profiles of system use. Often they are designed based on use cases documented while capturing requirements or early in the system design phase. Some of them are equivalent to use cases. Compared to regular test cases, end-to-end scenarios are more customer-directed.

Since end-to-end scenarios involve many components of the system working together, they are highly effective at finding regression faults. When test time is limited, we might be not able to run all end-to-end scenarios for the system. Thus, to reduce *risk* efficiently, we give the following two rules for scenario selection:

R1: Select scenarios that test the most critical *requirement attributes*.

R2: Have the scenario tests cover as many *requirement attributes* as possible.

An end-to-end scenario runs across several components of the system under test. Often it is a serialized sequence of regular test cases for these components. Since every regular test case tests one or more *requirement attributes*, by the *requirements traceability* (see Section 4.3), the rules shown above are equivalent to:

T1: Select scenarios that contain the most critical regular test cases.

T2: Have the test suite of scenarios cover as many regular test cases as possible.

This is very similar to G. Myers "High-Yield" test design strategy [Myers 79] and the work by Saleh et al on "High-Yield" testing [Saleh+02].

To be brief, in this section, we use term "scenario" for "scenario test cases", and "test case" for "regular test case".

5.4.1 Scenarios and coverage of test cases

Before doing scenario selection, for every end-to-end scenario, we have to know which test cases are covered by each scenario. Since one scenario is a sequence of test cases, we keep this information in a table, such as Table 5-7.

To determine the relationships between scenarios and test cases, in our research, we re-organize the information in Table 5-7, and create a traceability matrix to trace these relationships.

Scenario	Test Case
s001	t0010, t0020, t0030, t0040, t0050, ...
s002	t0030, t0050, t0060, t0070, ...
s003	t0040, t0070, ...
...	...

Table 5-7: Scenarios and coverage of test cases

For each scenario, we mark all test cases that are covered by this scenario with a one, and the rest of the test cases (un-covered) are marked with a zero. Table 5-8 is an example of a traceability matrix corresponding to Table 5-7.

Test Case	Scenario			
	s001	s002	s003	...
t0010	1	0	0	...
t0020	1	0	0	...
t0030	1	1	0	...
t0040	1	0	1	...
t0050	1	1	0	...
t0060	0	1	0	...
t0070	0	1	1	...
...	...	...	...	...

Table 5-8: Traceability between test cases and scenarios

(0 -- The scenario does not cover the test case, 1 -- The scenario covers the test case)

We can now adapt the *Risk Exposure* model-based test selection method to end-to-end scenarios.

5.4.2 Risk Exposure model-based end-to-end scenario selection method

There are also four main steps for end-to-end scenario test selection in our approach:

End-to-end Safety Test Scenarios Selection Method:

Step 1. Calculate *Risk Exposure* RE_s for each scenario. (Subscript s refers to scenarios.)

Step 2. Select the scenario that has the highest RE_s for regression testing.

Step 3. Update the traceability matrix and re-build table of RE_s.

Step 4. Repeat Step 2 and Step 3 until we run out of time or resources, or better yet, we achieve a *risk* coverage objective.

The regression scenario test selection process can be described as in Figure 5-1.

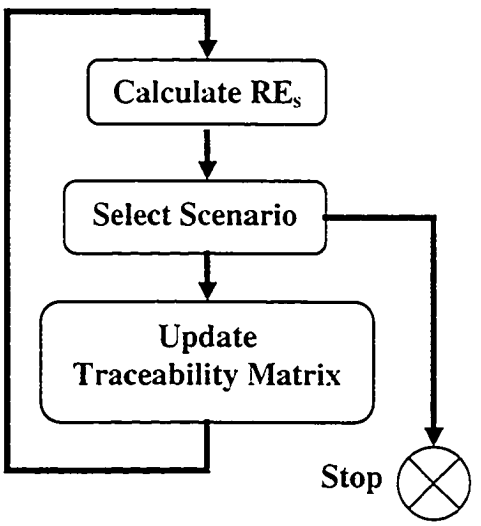


Figure 5-1: End-to-end Safety Test Scenarios Selection Method

We will use the same example to illustrate the steps in our end-to-end scenario selection approach.

5.4.2.1 Calculate *Risk Exposure* for each scenario (Step 1)

In Table 5-5 we show the *Risk Exposure* RE_t for each test case (see section 5.3.3). Since scenarios consist of test cases, we can simply calculate the *Risk Exposure* for each scenario RE_s by summing up the *Risk Exposure* RE_t of all test cases that this scenario covers. If a test suite consists of n test cases, the formula for RE_s is:

$$RE_s = \sum RE_{ti}, \{1 \leq i \leq n \mid \text{test case } t_i \text{ is covered by this scenario}\}$$

Based on the information in Table 5-8, and applying the formula to our example, we calculate *Risk Exposure* RE_s for all scenarios as shown in Table 5-9.

Scenario	RE_s
s001	985
s002	463
s003	732
s004	213
s005	195
s006	127
s007	70
...	...

Table 5-9: Risk Exposure RE_s for scenarios

5.4.2.2 Select scenarios that has the highest RE_s (Step 2)

The scenario with the highest RE_s covers the most critical test cases, or has the highest coverage of risk severity-covering test cases. According to our selection rules, it should be included in the regression test suite. In our example, scenario s001 has the highest RE_s in Table 5-9. Therefore, it is put first into the regression scenario test suite.

Now, we need to reduce RE_t for all test cases that are covered by this scenario and look for the next high RE-covering scenario.

5.4.2.3 Update the traceability matrix and re-build RE_s table (Step 3)

Because high RE-covering scenario has been included, all test cases covered by that scenario will be executed. According to our selection rules, these test cases should not affect our selection any more. We need to focus on those test cases that have not yet been covered.

In our approach, after the chosen scenario has been executed, we delete the column for the chosen scenario and rows for all the test cases that have been covered by this scenario from the traceability matrix, which is Table 5-8 in our example. This process is shown in Table 5-10. In this table, because we have put scenario s001 into the regression test suite, we cross out the column for scenario s001, and rows for test cases t0010, t0020, t0030, t0040, and t0050, which are covered by scenario s001.

Simplifying Table 5-10, we get Table 5-11, as the result of this step.

Test Case	Scenario			
	~~s001~~	s002	s003	...
~~t0010~~	~~1~~	~~0~~	~~0~~	...
~~t0020~~	~~1~~	~~0~~	~~0~~	...
~~t0030~~	~~1~~	~~1~~	~~0~~	...
~~t0040~~	~~1~~	~~0~~	~~1~~	...
~~t0050~~	~~1~~	~~1~~	~~0~~	...
t0060	~~0~~	1	0	...
t0070	~~0~~	1	1	...
...	...	...	...	...

Table 5-10: Process of updating traceability matrix

(0 -- The scenario does not cover the test case, 1 -- The scenario covers the test case)

Test Case	Scenario		
	s002	s003	...
t0060	1	0	...
t0070	1	1	...
...	...	...	...

Table 5-11: Updated traceability between test cases and scenarios (Updated version of Table 5-8)

(0 -- The scenario does not cover the test case, 1 -- The scenario covers the test case)

Based on Table 5-11, we can calculate RE_s again for the remaining scenarios and re-build Table 5-9 yielding Table 5-12.

Scenario	RE_s
s002	356
s003	611
s004	176
s005	180
s006	96
s007	68
...	...

Table 5-12: Revised Risk Exposures for scenarios (Updated version of Table 5-9)

5.4.2.4 Repeat Step 2 and Step 3 as necessary and possible

In Table 5-12, scenario s003 is the one with the highest RE_s and is our next choice for regression testing. After adding s003 to our regression scenario test suite, we repeat the process (Step 2 and Step 3) until it is time to stop.

The size of our test suite of scenarios can be dependent on the time and resources available. Regression test selection is terminated whenever we run out of time or resources. Each time we select one test scenario, the *Risk Exposure* for un-selected scenarios are decreased. Therefore, we can also decide when to stop choosing more regression test scenarios based on the Pareto principle (also known as the *80:20 Rule* and

the *law of the vital few*, which states that for many phenomena 80% of consequences stem from 20% of the causes) [Kan 02], or based on whether we have reached a pre-defined *Risk Exposure* range of coverage.

As a related observation, for large systems, scenarios are often executed in a manual mode. In this case, retesting all scenarios is not a cost-effective regression strategy, and our optimization-based selection method for regression scenario becomes very useful and important.

The final selected regression test suite is the union of the *Targeted Tests* (see Chapter 4) and the *Safety Tests* (see Chapter 5).

5.5 Chapter Summary

This chapter discussed selecting the complementary second category of our regression test cases, which is *Safety Tests*. We provided methods of choosing both test cases and end-to-end scenarios for the *Safety Tests* and gave an example. The methods we addressed were based on *Risk Analysis*.

This completes the presentation of our regression testing approach including both *Targeted Tests* selection (in Chapter 4) and *Safety Tests* selection (in this chapter). Our final regression test suite is the union of the *Targeted Tests* and the *Safety Tests*.

We now evaluate the quality of our approach to regression test case selection.

Chapter 6. Industrial Case Study with Evaluation and Discussion

6.1 Introduction

In previous chapters, we presented our *specification-based* regression testing technique. To evaluate our approach, we applied it to choose regression test suites for some components of a real software product. Part of the data of our case studies has been used as examples in previous chapters.

Based on the results of the case studies, we perform a serious of analyses to examine the performance of our methods. First, in Section 6.2, we define a matrix to quantify the results, collect corresponding measurements (targeted data) for both our approach and the manual approach (historical data), and then analyze and compare the results with respect to the matrix. In Section 6.3, we apply a commonly used evaluation framework from the scientific literature to evaluate our technique. Also, in Section 6.4, we list some other advantages of our methods. In Section 6.5, we use a statistical analysis tool, named SPSS, to create statistical reports. Finally, in Section 6.6, we discuss the limitations of our approach and give recommendations of applying our approach.

Since the concept of scenario used in IBM ECD is different to the concept we used in this research, we cannot compare the results of our end-to-end test scenario selection approach to historical data. The comparison will be only applied to test case selection.

6.2 Case Studies: Overview and Initial Analyses

To assess the validity and efficiency of our approach, we chose three components from an industrial software product as three case studies. Each component was owned by one experienced software engineer. For each component, 5 to 10 *activity diagrams* were included in the design document. In total, there were 306 test cases in the original full test suite created based on these *activity diagrams*.

In Appendix B we show an activity diagram chosen from the design documents. This is an activity diagram for Pricing and Shipping feature, which is our case study one. In the design document of this feature, there are seven activity diagrams in total. Because of copyright and non-disclosure, we changed some names of activities, and hid all guard conditions in this diagram.

6.2.1 Specification of data sets

In our case studies, the three components have different characteristics.

- Case study One: Pricing and Shipping feature (Component PCSP)

This feature provides functions that calculate the total amount of products, and deal with chosen shipping services. Functionality is the main focus of testing. There were seven activity diagrams drawn for this component. We designed 79 test cases for full test. Only one or two test cases take a specific path.

- Case study Three: Payment and Credit Management feature (Component PCMT)

To provide convenience to the customers, payment and credit management

feature has to handle all possible payment methods. In the software product, there are more than 15 payment methods. Data is very important in this component. Sometime it plays an even more important role than functionality in testing. There were ten activity diagrams, and 112 test cases. For some functions, we have to create one or more test cases for each payment method. For example, we have 48 test cases for one specific path.

- Case study Three: Contact and Account Management feature (Component CAMT)

This component deals with contact and account. Since the management of contact and account is based on their status, for this component, requirement analyzers and system architectures used UML *state diagram* instead of *the activity diagram* to capture requirements and create design models. There were three state diagrams, and 115 test cases were designed. Both functionality and data are important for this feature. We have up to 25 test cases that take the same path.

6.2.2 Case study methodology and summary of results

Our case studies take four steps applied to each of the three components:

Step 1. Run the full test suite and get the *test profile*. We define this test profile as the *first pass test profile*. 65 defects were opened for all components in the *first pass*.

Step 2. Apply our regression test selection approach to select *risk-based* regression test suites based on the *first pass test profile*. At the same time, the three experienced software engineers manually chose their regression test suites for the components they own. We call these *manual regression test suites*.

Step 3. Rerun the full test suite on the modified system after the developers fix defects. This step would not normally be performed; however, we do it to get the total number of defects in the three components for our comparison. We call this run the *second pass*. There were ten defects found in the *second pass*, which caused 28 test cases to fail.

Step 4. Create reports to measure the quality of both *risk-based* regression test suites and manual regression test suites based on the *test profile* obtained in the *second pass*.

As the result of our case studies, we generated a series of reports (in Step 4 above). In later sections, we use these reports to evaluate our approach. One example is Table 6-1.

	Number of Defects in the Second Pass	Risk-based Regression Test Suite		Manual Regression Test Suite	
		Defects Detected	Omitted Defect-revealing Test Cases	Defects Detected	Omitted Defect-revealing Test Cases
Component PCSP	3	3	1	2	4
Component PCMT	7	7	0	6	1
Component CAMT	0	0	N/A	0	N/A
Total	10	10	1	8	5

Table 6-1: Report of detected defects and omitted defect-revealing test cases

Both *risk-base* regression test suite and manual regression test suite cover 30% test cases of the *full test suite*. The second column is the number of defects existing in each component. During test execution, each defect fails one or several test cases. These test cases are *defect-revealing* test cases of this particular defect. If at least one *defect-revealing* test case for a defect is selected, the regression test suite is able to detect this defect.

For both the *risk-based* regression test suites and the manual regression test suites, Table 6-1 records the number of defects that can be detected, and the number of *defect-revealing* test cases that are omitted. Results in Table 6-1 show that our *risk-based* regression test suites are able to detect all ten defects, while the manual regression test suites miss two defects. Thus, our method is more powerful in finding defects. Even though both techniques omit some *defect-revealing* test cases, we missed less than the three software engineers did. Therefore, our method is more sensitive to *defect-revealing* test cases. We discuss this in more detail in the next section.

6.3 Evaluation According to Published Standard

After examining regression test patterns listed in section 2.3, we find that our approach is a combination of pattern *Retest risky use cases* and pattern *Retest by profile*, with concepts borrowed from pattern *Retest changed segments*.

Rothermel and Harrold presented a standard framework for analyzing regression test selection techniques in [Rothermel+96], which has been used widely to evaluation regression test methods. This framework consists of four categories: inclusiveness, precision, efficiency, and generality. We now apply this framework to evaluate our approach.

6.3.1 A standard framework for evaluation of regression test selection

In this section we introduce Rothermel and Harrold's widely accepted framework briefly. Detailed discussion about this framework is out of the range of this thesis, and can be found in [Rothermel+96].

1. *Inclusiveness*: the extent to which a regression test selection technique chooses test cases that may reveal regression faults. It is the percentage of baseline tests that may show regression faults and that are selected for a reduced test suite.

   For example, if the full test suite T has 50 test cases of which eight will manifest regression failures, and if a regression test selection technique M selects two of these eight test cases, then M is 25% inclusive. Note that the measure of inclusiveness is not relevant to the number of test cases in T, but to the number of defect-revealing test cases. The more defect-revealing test cases selected by M, the better regression testing technique M is. Omission of defect-revealing test case is very serious. Some regression defects may not be detected because of the omission.

2. *Precision*: the extent to which a regression test selection technique omits test cases that are not able to reveal regression faults. It is the percentage of baseline test cases in a reduced test suite that cannot reveal regression faults and that are not selected for the reduced test suite.

   For example, if the full test suite T has 50 test cases of which 44 are not able to reveal regression faults, and a regression test selection technique M omits 33 of these 44 test cases, then M is 75% precise.

   A 100% precise regression suite does not include any test cases that cannot reveal regression fault, because these test cases can be safely removed.

3. *Efficiency*: the cost of identifying a reduced regression test suite, including the space and time requirements for the regression test selection technique. The higher cost that a regression testing technique M spends, the less efficient M is.
4. *Generality*: the ability of the regression test selection technique to function in a wide and practical range of situations. It indicates the range of application for the selection strategy.

6.3.2 Analysis within the framework

In this section, we use the framework we introduced in the previous section to analyze our regression test selection approach.

- Inclusiveness:

  From the discussion in previous chapters, it is obvious that only test cases that test affected entities may reveal regression faults. Our *Targeted Tests* strategy focuses on affected entities. If all changes in the system are completely documented, and the relationships between components of the system are correctly identified, the *Targeted Tests* will include all test cases that test affected entities. Since regression faults are induced by affected entities [Rothermel+96], all test cases that may reveal regression faults will be included in the *Targeted Tests*. Hence, the inclusiveness should be one hundred percent.

  However, in practice, documentation is often out-of-date, and it is very difficult to identify the relationships between components completely and correctly because of the complexity of the system under test. Also, our approach is *specification-based* and we do not access the code. Therefore,

we may omit or overlook some test cases that may reveal regression faults. As shown in Table 6-1, our *risk-based* regression test suites omit one *defect-revealing* test case for Component PCSP.

Compared to the manual regression test suites, our *risk-based* regression test suites have higher quality even though they are not 100% inclusive. We omitted fewer *defect-revealing* test cases (one test case) and our test suites detected all defects for the three components, while the manual regression test suites omitted more *defect-revealing* test cases (five test cases) and only detected eight defects out of ten. A good regression test suite should be able to find all ten defects. Thus, in our case studies, our approach was more effective in finding defects, and we chose effective test suites for regression testing.

- Precision:

In the part of *Targeted Tests* selection of our approach, we require testers to select all test cases that test changed components. These test cases can be called as *modification-traversing* test cases [Rothermel+96]. Since our method is a *black box* or *grey box* technique, some *modification-traversing* test cases may be omitted because of out-of-date or incomplete documentation. Therefore, we use a *Safety Tests* selection strategy to choose additional test cases beyond the *Targeted Tests*. Some *non-modification-traversing* test cases will be picked up as regression test cases. Therefore, some tests that could have been skipped may be repeated.

- Efficiency:

The time and cost associated with testing are always limited. Since selection is based on the *activity diagrams* and *test profile* that are not implementation dependent, regression analysis can be done without code analyzers or in-depth technical knowledge of the system under test. If the requirement traceability matrix and the *test profile* have already been derived,

the time or cost of the regression analysis will be relatively low.

Compared to manual selection, our approach is more objective and systematic. The selections of the three software engineers in our case studies were subjective. The current selection process is based on experience and thus has to be done manually. Our new approach can guide regression test selection, particularly benefiting new or less-experienced test personnel.

- Generality:

As a *specification-based* technique, our method can be applied in a wide variety of development contexts and process methodologies. Also, since it is *specification-based* and is able to be automated, our method can be applied to large-scale systems. This is a big advantage over other code-based approach, such as Rothermel's approach.

In additional to *activity diagrams*, our approach can be applied to some other recent notions that have been widely used in industry, such as UML *state diagrams*. For example, for the *Targeted Tests*, based on any graph-like diagrams, we can obtain requirement traceability metrics, apply the traverse algorithm and conduct regression analysis similarly. Also, our *Safety Tests* selection is based on the *test profile*, and is not dependent on technologies used for requirements capturing or system design.

In our case studies, there was one component, namely CAMT, which was designed, implemented, and tested based on a *state diagram* model. For that component, every test case was selected to exercise a path consisting of states and transitions of the *state diagrams*. We can label the *state diagram* and apply our approach, just as we did for *activity diagrams*.

Generally speaking, our approach should apply to any kind of diagrams that have paths for test cases to exercise.

6.4 Additional Advantages of Our Approach

Beyond what we have discussed in the previous section, our technique has some other advantages summarized below:

1. Our approach is systematic, and therefore relatively objective.

   When we presented our method in previous chapters, we had always listed the steps as guidelines for the selection process. Based on the guidelines, testers know what need to be done in this step and what the next step is. Therefore, our method is systematic and less subjective (and more efficient as described in the previous section).

2. Our approach is sensitive to *risk*, and therefore is customer-oriented in setting priorities for regression tests.

   Our approach is *risk-based*. Therefore, we are able to use risk metrics to quantitatively measure the safety of test suites (see section 2.5). Table 6-2 is the risk coverage matrix we obtained from our case studies.

	Risk-based Regression Test Suite	**Manual Regression Test Suite**
RE_t Coverage (%)	52.60%	30.46%
Average RE_t	16.20	10.43

Table 6-2: Risk coverage report

Both our *risk-based* regression test suite and the manual regression test suite choose 30% test cases from the *full test suite*. As we can see in the matrix, *Risk Exposure* coverage and Average *Risk Exposure* of our regression test suites are much higher than those selected manually by the software engineers, which means our test suites cover more critical test cases. This means that we select test cases that have or higher cost or higher failure probability.

3. This approach is easy to automate as can be seen by the following arguments, and by our experience

The procedure of our approach involves only straightforward calculations, which are easily carried out by simple spreadsheet software. The necessary data are easy to be collected as well. We can implement the method in test tools to select test cases automatically. In this case, the time of test case selection is very short. Compared to the time and resources of re-running all test cases, our method is cost-effective. In our case studies, running all test cases for the whole product required several days, and regression testing needed to be done daily on every new build of the system. Applying our method to this large system, significant savings occurred in time and resources.

Necessary data for *risk-based* regression test selection can be summed up as follows:

- Traceability matrix between test cases and the *activity diagrams*

  When testers design a test case, path of the *activity diagrams* that covered by this test case shows the test logic. Therefore, it is necessary to record the path in test plan. If the test plan is stored electrically, traceability matrix can be obtained from the information automatically.

- Relationship matrix between scenarios and test cases

  Scenarios consist of a set of test cases. If the testers design test

scenarios by reusing test cases, the information about which test cases are covered by a specific test scenario should be included in the test plan. Therefore, we can generate the relationship matrix automatically from the test plan database.

- Cost of test cases

  This is the most subjective part of our technique. The cost assessment process has to be done manually, and the results should be put into database for future use. This is a one-time task. Testers do not need to repeat cost assessment during regression testing.

- Defects with severity information

  Typically, this information exists in test tracking tools. It is easy to have the test tracking tools classify defects by test cases and calculate the average severity.

- Location of modification in the *activity diagrams*

  In our approach, the testers are required to identify locations of defects when the defects are opened, and the developers are required to record the locations of any modifications, including *correctiveness maintenance* and *progressive maintenance*. This data are kept in database for automation.

In our case studies, we collected necessary data manually, and built a spreadsheet using Microsoft Excel to process the data and select regression test cases. Using more powerful tools, the whole regression test case selection process including data collection can be automated. It was easily to analyze metrics from the data, as well as using a standard tool set, namely SPSS, as well.

4. Pre-defined coverage target of *risk-based* regression test suite

To start *risk-based* regression testing, we have to setting up a target of test suite coverage. If the target is too low, the regression test suite may miss some critical defects. Thus, an appropriate coverage target is essential to ensure "good-enough quality".

In our research, the testers used to choose 30 percent test cases of full test suite as regression test suite. This target was chosen subjectively based on experiences. We found that it was not appropriate in some cases. For example, there were many defects found in the *first pass* and there might be more than 30% test cases failed. Obviously, these test cases should be re-run in the *second pass*, which means the 30 percent target was not enough.

In our method, we can use the number of *Targeted Tests* as a guide of setting up appropriate coverage. If the number of *Targeted Tests* is high, we need to define a high coverage target, and if the number is low, a low coverage target will be enough.

6.5 Overall Statistical Analysis of *Targeted Tests* and *Risk-based* Tests

In software testing, we use metrics to guide testing process improvement. Statistical techniques and statistical analysis tools are useful to analyze measurement data, and compute and display in formatted metrics.

In our research, we select a commercial tool, named SPSS, for statistical analysis. SPSS is a widely used statistical analysis tool. It can take data from almost any type of files, generate decision-making information quickly using powerful statistics, and effectively present the results with high-quality tabular and graphical output, such as tabulated reports, charts and plots of distributions and trends, descriptive statistics, and

complex statistical analyses. Also, it can share the results with others using a variety of reporting methods, including secure Web publishing. All these empower the user to make smarter decisions more quickly by uncovering key facts, patterns and trends. Successful stories include HSBC Bank, Southwestern Bell, and Interview Service of America (ISA) [SPSS].

In our research, we generate several kinds of reports from the data of our case studies with SPSS. Discussion of three kinds of reports in the following sections includes:

- What is our goal (the goal)?
- What are the metrics we generate (the metrics)?
- How to apply the SPSS tool to collect data and create the metrics to assess achievement (the tool)?
- What can we see from the metrics (the results)?

6.5.1 Types of components tested

- The *goal*: to understand the distribution of the test cases.

  In our case studies, there were 306 test cases and 20 *activity diagrams* in total. Test cases were designed based on the *activity diagrams*. Each test case took a specific path of the *activity diagrams* from the start marker to the end marker. Some test cases covered more than one *activity diagram*, since some other *activity diagrams* were pre-requirements of the targeted *activity diagram*. As we said in section 6.2.1, the three components had different characteristics. Understand the distribution of the test cases can help us examine these characteristics.

  Given our experience of applying our approach to different types of

components, it appears that our approach is applicable to various types of components.

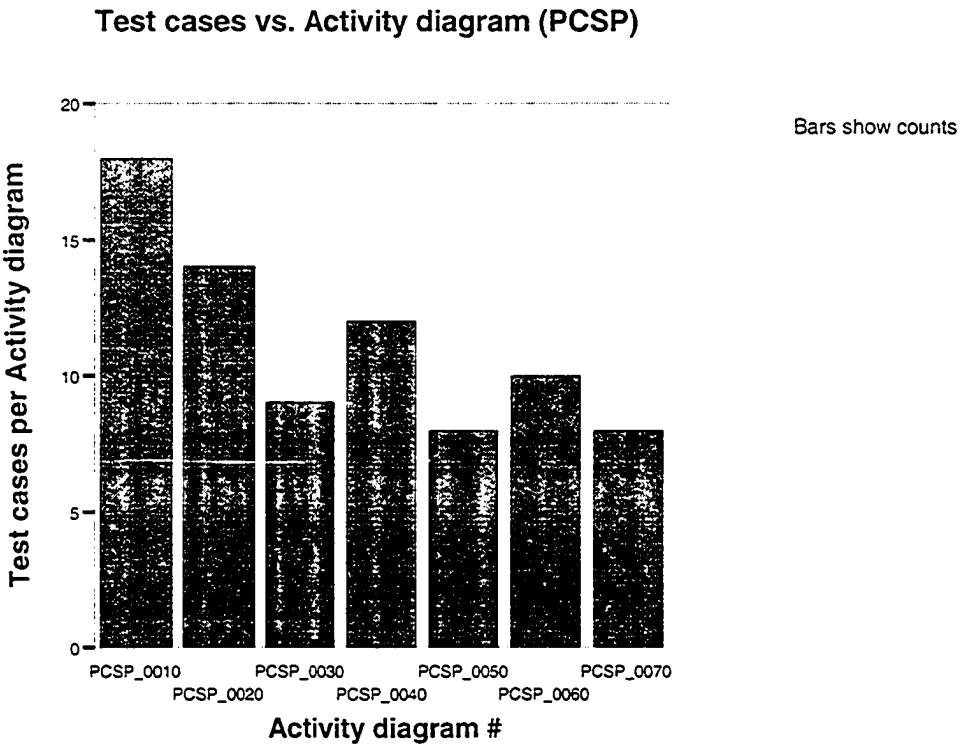
- The *metrics*: number of test cases per activity diagram

We use bar charts to display the relationship between the *activity diagrams* and the number of associated test cases. The bar charts are shown with horizontal axis indicating the *activity diagram* ids and the vertical axis showing the number of test cases designed for each *activity diagram*.

- The *tool*:

The inputs for the metrics are the *activity diagram* ids and test cases associated. With SPSS bar charting feature, we generated one bar chart for each case study as shown in Figure 6-1.

- The *results* (Figure 6-1):



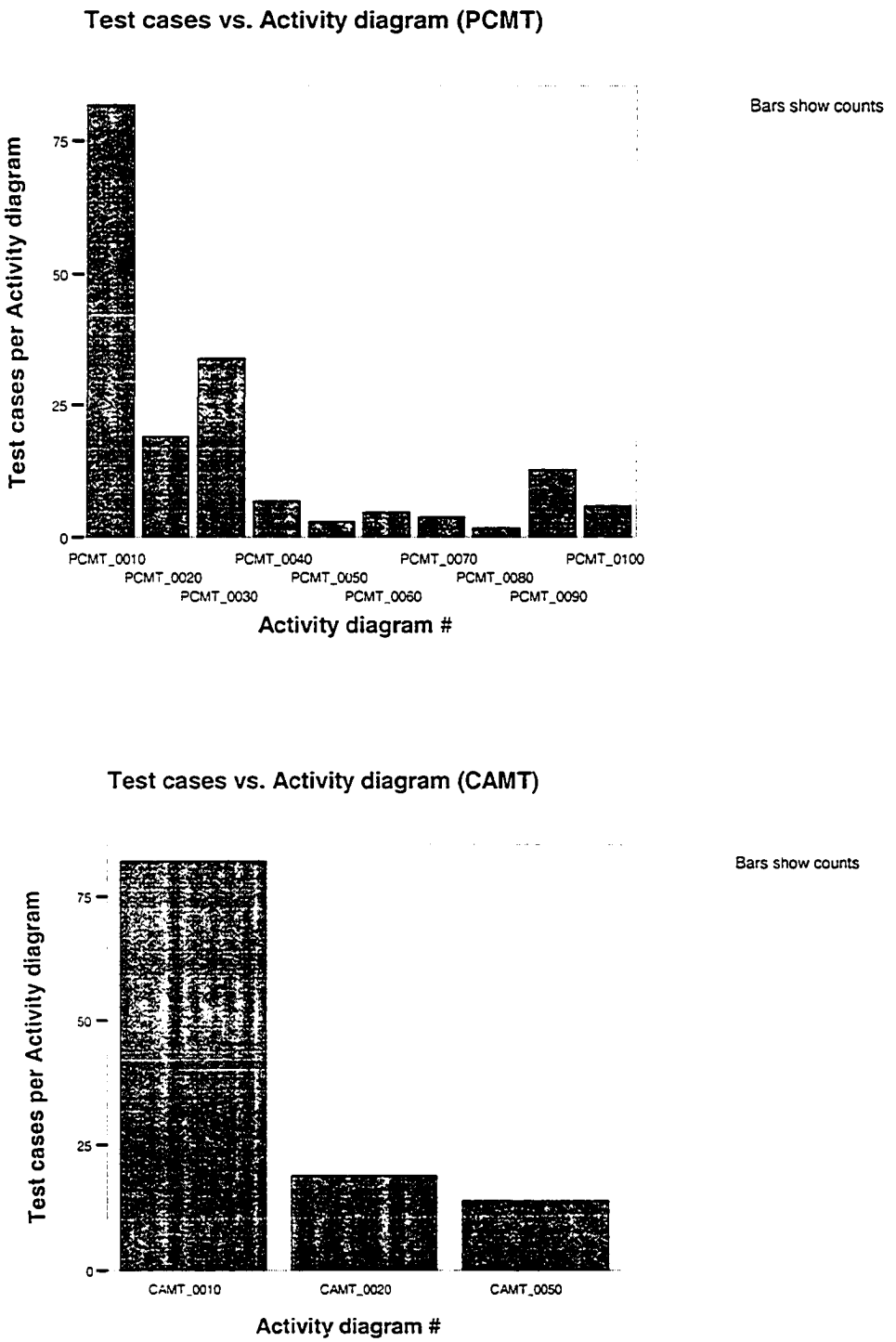


Figure 6-1: Bar charts of test cases distribution for three components

(1) PCSP, (2) PCMT, (3) CAMT

From these three bar charts, we found that component PCSP had the most balanced test case design, while component PCMT had the least balanced design.

The number of test cases designed for an *activity diagram* is dependent on the number of paths that need to be tested in this diagram and the number of associated data sets. As we said in section 6.2.1, testing of component PCSP focused on functionality, and only one or two data sets were associated with one test logic. Component PCMT dealt with payment. There were more than 15 payment methods exist in the system, which worked as data sets of test cases. Therefore, if payment methods need to be tested individually with a path, the number of test cases increases sharply.

6.5.2 Effectiveness and efficiency of Targeted Tests selection

- The *goal*: to approve that the *Targeted Tests* are powerful in finding defects.

  We usually consider the ratio of defects that can be detected by the selected test suite, and the ratio of selected *defect-revealing* test cases to measure the effectiveness and efficiency of test case selection. We hope to create metrics to show that the *Targeted Tests* detect high ratio of defects, and cover high ratio of *defect-revealing* test cases.

- The *metrics*: the ratio of detected defects and the ratio of selected *defect-revealing* test cases

  We create a tabulated summary report, which shows the two ratios for all three case studies, and pie charts, which describe the relationship between the *Targeted Tests* and *defect-revealing* test cases.

- The *tool*:

To generate the summary report, we need to identify number of defects that can be detected by the *Targeted Tests*. This number can be easy to get by recording all defect ids for defects opened by the *Targeted Tests* in the *second pass*. Also, we identify the number of test cases that belong to both the *Targeted Tests* and failed test cases in the *second pass*. This number is used to measure the ratio of selected *defect-revealing* test cases.

To create the pie charts, we separate all test cases into four categories and use pie charts to show the percentage of each category:

- TT & DRT: test cases that belong to the *Targeted Tests* and are *defect-revealing* test cases
- TT but Non-DRT: test cases that belong to the *Targeted Tests* but are not *defect-revealing* test cases
- Non-TT but DRT: test cases that are *defect-revealing* test cases but do not belong to the *Targeted Tests*
- Non-TT & Non-DRT: test cases that are not *defect-revealing* test cases and do not belong to the *Targeted Tests*.

- The *results*:

	Defects Detected (%)	**Defect-revealing Test Cases Covered (%)**
Component PCSP	100.00%	72.73%
Component PCMT	100.00%	100.00%
Component CAMT	N/A	N/A
Average	100.00%	90.91%

Table 6-3: Effectiveness and efficiency of Targeted Tests selection

Table 6-3 is the summary report for our case studies. Because there were no defects in component CAMT in the *second pass*, the cells are marked as N/A. As appeared in the report, our *Targeted Tests* were able to detect all defects and covered most *defect-revealing* test cases.

The pie chart for all three components is shown in Figure 6-2. Pie charts for individual component are listed in Appendix C. In this chart, we can see that most of the *defect-revealing* test cases are chosen as *Targeted Tests*. The reasons of omission can be incomplete documentation, or inaccurate identification of the relationship between *requirement attributes*.

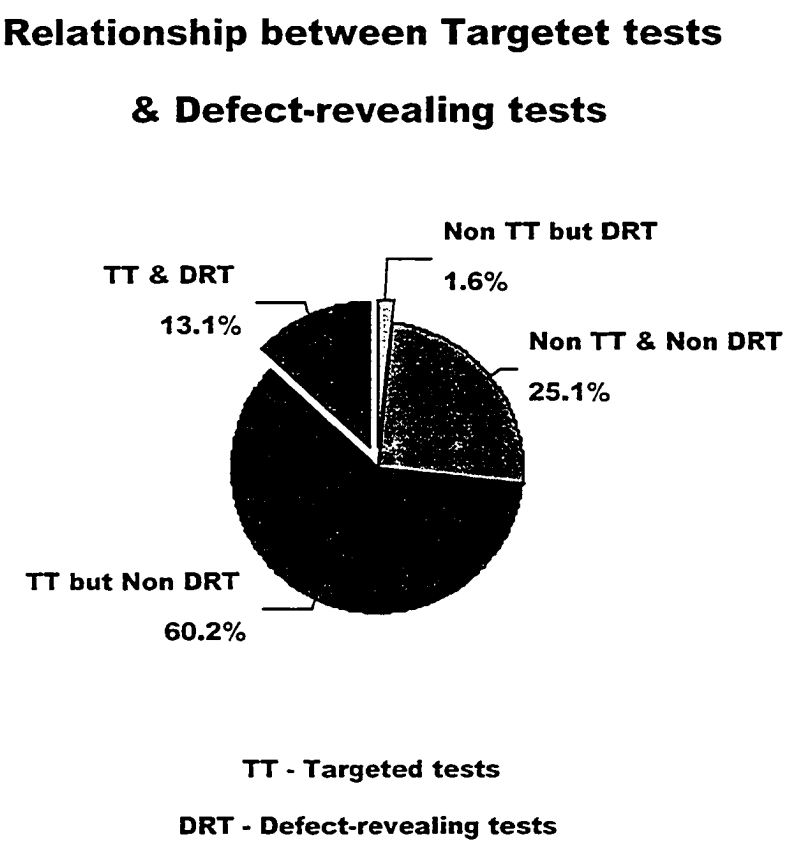


Figure 6-2: Pie chart of Targeted Tests and defect-revealing test cases

Many test cases of our *Targeted Tests* are not *defect-revealing* test cases. That is reasonable since test cases that test modified or affected *requirement attributes* are not necessary to be failed. One solution to decrease the percentage of TT but Non DRT is having more detail *activity diagrams* to separate test cases.

6.5.3 Effectiveness and efficiency of risk-based test case selection

- The *goal*: to approve that *risk-based* regression test suites are powerful in finding defects.

  Similar to the *Targeted Tests*, we use the ratio of defects detected and the ratio of *defect-revealing* test cases selected to measure the effectiveness and efficiency of *risk-based* test case selection. We hope to create matrix showing that most defects are detected by test cases with top *risk*, and most *defect-revealing* test cases are with top *risk*.

- The *metrics*: the ratio of detected defects and the ratio of selected *defect-revealing* test cases.

  We generate line charts to describe the pattern of defects detecting, and distribution of *defect-revealing* test cases. In the line charts, horizontal axes describe the coverage of *risk-based* regression test suites. The ratio of detected defects and selected *defect-revealing* test cases are indicated vertically.

- The *tool*:

  To generate the metrics, we select several *risk-based* regression test suites with increasing test suite coverage by 10%. For each test suite, we measure the ratio of defects detected and the ratio of *defect-revealing* test cases selected. We

identify the number of defects that can be detected by each *risk-based* regression test suite to measure the ratio of defects detected. Also, we identify the number of selected test cases that are failed test cases in the *second pass*. This number is used to measure the ratio of selected *defect-revealing* test cases. With line charting function of SPSS, we can generate the metrics.

- The *results*:

Figure 6-3 is the chart created for all three case studies. We also created line charts for each case study individually, which are listed in Appendix C.

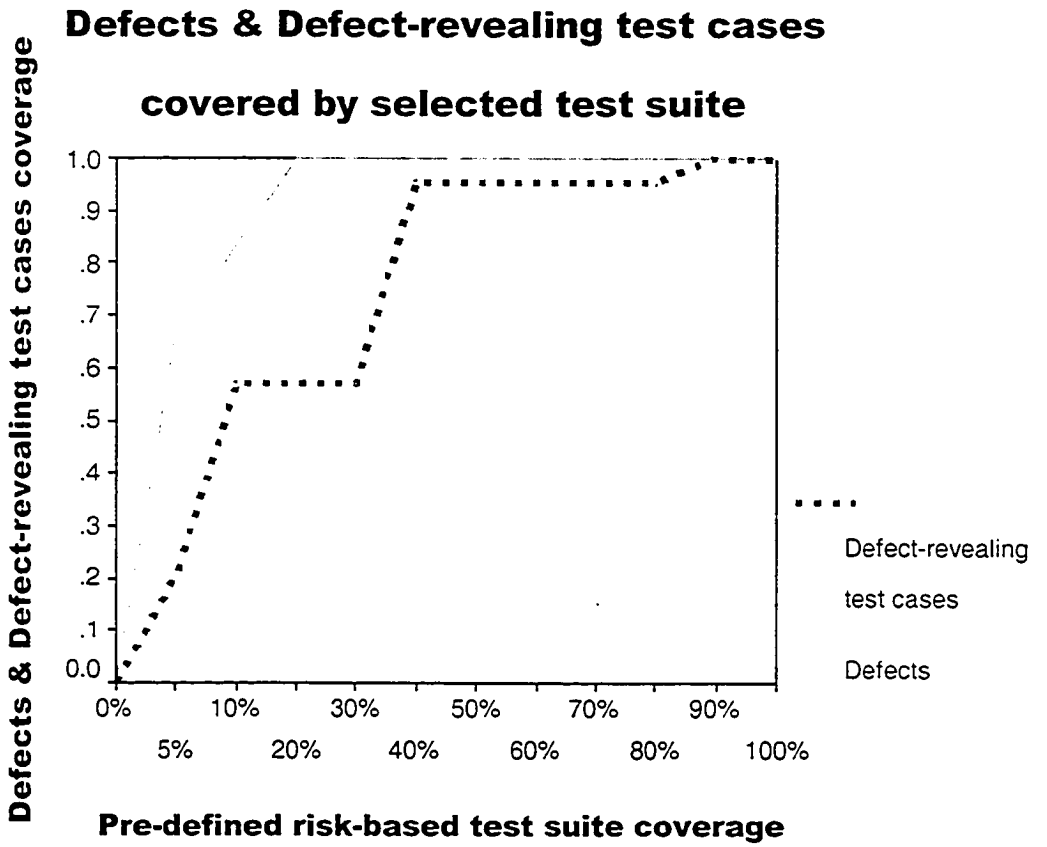


Figure 6-3: Effectiveness and efficiency of risk-base test case selection

In the line charts, we use solid line to indicate the ratio of detected defects, and dashed line for the ratio of selected *defect-revealing* test cases. Obviously, when the coverage of test suite increases, both ratios increase. The lines of both ratio increase sharply before the coverage of test suite reaches 40 percent. Indeed, 100% defects have been detected by test suite with 20% coverage, and more than 95% defect-revealing test cases are covered when the coverage reaches 40%. This result proves that *risk-based* test case selection strategy chooses the most critical test cases (revealing regression faults). Compare to manual regression test case selection (see the report in Section 6.2.2), it provides an effective and efficient way to selection regression test cases. Therefore, our *risk-based* regression testing technique seems to be cost-effective based on our three case studies.

6.6 Limitations and Recommendations

6.6.1 Limitations of the approach

Besides the advantages we discussed in previous sections, there are some limitations in our approach.

1. The correct *Targeted Tests* selection is based on completeness and correctness of documentation. If there is any error exists in the documentation, for example, modifications in either specification or code are not recorded, some *defect-revealing* test cases may be omitted. That is the reason for selecting the *Safety Tests*.
2. To do regression analysis for selecting the *Targeted Tests*, we must identify the

relationships between *requirement attributes* correctly and completely. It is difficult and error-prone even we are working on specification level.

3. In selecting the *Safety Tests*, *Cost* assessment is very important to calculate *Risk Exposure*. But it is also the most subjective part in our method. If the testers do not follow the steps correctly and carefully, or the testers do not understand how the customers use the system, they are not able to assess the *Cost* accurately. The inaccurate *Cost* will lead some problems. For example, many test cases may have same *Risk Exposure*. It is hard to tell which test cases should be chosen as the *Safety Tests*. In this case, the test suites chosen by our approach can only be suggestions. After applying our approach, the testers need to pick up test cases manually from the suggested test suite.

6.6.2 Observations and recommendations for applying the approach

Based on our experience, we compare our methods, including the *Targeted Tests* selection and the *risk-based* test case selection, with manual test case selection. Table 6-4 shows the results.

The manual selection strategy in our case studies focuses on specification coverage. The testers tend to select test cases that can increase specification coverage. Some other manual strategies may not have even this benefit.

Our approach focuses on defect-revealing test cases coverage instead of specification coverage. As we can see in this table, both the *Targeted Tests* selection and the *risk-based* test case selection seem to be more cost-effective except the specification coverage aspect. Therefore, our approach is powerful in finding *regression faults* with low analysis cost.

Approach	Cost	Effectiveness		
	Analysis Cost	**Defects Detected**	**Defect-revealing Test Cases Coverage**	**Specification Coverage***
Targeted	Medium	High/Medium**	High/ Medium	Medium
Risk-based (Depends on coverage)	Low	High	High	Medium
Manual	High	Medium	Medium	High

* Not studied in detail, but based on experience

** Apply in some cases (e.g. with incomplete documentation)

Table 6-4: Cost-effectiveness comparison

The discussion of limitations shows that complete and correct documentation are essential to ensure the effectiveness of our methods. But in practice, we cannot always get ideal documentation. Hence, in some cases, manual selection is needed as a backup. We recommendation is applying our approach to select test suites as suggestions, then choosing real regression test suites manually from them. Our justification is as follows:

- Our approach is powerful in detecting regression defects. The selected regression test suite covers high ratio of defect-revealing test cases.
- Our approach can be automated. Therefore, the regression analysis is done by computer and the cost of regression analysis is low.

6.7 Chapter Summary

In this chapter, we evaluated the regression test selection approach presented in previous chapters. Both benefits and limitations of our approach were addressed. Some components of a real software product were used as the case studies. The evaluation included the following actions:

- Create metrics to quantify the results of our case studies
- Compare the results obtained from our methods to historical data
- Evaluate our technique under a standard evaluation framework
- Address additional advantage of our approach
- Create statistical analysis reports.

Overall, the method seems to be satisfactory with respect to general quality measure, and superior to the manual approaches used in three case studies. In additional, sample commercial spreadsheet tools were effective for managing the select process. In our case studies, we used Microsoft Excel to process data and manage regression test case selection.

Chapter 7. Conclusions and Future Work

7.1 Summary

Throughout this thesis, we demonstrated the need for a systematic methodology that can be applied in practice to handle both requirement modifications and code modifications. We achieved this by detecting design and code changes and deriving the ripple effects based on the *activity diagram*. The problem of regression analysis and testing in practice is defined. Existing regression analysis and testing methodologies are discussed and criticized. A *specification-based* regression test selection technique is presented. This technique is customer-oriented and also *risk-based*. It provides methods to obtain both *Targeted Tests* and *Safety Tests*.

Chapter 2 and Chapter 3 provide the reader with background information related to *regression testing* and *risk-based* testing. The problems of regression analysis and testing were identified and related work was discussed. Chapter 4 and Chapter 5 described the details of the method with results from small but real industrial case studies in e-commerce development. The method provides a new objective regression test case selection criterion that has good potential to guide regression test selection, even for new or less experienced test personnel. It also provides an effective means of quantifying quality of test suite as a by-product. In Chapter 6, evaluations and discussions were presented based on the case studies. The results of the case studies indicated that this technique was cost-effective and was efficient in finding defects.

7.2 Contributions of Thesis

The contributions of this thesis can be summarized as follows:

- Demonstrate that traceability is essential for conducting and managing regression analysis and testing, and define useful requirement traceability links for regression analysis.
- Provide a new *specification-based* strategy for selecting regression test cases. Unlike previous *specification-based* strategies, our approach has less subjective selection criteria, is systematic, and can be implemented in test tools. In addition, it is directed toward coverage of the modification ripple effect, and toward coverage of *risk*.
- Present techniques for implementing our approach to yield regression tests consisting of both *Targeted Tests* and *Safety Tests*. The definitions of *Targeted Tests* and *Safety Tests* provide a clean and useful focus for the definition of *specification-based* regression tests.
- Define and illustrate the use of *risk analysis*, a customer-oriented concept, in *regression testing*, and show how the *Risk Exposure* metric can be used to measure the quality of regression test suites.

7.3 Future Work

Our future work includes using more components for case studies, performing additional empirical studies to evaluate the effectiveness of our technique, applying metrics (for example, specification traceability and coverage reports) to guide when to stop regression testing, and running pilot project implementing our approach in a

production test environment. We will also continue to use statistical analysis tools such as SPSS to aid in determining the cost-effectiveness of our technique in practice.

In the following, we briefly expand on some of these issues:

- Develop metrics to help decide when to stop

  In practice, time and resources are always limited. When to stop testing is a problem since the first day. We want to use time and resources in the most effective way and gain enough confidence about the quality of the system under test. As a phase of project development, sometime we need to stop testing when we feel that system quality is good enough, and leave time and resources to other tasks. Metrics are useful tools to help make the decision. One example matrix is specification coverage report. When a specific specification coverage target has been reached, we can stop testing with enough confidence.

- Implement the technique in production test environment

  Compared to computer processing time, the time of software engineers are much more costly. Hence, automation is essential to the usability of a testing methodology. Our regression testing technique can be automated and the implementation of the technique in a production test environment is under discussion.

- Use statistical analysis tools to determine cost-effectiveness of our technique in practice

  When we evaluated our technique with SPSS, we found that statistical analysis tools were powerful in measure processes and the improvement of the processes. To determine the cost-effectiveness of our technique in practice, we will keep on using statistical analysis tools. We will also work on statistical methods in quality management.

  From our experience, incomplete and out-of-date documentation exists

throughout project development and always causes big problems. Because requirement capturing and system design are often done in an informal way, requirement and design documentation is written manually. That causes more serious problems since the document is error-prone. As a part of testing, documentation testing is not done efficiently. A possible solution is applying formal methods in requirement capture and design phases. But formal validation and verification methodologies in the O-O paradigm still have practical problems. Hence, investigation and new research is needed in order to work out these problems. Also, an appropriate documentation testing method needs to be defined. This thesis has opened up new research topics related to formal methods, documentation testing, and Object-oriented regression testing.

References

[Ambler 00] Scott W. Ambler, When to Use UML Activity Diagrams, *IBM developerWorks web site*, http://www-4.ibm.com/software/developer.

[Amland 99a] Stale Amland, Risk Based Testing and Metrics: Risk analysis fundamentals and metrics for software testing including a financial application case study, 5th *International Conference EuroSTAR'99*, November 1999, Barcelona, Spain.

[Amland 99b] Stale Amland, Risk-based testing: Risk analysis fundamentals and metrics for software testing including a financial application case study, *The Journal of Systems and Software* 53 (2000), 2000, pp. 287 – 295.

[Amland 00] Stale Amland, Risk Based Testing and Metrics: Risk analysis fundamentals and metrics for software testing including a financial application case study, *The Journal of Systems and Software*, Vol. 53, 2000, pp. 287 – 295.

[Bach 98] James Bach, Good Enough Quality: Beyond the Buzzword, *IEEE Computer*, August 1998, pp. 96 - 98.

[Bach 99] James Bach, Heuristic Risk-Based Testing, *Software Testing and Quality Engineering Magazine*, November 1999, pp. 96 - 98.

[Bashir+99] Imran Bashir, and Amrit L. Goel, *Testing Object-Oriented Software: Life Cycle Solutions*, Springer-Verlag New York, Inc., 1999.

[Binder 00] Robert V. Binder, *Testing Object-Oriented Systems*, Addison-Wesley Publishing Inc., 2000.

[Binkley 97] David Binkley. Semantics Guided Regression Test Cost Reduction, *IEEE Transactions on Software Engineering*, VOL 23, NO 8, August 1997, pp. 498 – 516.

[Booch 94] G. Booch, *Object-Oriented Analysis and Design with Applications*, Benjamin/Cummings, 2nd edition, 1994.

[Booch 96] G. Booch, *Object Solutions*, Addison-Wesley, 2nd edition, 1996.

[Brown+90] P. A. Brown and D. Hoffman, The Application of Module Regression Testing at TRIUMF, *Nuclear Instruments and Methods in Physics Research*, Section A, A293 (1-2), August 1990, pp. 337-381.

[Burr+96] A. Burr, M. Owen, *Statistical Methods for Software Quality: Using Metrics for Process Improvement*, International Thomson Computer Press, 1996.

[Chen+02] Yanping Chen, Robert L. Probert, D. Paul Sims, specification-based Regression Test Selection with Risk Analysis, *In Proceeding of CASCON'02*, September 2002, pp. 60 – 73.

[Fowler+00] Martin Fowler, and Kendall Scott, *UML Distilled Second Edition: a Brief Guide to the Standard Object Modeling Language*, Addison Wesley, 2000.

[Gotel+94] Orlen C. Z. Gotel, and Anthony C. W. Finkelstein, An Analysis of the Requirements Traceability Problem, *In Proceedings of the First International Conference on Requirements Engineering*, IEEE Computer Society Press, 1994, pp. 94-102.

[Gunnar+01] Gunnar Overgaard, Bran Selic, Conrad Bock, and Morgan Bjorkander, Object Modeling with OMG UML Tutorial Series: Behavioral Modeling, *In Proceedings of UML Workshops 2001*, December, 2001.

[Hajla 97] Halim Ben Hajla, *Traceability in Object-Oriented Quality Engineering, A Basis for Regression Analysis of Object-Oriented Software*, Masters Thesis, University of Ottawa, Canada, 1997.

[Harmelen+97] Mark van Harmelen, and Bernard Horan, Object-Oriented Models in User Interface Design, *A CHI97 Workshop SIGCHI*, October 1997.

[Harrold+93] Mary Jean Harrold, R. Gupta, and M. L. Soffa, A Methodology for Controlling the Size of a Test Suite, *ACM Transactions on Software Engineering and Methodology*, 2(3), July 1993, pp. 270-285.

[Harrold+01] Mary Jean Harrold, James A. Jones, Tongyu Li, and Donglin Liang, Regression Test Selection for Java Software, *In Proceeding of the ACM Conference on OO Programming, Systems, Languages, and Applications* (OOPSLA'01), 2001.

[IEEE 610] ANSI/IEEE standard 610.12-1990: *Glossary of Software Engineering Terminology*. New York: The Institute of Electrical and Electronic engineers, 1987.

[Jones 97] Capers Jones, *Software quality: analysis and guidelines for success*, International Thompson Computer Press, 1997.

[Kan 02] Stephen H. Kan, *Metrics and Models in Software Quality engineering (Second Edition)*, Pearson Education Inc., 2002.

[Kung+95] David C. Kung, Jerry Gao, and Pei Hsia, Class Firewall, Test Order, and Regression Testing of Object-Oriented Programs, *Journal of Object-Oriented Programming* 8(2), May 1995, pp. 51 - 65.

[Leung+90] H.K.N. Leung and L.J. White, A study of integration testing and software regression at the integration level, *In Proceeding of the Conference on Software Maintenance*, November 1990, pp. 290 - 300.

[Leung+91] Harenton K.N. Leung and Lee J. White, A Cost Model to Compare Regression Test Strategies, *In Proceeding of the Conference on Software Maintenance*, 1991, pp. 201-208.

[Lieberman 01] Ben Lieberman, Using UML Activity Diagrams for the Process View, *Web site of Rational Software Inc.*, http://www.therationaledge.com/.

[Luiu+93] H. Luiu, L. Robson and R. Ellis, A data managements system for regression analysis and testing. *International Conference on Software Quality Management*,

1993, pp. 527 – 539.

[Mayrhauser+94] Anneliese von Mayrhauser, Richard Mraz, Jeff Walls, and Pete Ocken, Domain Based Testing, *In Proceedings of the Conference on Software Maintenance*, September 1994, pp. 26 – 35.

[McGregor+01] John D. McGregor, and David A. Sykes, *A Practical Guide to Testing Object-Oriented Software*, Addison Wesley Inc., 2001.

[Myers 79] Glenford L. Myers, *The Art of Software Testing*, Wiley-Interscience, 1979.

[Nguyen 01] Hung Q. Nguyen, *Testing Applications on the Web*, John Wiley & Sons Inc., 2001.

[Odell 98] James Odell, *Advanced Object-Oriented Analysis and Design using UML*, Cambridge University Press, 1998.

[Pfleeger 00] Shari Lawrence Pfleeger, Risky Business: What We Have Yet to Learn about Risk Management, *The Journal of Systems and Software*, Vol. 53, 2000, pp. 265 – 273.

[Probert 02] Robert L. Probert, Software Quality Engineering course notes, University of Ottawa, 2002.

[Rothermel+96] Gregg Rothermel and Mary Jean Harrold, Analyzing Regression Test Selection Techniques, *IEEE Transactions on Software Engineering*, V.22, no. 8, August 1996, pp. 529 – 551.

[Rothermel+00] Gregg Rothermel, Mary Jean Harrold, and Jeinay Dedhia, Regression Test Selection for C++ Software, *Journal of Software Testing, Verification, and Reliability*, V.10, no. 2, June 2000.

[Rumbaugh+99] James Rumbaugh, Ivar Jacobson, and Grady Booch, *The Unified Modeling Language Reference Manual*, Addison Wesley Inc., 1999.

[Saleh+02] K. Saleh, R. L. Probert, W. Li, and W. Fong, An approach for high-yield requirements capture for e-commerce and its application, *International Journal on Digital Libraries*, Vol. 3, No. 4, May 2002, pp. 302-308.

[SIAM 96] Inquiry Board Traces Ariane 5 Failure to Overflow Error, *SIAM News*, Society for Industrial and Applied Mathematics, Vol. 29., Number 1996.

[Spanoudakis 02] George Spanoudakis, Plausible and Adaptive Requirement Traceability Structures, *In the Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering*, 2002.

[SPSS] SPSS Inc., *SPSS web site*, http://www.spss.com.

[Stocks+96] Phil Stocks, and David Carrington, A Framework for Specification-Based Testing, *IEEE Transactions on Software Engineering*, Vol. 22, No. 11, November 1996, pp: 777 - 793.

[Toranzo+99] Marco Toranzo, and Jaelson Castro, A comprehensive Traceability Model to Support the Design of Interactive Systems, *the 13th European Conference on Object-Oriented Programming (ECOOP'99)*, June 1999, pp. 283-284.

[UML 01] OMG Unified Modeling Language Specification, v1.4, Object Management Group, September 2001.

[Vieira+00] Marlon E. Vieira, Marcio S. Dias, and Debra J. Richardson, Object-Oriented Specification-Based Testing Using UML Statechart Diagrams, *ICSE2000 -Workshop on Automated Program Analysis, Testing, and Verification*, June 2000.

[White 91] L. J. White, *Software Testing and Verification*, Elsevier Science B.V., 1991.

[Wong+95] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur, Effect of Test Set Minimization on Fault Detection Effectiveness, *In Proceeding of 17th International Conference on Software Engineering*, April 1995, pp. 41 -50.

Appendix A: Example Customer Cost Questionnaire

Basic Questionnaire (for Assessing Cost and Risk)

Control-flow

1. Does the test case test a new feature?

   Yes – 2 points No – 0 points

2. Does the test case test a changed feature?

   Yes – 2 points No – 0 points

3. Does the test case test basic functionality?

   Yes – 2 points No – 0 points

4. How critical is the feature?

   Very critical – 2 points Somehow critical – 1 point Not critical – 0 points

5. How often will the customer use the feature?

   Very frequently – 2 points Frequently – 1 points Seldom – 0 points

6. How many other pieces of the system are dependent on the feature tested by the test case?

   Many – 2 points Some – 1 points Few – 0 points

7. Does the test case test a feature that is very important to the business, such as sale points that the marketing team has promised to customers, or features that set us apart

from our competition?

Yes – 2 points No – 0 points

8. Does the test case test a feature that has customer open defect(s) in previous release?

Yes – 3 points No – 0 points

9. Does the test case test a feature that is related to company's reputation, such as security of user's data?

Yes – 2 points No – 0 points

Data

10. How many customers will use similar data?

Many – 2 points Some – 1 points Few – 0 points

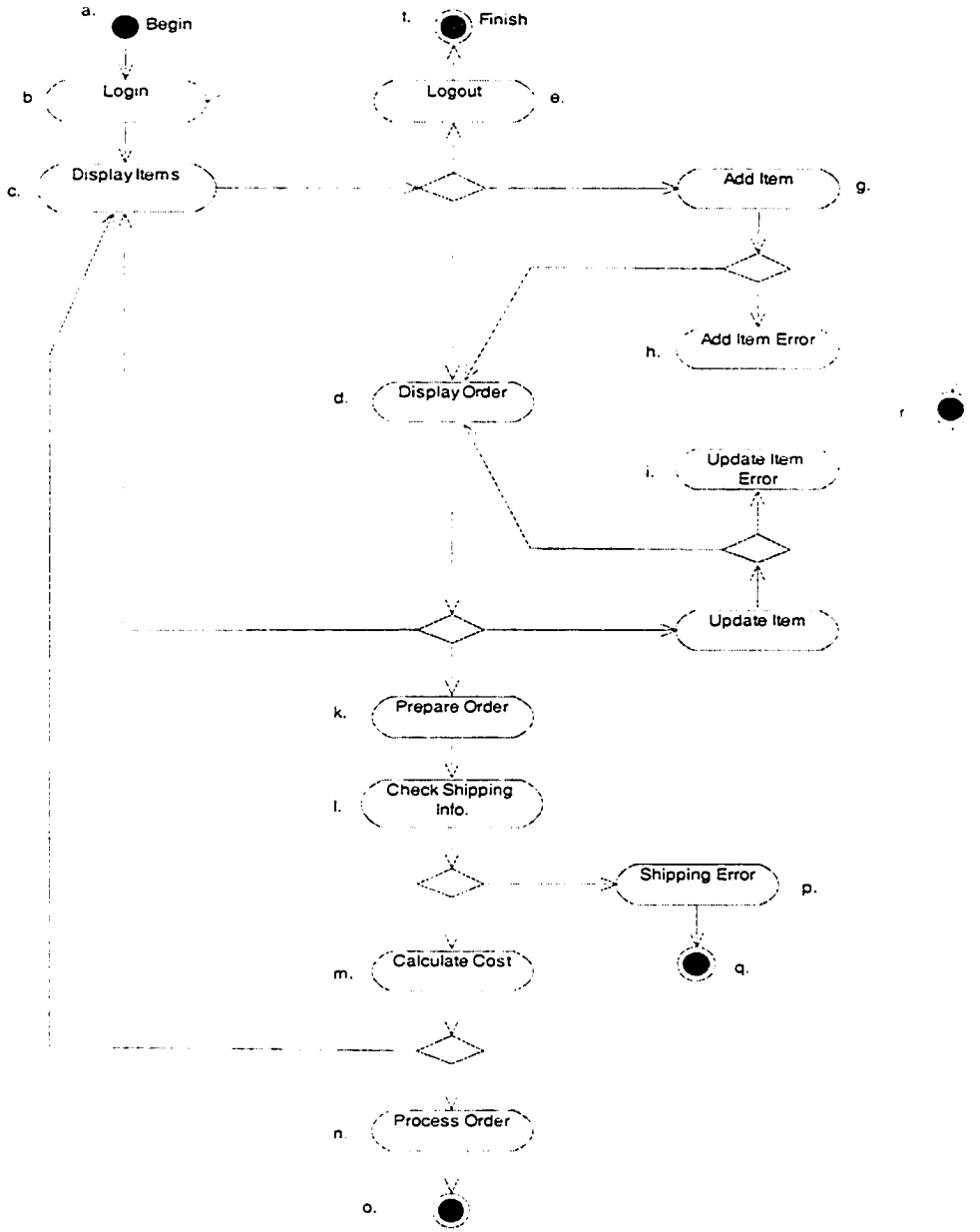
11. How often does the customer use similar data?

Very frequently – 2 points Frequently – 1 points Seldom – 0 points

12. Does the data have issues related to company's reputation, such as security?

Yes – 2 points No – 0 points

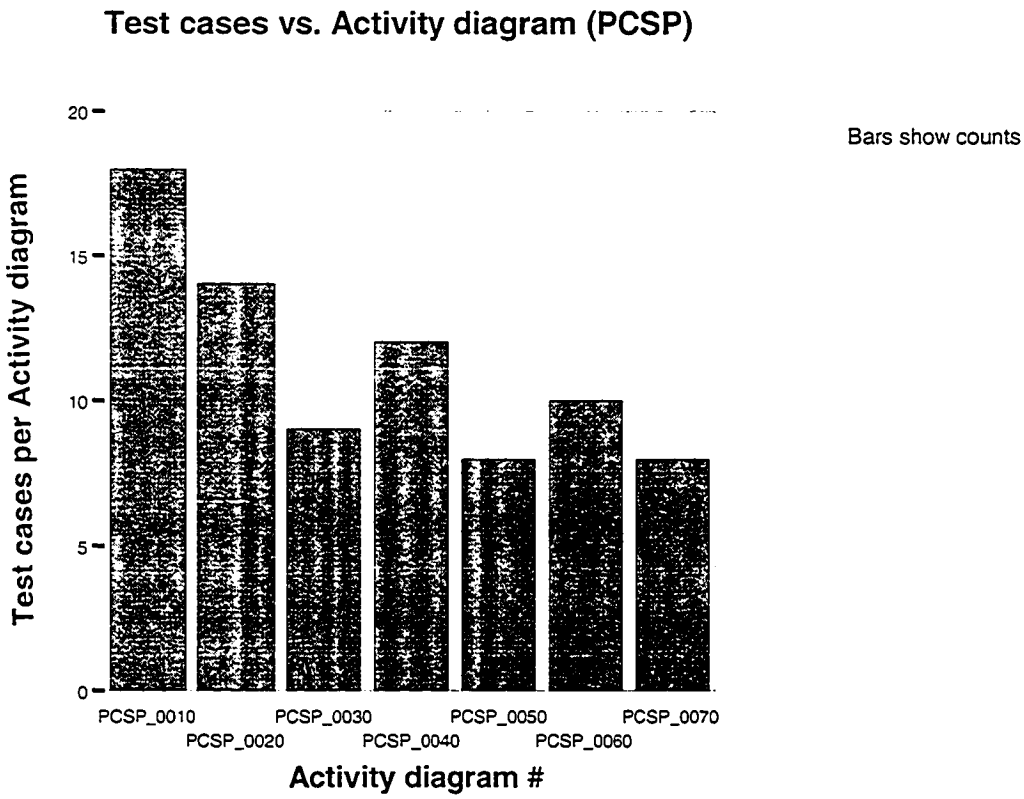
Appendix B: Simplified Activity Diagram of Case Studies

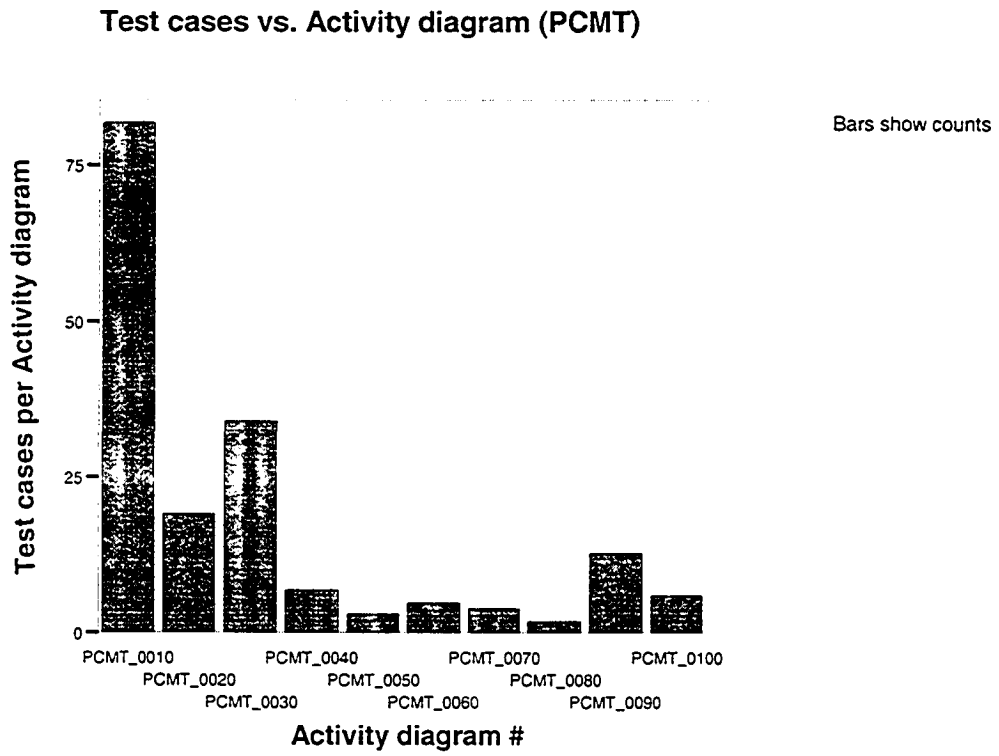


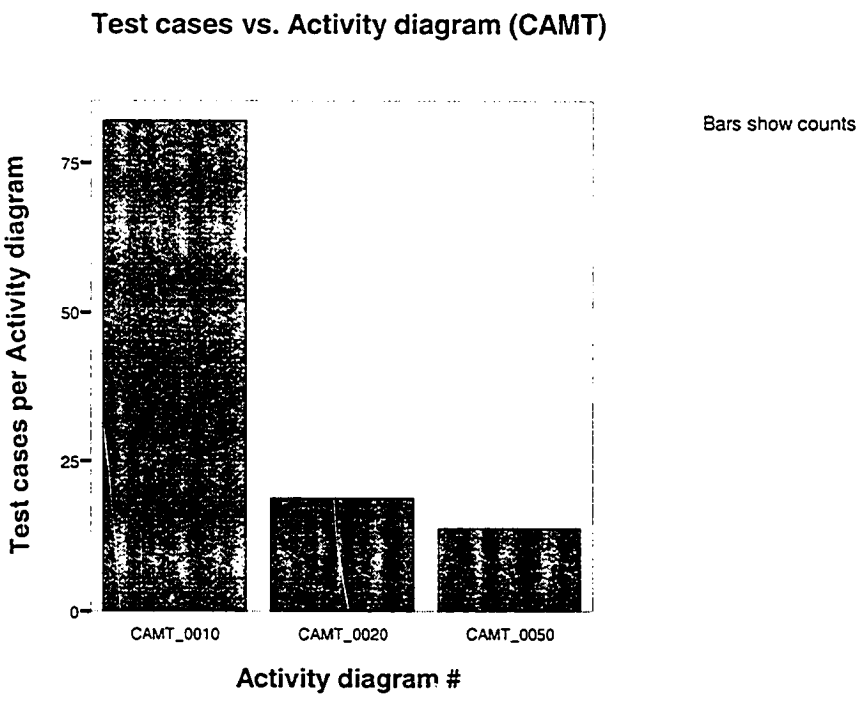
Appendix C: Statistical Analysis Reports

Reports for three components together

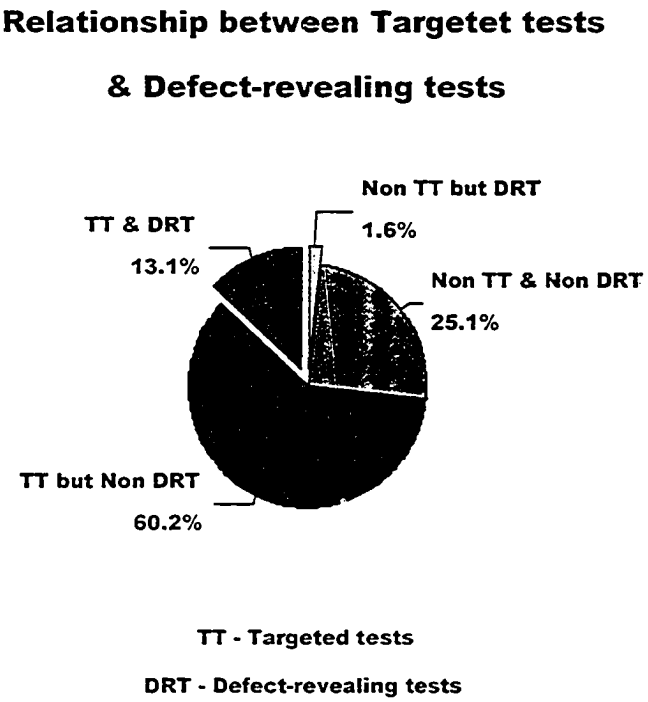
1. Bar charts of test cases distribution



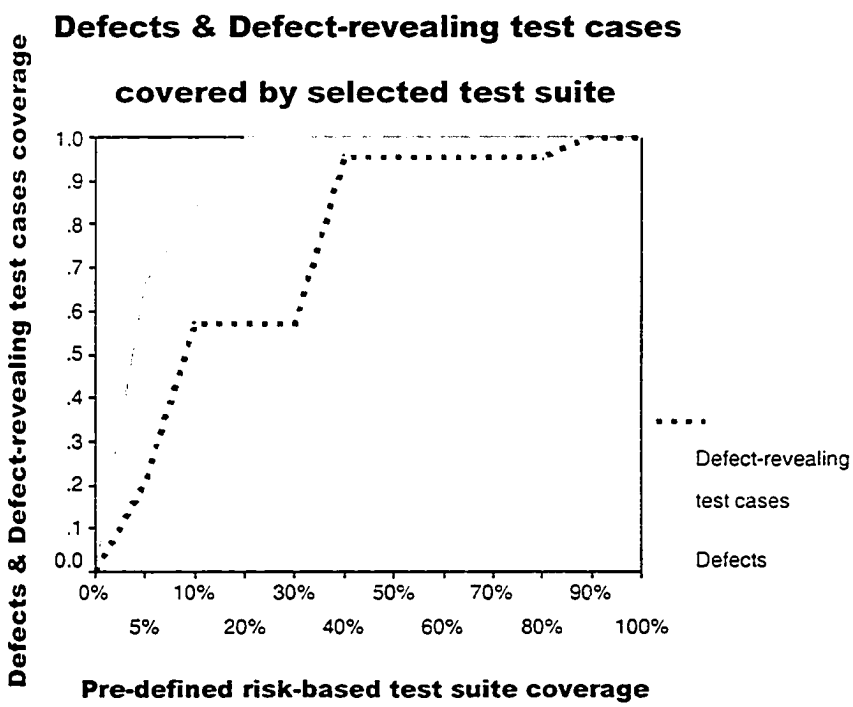

Test cases vs. Activity diagram (PCMT)
Bars show counts
Test cases per Activity diagram
75
50
25
0
PCMT_0010
PCMT_0020
PCMT_0030
PCMT_0040
PCMT_0050
PCMT_0060
PCMT_0070
PCMT_0080
PCMT_0090
PCMT_0100
Activity diagram #


Test cases vs. Activity diagram (CAMT)
Bars show counts
Test cases per Activity diagram
75
50
25
0
CAMT_0010
CAMT_0020
CAMT_0050
Activity diagram #

2. Effectiveness and Efficiency of Targeted tests selection



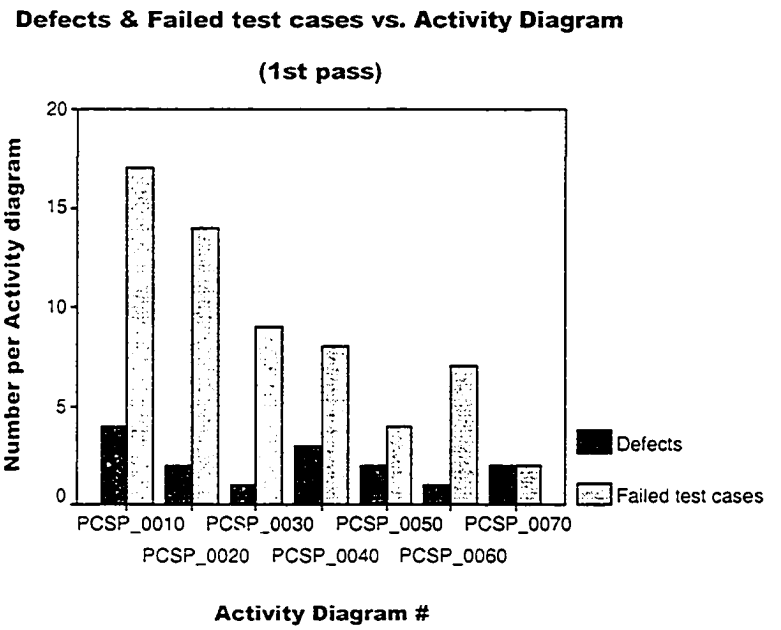
3. Effectiveness and Efficiency of risk-based test cases selection

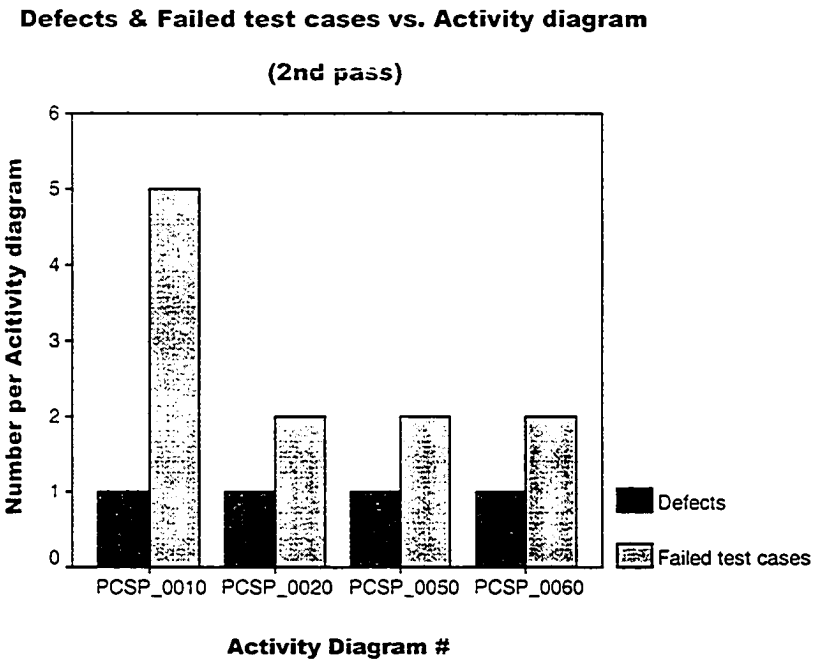


Reports for each individual component

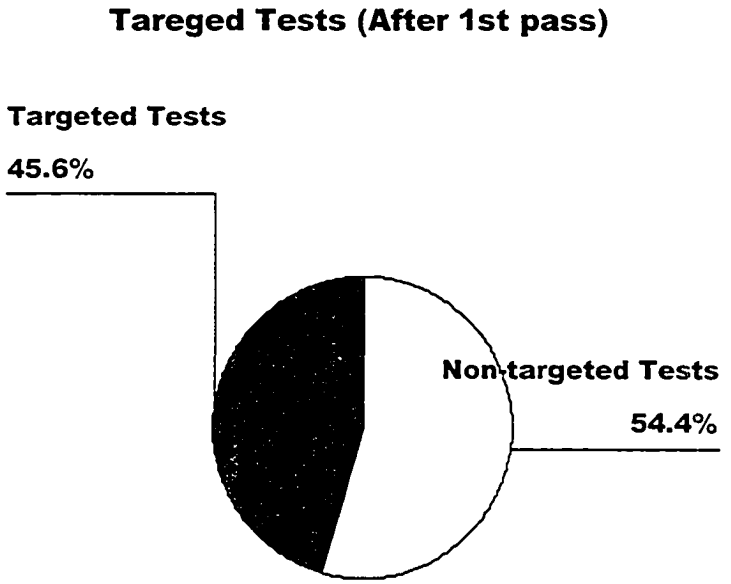
1. Component PCSP

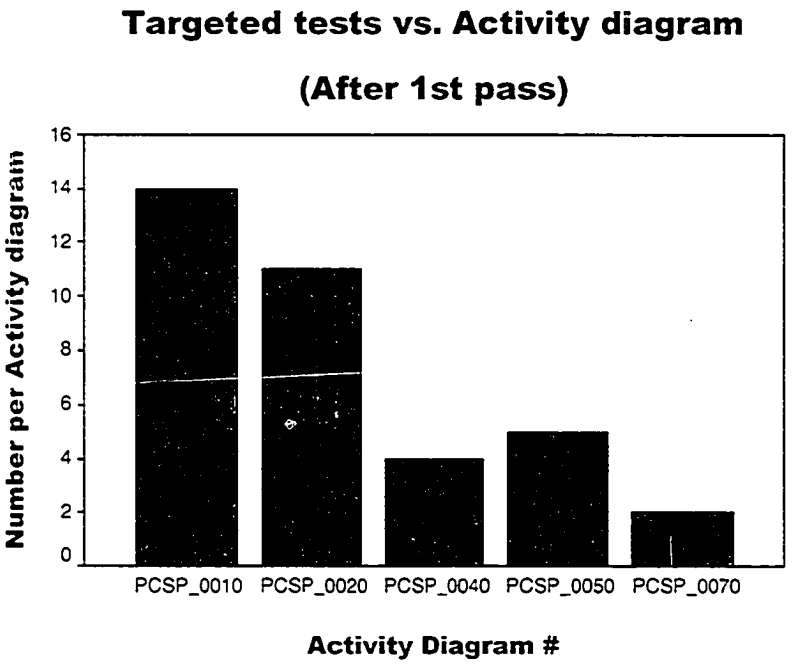
(1) Defect summary

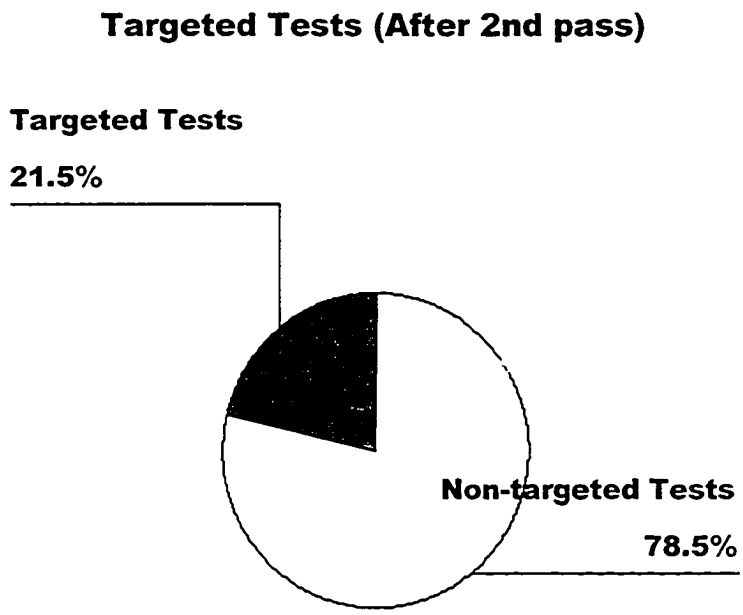


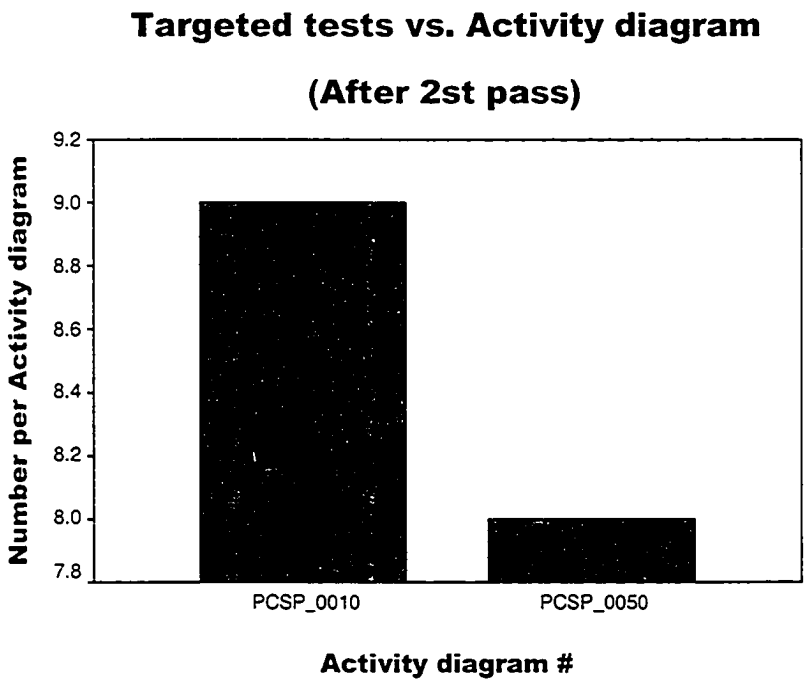


(2) Targeted Tests selection and distribution

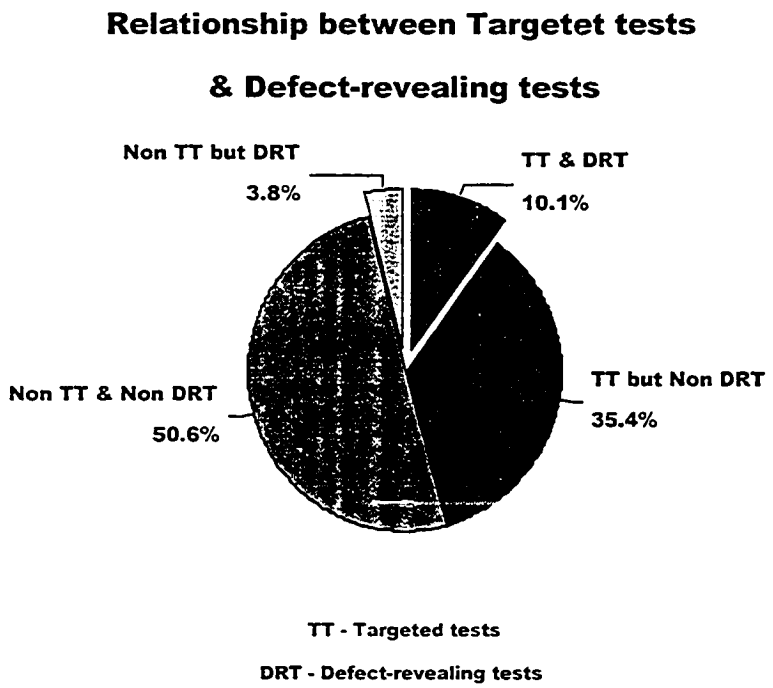




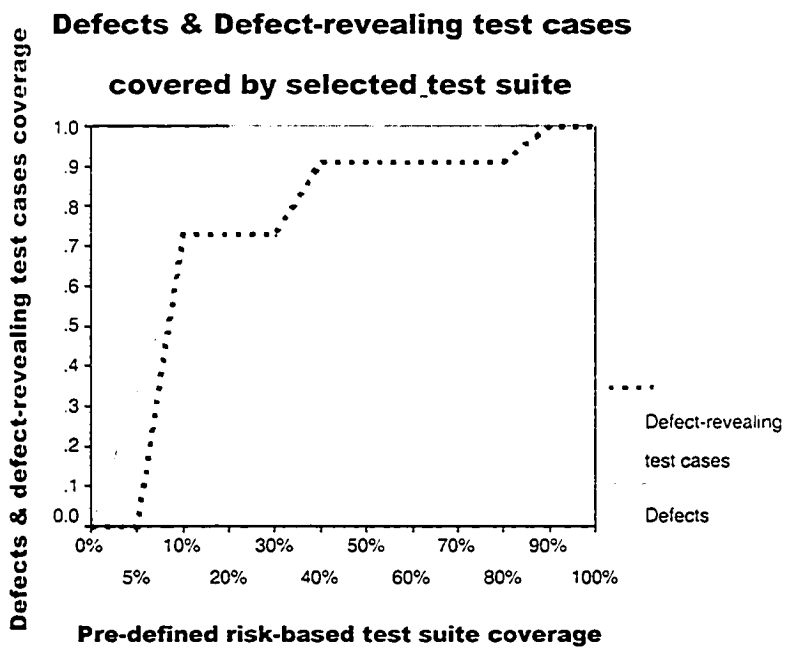




(3) Effectiveness and efficiency of Targeted Tests selection

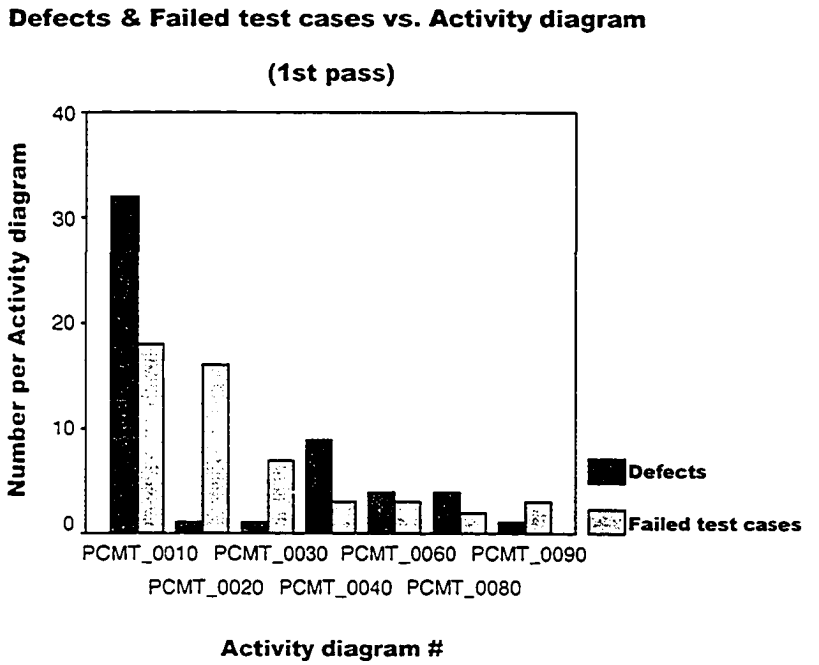


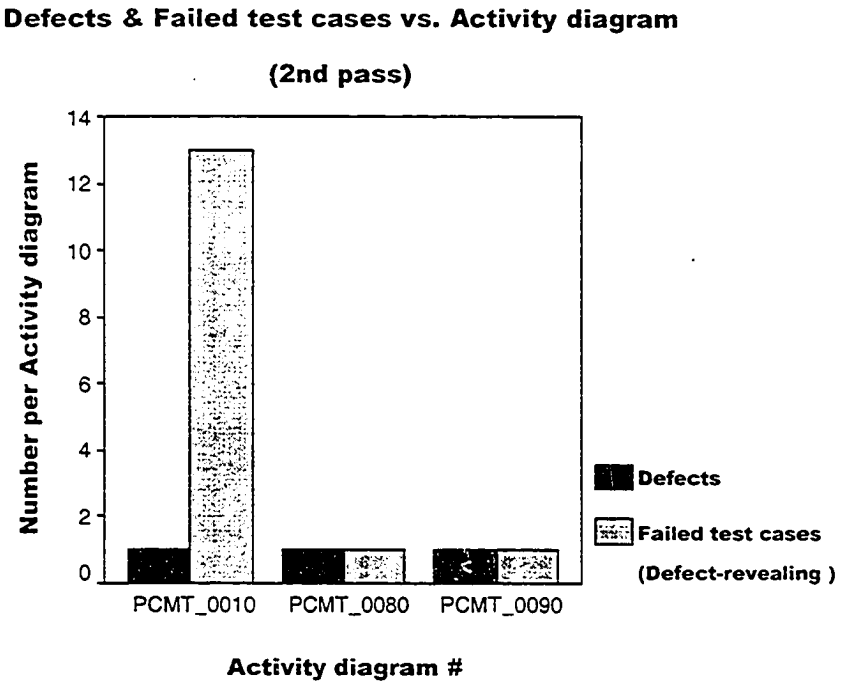
(4) Effectiveness and efficiency of risk-based test case selection



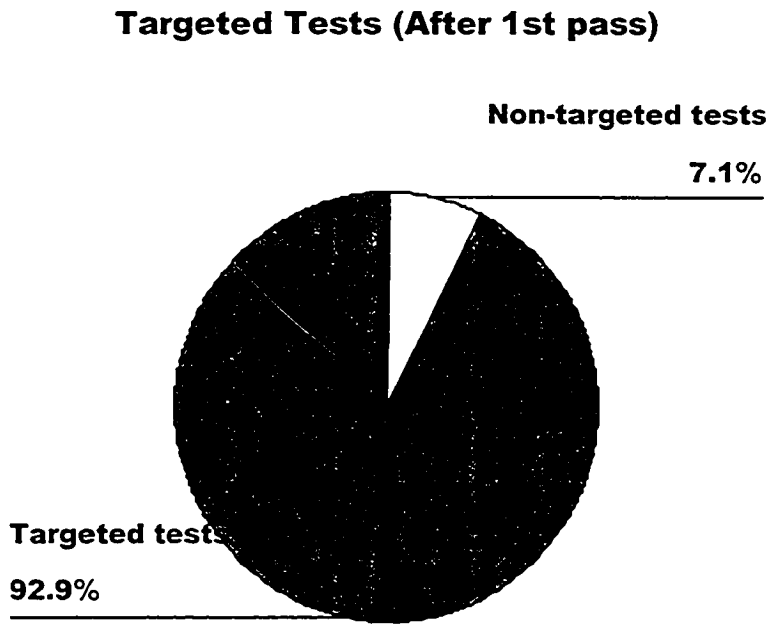
2. Component PCMT

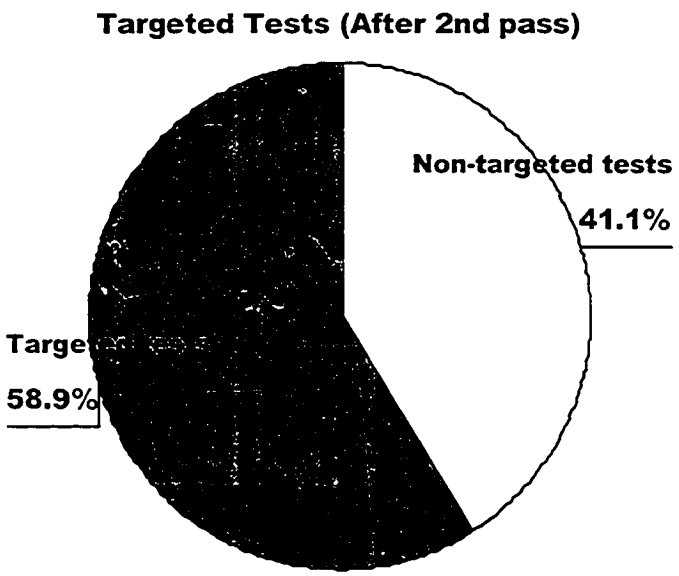
(1) Defect summary



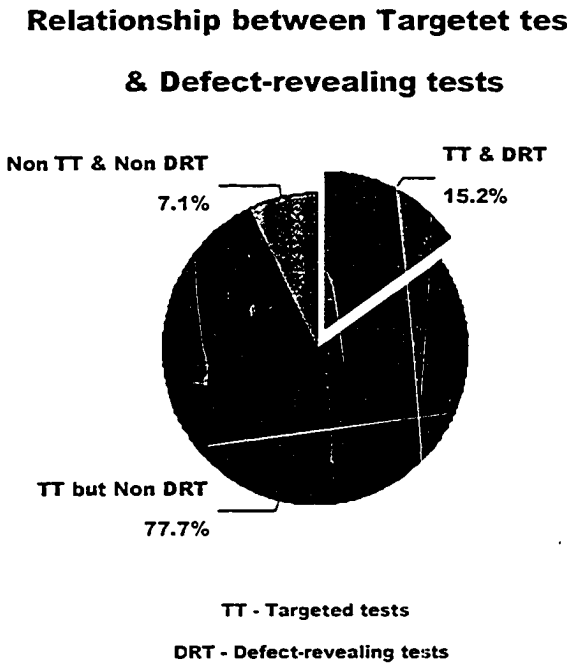


(2) Targeted Tests selection and distribution

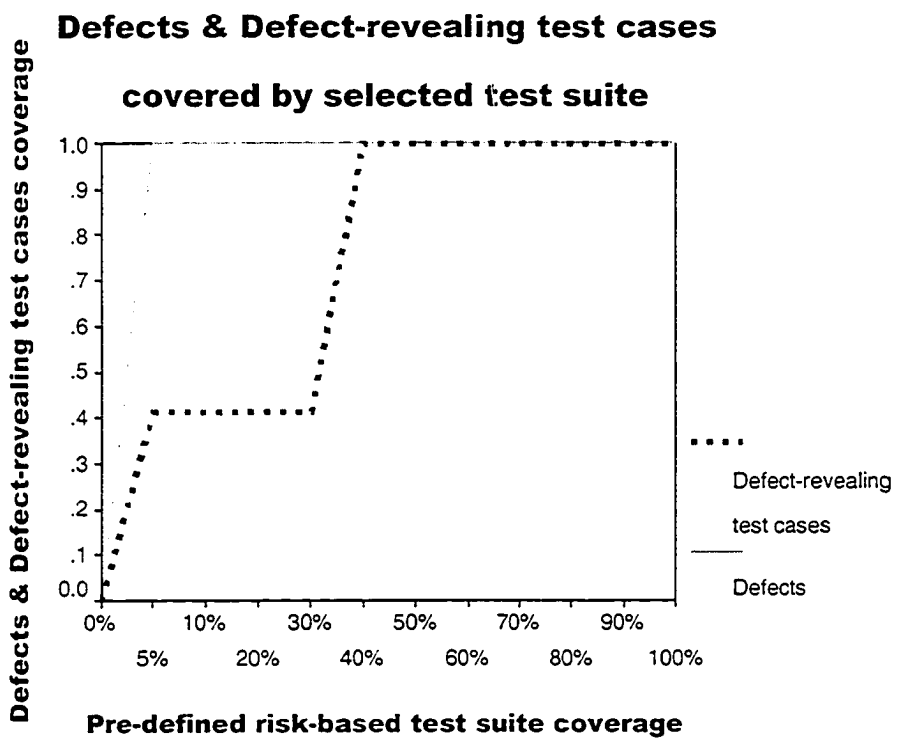




(3) Effectiveness and efficiency of Targeted Tests selection

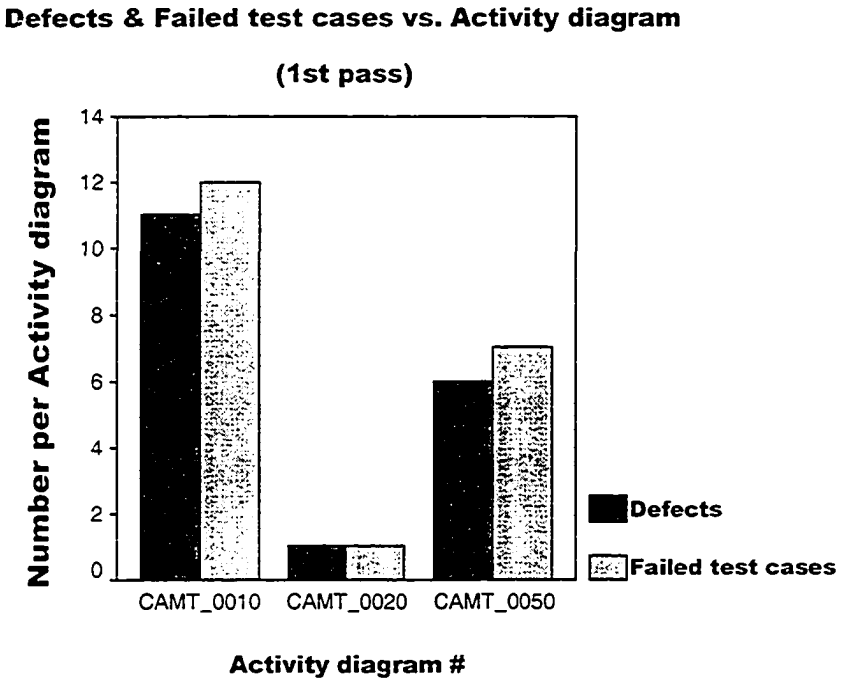


(4) Effectiveness and efficiency of risk-based test case selection



3. Component CAMT

(1) Defects Summary



(2) Targeted Tests selection and distribution