A Regression Test Selection Technique Applied to Legacy Systems

A REGRESSION TEST SELECTION TECHNIQUE APPLIED TO LEGACY SYSTEMS

BY
AKBAR ABDRAKHMANOV, B.Eng.

A THESIS
SUBMITTED TO THE DEPARTMENT OF COMPUTING AND SOFTWARE
AND THE SCHOOL OF GRADUATE STUDIES
OF MCMASTER UNIVERSITY
IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF APPLIED SCIENCE

Master of Applied Science (2011) McMaster University
(Computing and Software) Hamilton, Ontario, Canada

TITLE:	A Regression Test Selection Technique Applied to Legacy Systems
AUTHOR:	Akbar Abdrakhmanov B.Eng., (Organization and Technology of Information Security) Kazakh National Technical University, Almaty, Kazakhstan
SUPERVISOR:	Dr. Tom Maibaum and Dr. Alan Wassyng
NUMBER OF PAGES:	x, 80

Abstract

Software testing is a necessary step in both the software production chain and during future maintenance. Testing can be performed using many different techniques. It is much easier to test a piece of software during the original development process, rather than after the system has been deployed and in use for many years. However, testing must be performed whenever software is altered. How should testing be performed on legacy software? The answer to this question is the focus of this thesis.

Regression testing is performed on software during its maintenance stage. Usually it is done after changes have been made to the software or its environment. The concept of regression testing does not assume the existence of a test suite for the software. It might be the case that test cases have to be created from the scratch. However, when software with a huge existing test suite is under consideration, then it is not always a good idea to run the whole test suite in order to perform regression testing. This might be too expensive both in time and human effort. This problem motivate the concept of regression test selection, which tells us that by using certain selection criteria it is possible to select a subset of an existing test suite and run that subset on the software instead.

Our project involves reducing the burden of regression testing for a legacy system that is built on top of the Oracle E-Business Suite and Oracle database. The changes are made through patches that are sent by Oracle. Most of the time, patches do not introduce big changes to the E-Business Suite. Also, it is likely that the legacy software application uses a relatively small portion of Oracle's Enterprise Resource Planning (ERP) system. In this case, it is crucial to be able to select a subset of the test suite for regression testing. Running the whole test suite can be very costly, and

is almost always not necessary.

The approach we adopted relies on building the access dependency graph of the whole system first. An access dependency graph is a control flow graph but with fields and methods of classes as nodes. Then, after identifying which places in the Oracle E-Business Suite changed after patching, the inverse calling paths are built with changed places as starting points and functions in the customer's application as end points. Now, when affected functions have been identified, it should be possible to select a subset of the existing test suite which will exercise functions that have been directly or indirectly changed by the patch, and which will not include tests that are not impacted by the changes to the software system embodied in the patch. We also demonstrate that the selection technique proposed in this work is safe (does not miss tests that should be included). This gives us some confidence about the applicability of the algorithm.

Acknowledgements

I would like to acknowledge my supervisors Dr. Maibaum and Dr. Wassyng for their incredible support throughout my study at McMaster University. That was an unbelievable experience for me and I appreciate them for always providing me with motivation to sharpen my knowledge.

I highly appreciate support from my colleagues Wen Chen and Asif Iqbal. Together we could carry the project to its current state. That was a good team work. I would also like to acknowledge my colleagues Hao Xu and Quang Tran for their friendship and invaluable feedback on my thesis.

Thanks to all my friends and colleagues who made my days here in Canada unforgettable. Thanks to my parents for their endless support. You always believed in me and I never get tired of meeting your expectations.

Contents

List of Figures

Chapter 1

Introduction

In this chapter we provide a definition of the given problem, identify what the scope of the research is and list the contributions that were made in this work.

1.1 Definition of the Problem

Today, many companies around the world use a form of software that is called a legacy system. A legacy system is software that resists modification and evolution [SPL03]. In most cases legacy systems have been produced many years ago. And because of rapid development of technology these software become outdated, even though they still do their job and represent a certain asset for a company. There are lots of problems around maintaining such systems including the uncertainty in ways of testing them. For instance, to keep up with demands of a modern market a company wants to change some artifact of the software or its environment. Examples of such changes are numerous: migrating to another operating system, upgrading to another hardware platform, switching to a different DBMS [1], performing minor changes to the source code (if it is available), and so on. Doing regression testing after each such change is very crucial, because any kind of change might introduce a new fault, and thus the behaviour of modified software might not be as expected.

The legacy system that is given in our project is a software product implemented on top of the Oracle E-Business Suite and the Oracle database. It is assumed

[1] Database Management System

that it is written in Java. At a very high level, the whole system looks like the one depicted in the figure 1.1.

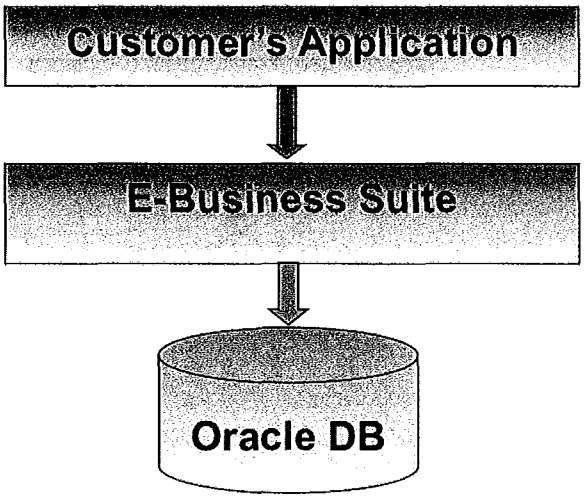


Figure 1.1: 3-layer representation of the system

From time to time Oracle releases patches that, if applied, eventually change parts of the E-Business Suite or the database. The customer's application on the other hand might directly or indirectly reference those changed parts. Figure 1.2 shows an example of dependencies that might exist in the system [AIC10].

Certainly, the customer needs to perform regression testing on its application after each such patch. Otherwise some faults in the functionality of the system might go undetected and would appear when they are least expected. Now, assuming that the customer has a test suite for its application, the following problems might occur:

1. Running the whole test suite might be very expensive both in terms of time and human effort, which would eventually cost a company a great amount of money.

2. Even if the whole test suite had been run, it would be very difficult to pinpoint where the detected faults come from because the testing is essentially "blind" [2].

[2]It is black-box testing where the customer does not have any insights about the application and the results of the tests are only relied on for the input-output correspondences of each test case.

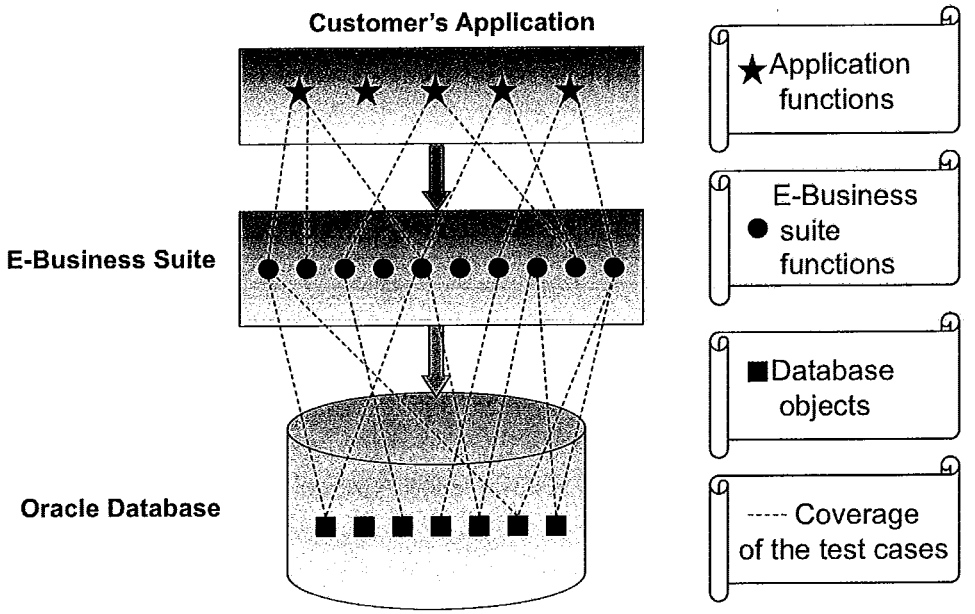


Figure 1.2: Dependencies in the system

In general this is a very large problem. So, before we discuss the details of how we solved the problem, we need to identify the scope of our work.

1.2 Scope and Goals of This Research

First, our object of concern is a 3-layer system as shown in figure 1.1 where we have an Oracle database, the E-Business Suite ERP system built on top of the database and the customer's application that accesses the objects in the E-Business Suite and the database.

The legacy system in our case is the customer's application that we assume is fully written in Java. It is shown in [Abd10a] that the Oracle E-Business Suite patches consist of many different file types that potentially change the E-Business Suite or the database. In this work we limited our focus only on parts that were written in Java, i.e. we are only concerned about files with a "class" extension (Java binaries). We have also excluded from our focus accesses to the database objects, e.g., all types of SQL statements and calls to stored functions/procedures.

It is worth noting that the basic configuration of the E-Business Suite that

we installed on our test server consists of around 230,000 Java classes and almost 4 million methods.

The goal of this research is to overcome the problems described in the previous section and provide a cost-effective solution that would certainly benefit the customer in performing the regression testing of their system. The method we propose is to select a subset from the existing test suite that would include all test cases that might expose faults in a changed version of the system. More details are in the next section with explanation of contributions that were made in this research.

1.3 Contributions

In this work we propose a safe regression test selection technique where the safety property means that a technique selects all test cases from the test suite which might expose faults in the changed system. The technique consists of the steps illustrated in figure 1.3.

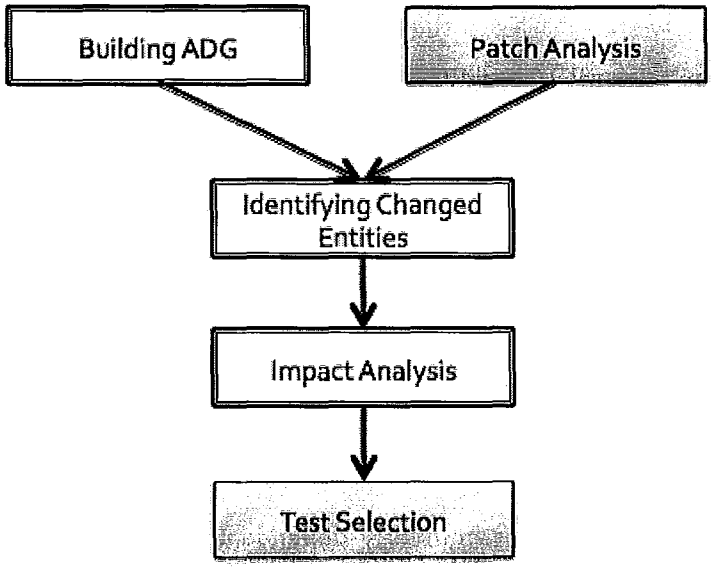


Figure 1.3: 5-step solution chart

The contributions to this work made in this thesis are those steps that are

shaded in figure 1.3, i.e., patch analysis and test selection. The following is a brief explanation of each step:

1. Build an access dependency graph (ADG) of the whole system where nodes are methods and fields of the classes (entities) and edges are dependencies between those nodes [Iqb11].

2. Run Oracle's "Auto Patch" command line tool in test mode to generate a log file that includes information on which existing files of the E-Business Suite and the database change [AIC10]. Then execute the LGIParser tool to output that information to the database in appropriate format [Abd10b].

3. Run the ClassDiff tool to compare original and modified versions of changed files to identify changed entities and output this information to user-defined XML files [Iqb11].

4. Get the names of changed entities from the XML files and, using ADG information, run an inverse calling path searching tool called InversePathGenerator to identify the entities in the customer's application that are directly or indirectly dependent on the changed entities in the E-Business Suite and the database and therefore are affected. Also load the output to the database [Che11].

5. Get the names of affected entities in the customer's application and run the test selection tool called TestSelect to identify which test cases need to be rerun and which entities miss being tested.

At an abstract level, identifying changed and affected parts of the system is illustrated in figure 1.4:

The figure 1.4 represents the steps of building the access dependency graph, identifying changed parts of the system (red coloured shapes), identifying affected parts that are directly or indirectly dependent on those parts that changed (yellow coloured shapes) in the E-Business Suite and the database and eventually identifying parts of the customer's application that are affected too (yellow stars). Now, knowing about which entities of the customer's application are potentially of concern, we can run the program for test selection to identify which test cases to rerun and for which

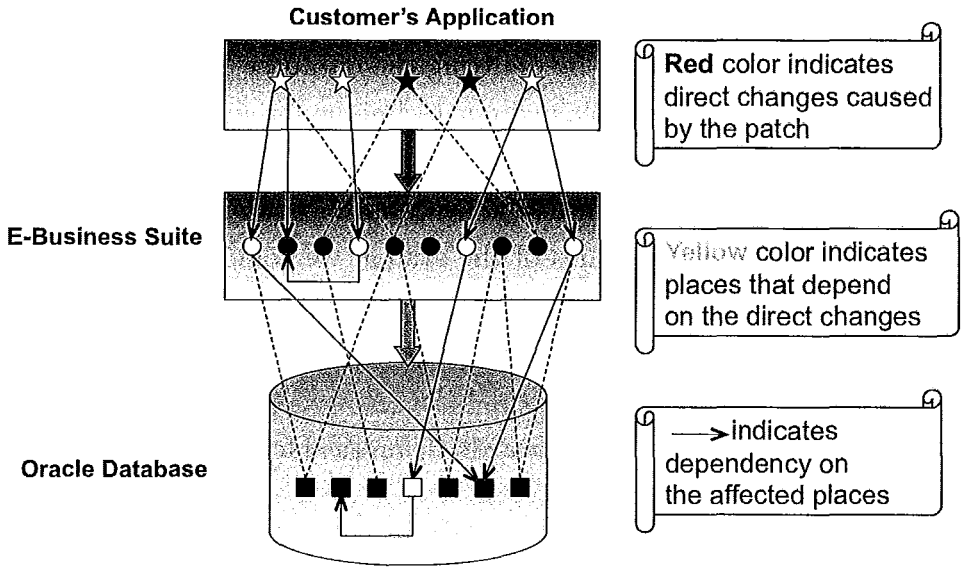


Figure 1.4: Identifying changed and affected parts of the system

methods there are no test cases in the existing test suite. Currently in this work we excluded the database layer from our analysis.

The other contribution in this work is establishing the relationship of our test selection algorithm to existing techniques and used the same framework to assess its properties. More details about the regression test selection techniques and the framework are provided in Chapter 2. Assessment of our technique is provided in Chapter 4.

1.4 Summary

In this chapter we identified the problem that we have been working on in our project. The legacy system in our case is the customer's application that is dependent on the Oracle E-Business Suite and the database. Since Oracle's patches are issued with certain regularity and running the whole regression test suite to verify if the behaviour of the system did not change is very expensive, there is a huge need in performing cost-effective regression testing of the system.

We have also narrowed the scope of this research as it is in general a very large problem. We assume that the customer's application is written in Java and as for the

E-Business Suite layer we only consider its Java classes. We also reduced our 3-layer system down to two layers by omitting the database part.

Then we briefly explained our technique for solving the problem. By building an access dependency graph (ADG) of the whole system, where nodes are methods and fields of the classes (entities), we could identify all dependencies among nodes. And thus from changed parts of the system we could also identify affected parts of the customer's application. Then we run our test selection program to identify only necessary test cases and also output those customer's application entities that the tests miss.

In the next chapter we provide theoretical foundations necessary to understand this research.

1.5 Literature Survey

In this section we provide information about literature sources that were used throughout this work.

In Chapter 2 we elaborate on some theoretical aspects of this work. The materials about general aspects of software testing were mainly from [Mye04; Bei90; AO08] which are prominent books in this area. Aspects about regression testing were mainly obtained from [AO08; LH08; YH10]. [AO08] gives a general idea about what regression testing is. [LH08] presents empirical studies about how accurate random regression testing can be using Chernoff's bound concept for measuring the accuracy. [YH10] is a survey for all the existing techniques that deal with optimization of regression testing. All the information about different existing regression test selection techniques was obtained from [RH96; Rot96; YH10]. We used [Ulr02; SPL03; Fea04] for general information about legacy systems as these are the most prominent books in this area. [All09] was a good source of some statistical information about using legacy systems in modern industry. [OM07] provides a good example of a real legacy system that is still being used in NASA.

In chapter 3 we describe our problem and discuss some challenges of solving it. [AACM09; ZRW09] are really good books for a novice to get acquainted with the Oracle database. Information about the Oracle E-Business Suite was obtained from

[PA10]. Oracle patching process details on release 12.1 are well explained in Oracle's technical report [Far09].

In chapter 4 we explain the details of our solution. We basically describe every step of our five step solution. The first and third steps, namely building the access dependency graph and identifying changed entities (methods and fields of the classes) are explained in detail in [Iqb11]. The tool that is used in the second step, called LGIParser that parses Oracle's log files after patching and outputs a list of changed files, is documented in [Abd10b]. The details about Oracle's tool that applies patches and generates log files can be found in [Tuc00]. The fourth part of our solution, which is an impact analysis, is obtained from a colleague via private communication [Che11]. Details about the fifth step are explained in this work. The tool called JCrasher that we used to generate test cases is well described in [CS04]. The comparison of existing open source test case generation tools is provided in [WO09], which contributed to our decision to use JCrasher.

In chapter 5 we discuss some aspects that need to be considered in future work to make our solution better and complete. The details about different file types that Oracle patches consist of are provided in [Abd10a]. [TSC+06; SCT+08] present an approach for test suite augmentation and provide a tool that implements it. [CO04] provides an explanation of the tool that can instrument the Java code that contributes to performing test suite augmentation. Particularly this tool is used in [TSC+06; SCT+08] to instrument the code and analyze if the defined requirements are covered.

Chapter 2

Theoretical Foundations

The following sections consist of a review of some basic concepts of software testing, regression testing, regression test selection, legacy systems and their challenges.

2.1 Software Testing

We are all human beings and therefore it is natural for us to make mistakes. It is generally accepted, and also noted in [Bei90] that for programmers it is natural to introduce bugs into software during the development process. Beizer also highlights the statistic that says for every 100 statements written by a good programmer there are still 1-3 bugs. In our modern life software is used in almost everything that surrounds us. Take, for example, home appliances like microwave ovens or washing machines, cell phones; take your car that alone might have up to several dozen computers installed in it, and so on. Bugs revealed in some of those would make one feel upset, but in others it might cost human lives. And that is why software testing is an important stage of the development process that cannot be omitted or ignored. The following is the definition of software testing from [AO08]:

Definition 1. *Testing:* Evaluating software by observing its execution.

When one executes software while performing its testing, the goal should be to find bugs, not to assure oneself about the correct execution of the program. This

latter way of thinking makes the process of testing somehow destructive [Mye04], that can put a stumbling-block between testers and programmers. But when the main concern is the quality of the produced software and when both sides accept that, following the former strategy will benefit the whole company. Unfortunately, in many cases people do not differentiate testing from debugging, which leads to producing unreliable software [AO08].

Although testing is intended to find errors in software, it has been shown to be infeasible to find all of them even for trivial programs [Mye04]. The reason for this is the infeasibility of covering the whole domain of possible inputs. For example, if the function takes a real number as a parameter, it is impossible to check the execution of that function with all possible values, because it is invariably an infeasibly large set.

We need to add a notion of *test case* which represents each single test run on a software artifact using a specific set of test inputs, execution conditions and expected results. A *test suite* is a set of all existing test cases. As was mentioned above, sometimes it is impossible to cover all possible inputs, so that is why there is also a notion of *test coverage*. The number of faults we may reveal during testing directly depends on what the coverage criteria for testing is. For example, there is a node coverage criterion that requires test cases to walk through every single statement in a program. Node coverage is considered as being the weakest coverage criterion and therefore it does not give a good warranty that errors will be revealed.

There are two well-known testing strategies:

- *Black box testing* is performed without knowing internal structure of the program and so test cases are obtained from program requirements and specifications.

- *White box testing* is performed based on the source code of the program.

Black box testing is sometimes also called *data-driven* [Mye04] because essentially it is done without knowing what the implementation of the program is and its purpose is to verify if the behaviour of the program satisfies functional requirements. So the purpose of this testing strategy is to find such conditions that would make the program behave unexpectedly. Meanwhile, white box testing is performed considering

the internal structure of the program under test. It is also called *logic-driven* [Mye04] testing. So its main concern is to test a program's logic as thoroughly as possible.

Neither black box nor white box testing can provide exhaustive testing, which means there is never a warranty that the program will behave correctly and will not introduce any faults after the testing has been done. The reason is the same as that mentioned above - it is theoretically infeasible to cover the whole domain of valid and invalid input to the program.

Depending on the stage of the development process of a piece of software, these two testing strategies might both be used providing a chance that more faults would be revealed during the testing process. Let us review the so called "V model", which is a graphical representation of the different stages in the evolution of software.

Figure 2.1 shows all the development and testing activities during software

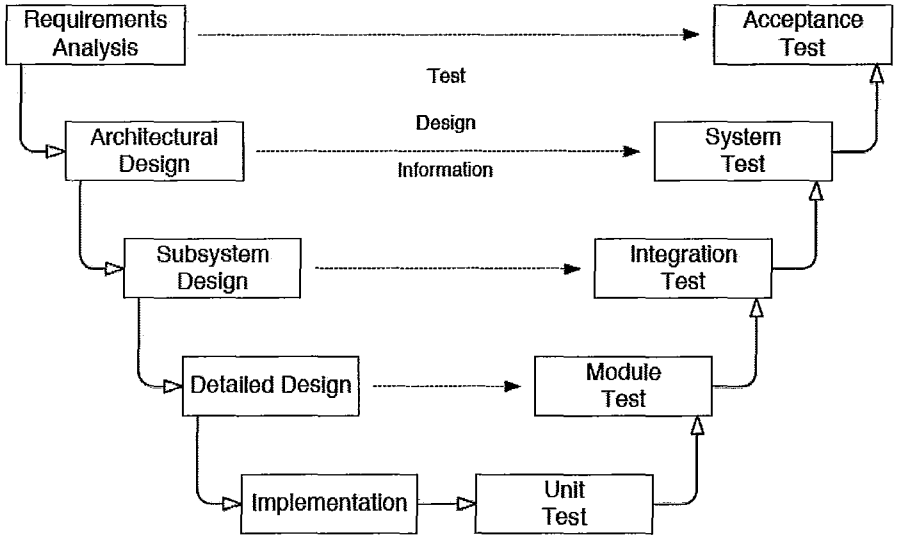


Figure 2.1: The V-model [RH96]

evolution. Specifically, this figure illustrates that for every stage of the development process there is a corresponding testing activity. It is considered good practice to follow the steps shown in the figure when developing a software. Unit testing in the figure follows a white box testing strategy, as its main concern is to test implementation of the program. And acceptance testing is essentially black box testing because

its test cases are written after the requirements and specifications of the program had been finalized, and so it does not consider the implementation at all.

The next section is about one of the variants of testing which is performed during maintenance of a software after certain changes have been introduced into it. It is called regression testing.

2.2 Regression Testing

Regression testing is done after certain changes have been introduced to a piece of software. It did not appear in the "V-model" above (figure 2.1) because it is performed during the maintenance phase of the evolution of the program. Regression testing's purpose is to verify that modified software preserves the expected behaviour and does not introduce errors. Even if the change is very small, the impact caused by that change can be very tangible and would affect the behaviour of the software, perhaps by introducing new unexpected faults.

One obvious way to perform regression testing is to run an existing test suite of the program on its modified version. This is the so called "retest all" technique. It has one drawback: during evolution of some software products, their test suites become extremely large. So it would be very costly to run the whole test suite both in terms of time and human effort, which eventually ends up being very expensive in terms of money as well. On the other hand, if only a few test cases are run, it might be insufficient for identifying many of the faults in a modified software, so the whole regression testing will fail as being unreliable. In a paper Harrold et al. [LH08] investigate a random test selection technique, where using the mathematical concept of Chernoff bound, they calculate what the number of test cases should be to gain a certain percentage of confidence in performed regression testing. This technique does not depend on the size of the test suite. But because the selection of the test cases is random, there is still a danger that some software faults might go undetected.

Even if some test cases fail during regression testing, it still does not mean that this is a fault. It might be the case that the test case is not satisfactory for the modified software as it was for a previous version. This is why, after identifying such test cases, further analysis is needed to see if those test cases are adequate [AO08].

A lot of approaches to optimizing regression testing have been invented, but there are three techniques that are considered to be most importnat [YH10]:

1. Test suite minimization
2. Test case selection
3. Test case prioritization

Following are the definitions of the problems that explain each of these techniques, taken from [YH10]:

Definition 1. *Test Suite Minimization Problem*

Given: A test suite T and set of requirements $\{r_1, r_2, ..., r_n\}$ that have to be satisfied for an adequate regression testing of the program, and also subsets of T $T_1, T_2, ..., T_n$ where all test cases of T_i satisfy requirement r_i.

Problem: Find a subset $T' \subseteq T$ where test cases satisfy all r_is.

A *test requirement* is a certain part of a software that the test case must satisfy or cover [AO08]. For the minimization problem, the set of test cases T' need to be a minimal hitting set of all T_is. The minimal hitting set problem is known as being NP-complete [YH10].

Definition 2. *Test Case Selection Problem*

Given: A program P, its test suite T and modified version of P, P'.

Problem: Find $T' \subseteq T$ with which to test P'.

Both the selection and the minimization technique's goal is to reduce the size of the test suite; however, the difference is that in the selection technique we try to select test cases that are "modification aware", which means that they are relevant to modified parts of the software and thus this technique is related to white box testing where software's source code is examined.

Definition 3. *Test Case Prioritization Problem*

Given: A test suite T, the set of permutations of T, PT and function $f : PT \rightarrow \mathbb{R}$.

Problem: Find $T' \in PT$ s.t. $\forall \big(T'' | T'' \in PT, T'' \neq T' : f(T') \geq f(T'')\big)$.

The test case prioritization technique's goal is to put test cases in an order such that this order would allow the detection of the maximum number of faults sooner. So it is assumed that, if the order is followed, then execution of test cases can halt any time and early detection of faults is guaranteed.

Since from patches we can retrieve information on which parts of the code changed and our aim is to select a subset of an existing test suite, the approach of optimizing regression testing that we used in our work is regression test selection. The next section explains the concepts underlying this approach, its properties and the different techniques on which it depends.

2.3 Regression Test Selection Techniques

As was mentioned above, regression testing is an expensive procedure. One of the approaches that aims to optimize regression testing of a software is called regression test selection (RTS). Its main purpose is to select a subset of an existing test suite which can be used to test a software instead of using "retest all" technique. If the subset is smaller in comparison to the size of the test suite, then tangible savings in testing can be achieved.

Before we describe the properties of the RTS techniques, it is worth explaining some variants of test cases that are relevant to this topic. The following formal definitions and notations are taken from [Rot96; RH96].

Let P be a program, P' its modified version and T a test suite for P. Also let S stand for specified output of P, S' stand for specified output for P' and D denote the domain of some specification. A test case t is a three-tuple $< n, i, S(i) >$, where n is a number (identifier) of the test case, i is a set of inputs and $S(i)$ is a set of specified outputs.

There is a notion of *obsolete* test cases which either specify invalid input for P' or specify an invalid input-output relation for P'. It can be formalized as follows:

t is obsolete if $i \notin D(S') \vee S'(i) \neq S(i)$

Test case t is called *fault-revealing* if it makes P' fail and thus detects a fault in P'. P' fails when, after executing t, the output of P' does not satisfy its specification.

Formally:

t is faul-revealing if $\neg Obsolete(t) \wedge P'(i) \neq S'(i)$

Unfortunately, there is no effective procedure for selecting fault-revealing test cases from T [Rot96].

t is called *modification-revealing* if it generates different outputs for P and P'. Modification-revealing test cases are formalized as following:

t is modification-revealing if $\neg Obsolete(t) \wedge P'(i) \neq P(i)$

Modification-revealing test cases are fault-revealing under the following two assumptions:

1. $t \in T$ when P was tested with t, P halted and produced a correct output.
2. There is an effective procedure for identifying obsolete test cases for P'.

Even if these two assumptions hold, there is still no effective procedure for precisely identifying modification-revealing test cases.

Another variant of test cases is *modification-traversing*. They execute new or modified code in P' or used to execute deleted code in P. These test cases can be a superset to modification-revealing test cases under the following assumption:

- When P is tested with t, all factors that might affect output of P except the code are kept constant in exactly the same state as they were when t was executed on P.

The following figure 2.2 visually illustrates the relations between different types of test cases.

As can be seen from figure 2.2 fault-revealing test cases can be both obsolete and non-obsolete. Obsolete fault-revealing test cases are those that do not halt for P', but were supposed to halt and produce a certain output. This itself is a fault and the only way to identify this kind of test case is to execute every obsolete test case to check if they do not exceed a certain time limit. If they do then they are fault-revealing. Also note that all modification-revealing and modification-traversing test cases are non-obsolete, because they specify valid inputs and valid input-output

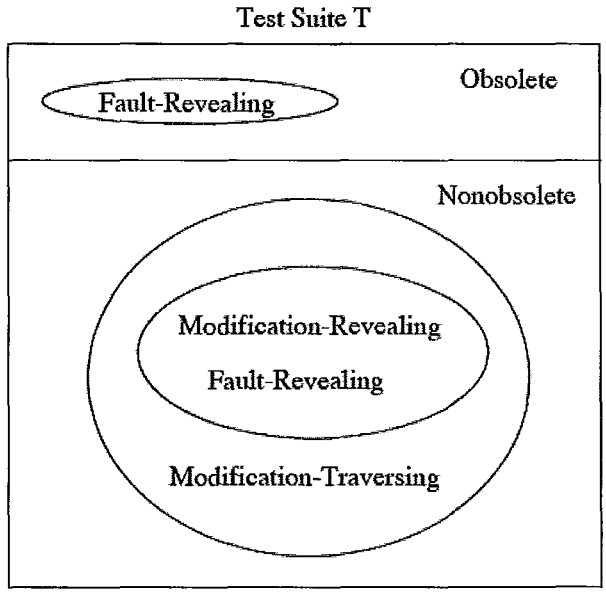


Figure 2.2: Test suite consisting of different test cases [RH96]

relation for P.

There are several properties of RTS techniques that describe their effectiveness and indicate why one would choose one RTS technique over the others [RH96; Rot96]. They are:

1. *Inclusiveness:* the extent to which an RTS technique includes modification-revealing test cases in a subset. If an RTS technique chooses all modification-revealing test cases from T then it is called *safe*.

2. *Precision:* the extent to which an RTS technique omits non-modification-revealing test cases in a subset.

3. *Efficiency:* is estimated from the computational time that a RTS technique requires. Any effective RTS technique should satisfy the following:

$$T_{all} > T_{selection} + T_{subset}$$

where T_{all} – time to run the whole test suite; $T_{selection}$ – time spent to effectively select a subset of a test suite that will test only necessary part of a software;

T_{subset} – time to run a subset of a test suite.

4. *Generality:* is the ability to handle different languages, language constructs and the ability to test real applications.

When discussing different existing RTS techniques it is very convenient to show their inclusiveness and precision properties in a graphical way such as shown in figure 2.3. In this figure circles represent modification-traversing test cases such as those that execute new or modified code in P' or used to execute code in P that was deleted in P'. Some test cases can execute some or all of them, that is why circles intersect in the figure. Dashed area represents modification-revealing test cases. Since there are modification-traversing test cases that are not modification-revealing there are areas inside of the circles that are not dashed.

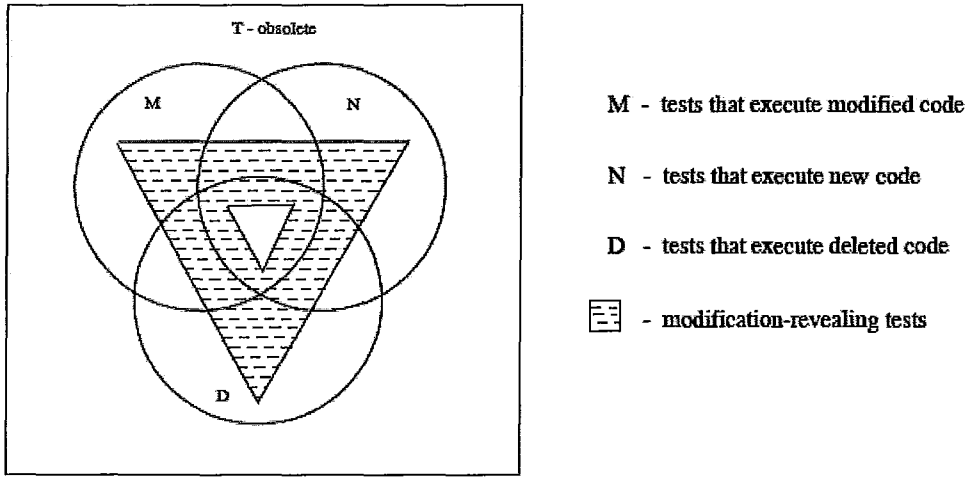
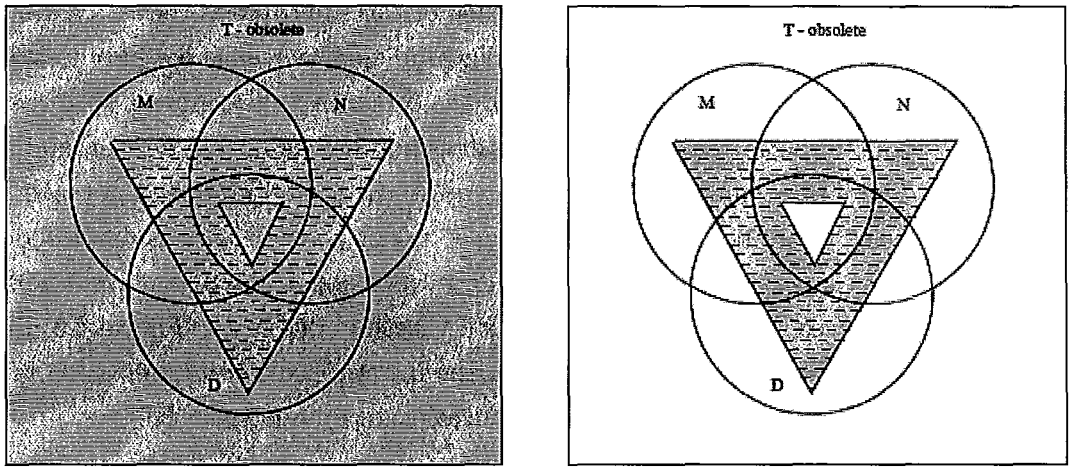


Figure 2.3: Depiction of inclusiveness and precision properties [RH96]

The following figure 2.4 depicts the worst case selection which is basically a retest-all technique, and best case selection when only modification-revealing test cases are selected (optimum). Shaded areas in the figures represent selected test cases. The A-part of the figure shows that all test cases of the test suite are chosen and thus no selection was applied. The B-part shows the RTS technique where only modification-revealing test cases are chosen, so it is safe and 100% precise.



(A) retest-all method (B) optimum method

Figure 2.4: Depiction of inclusiveness and precision properties for retest-all and optimum methods [RH96]

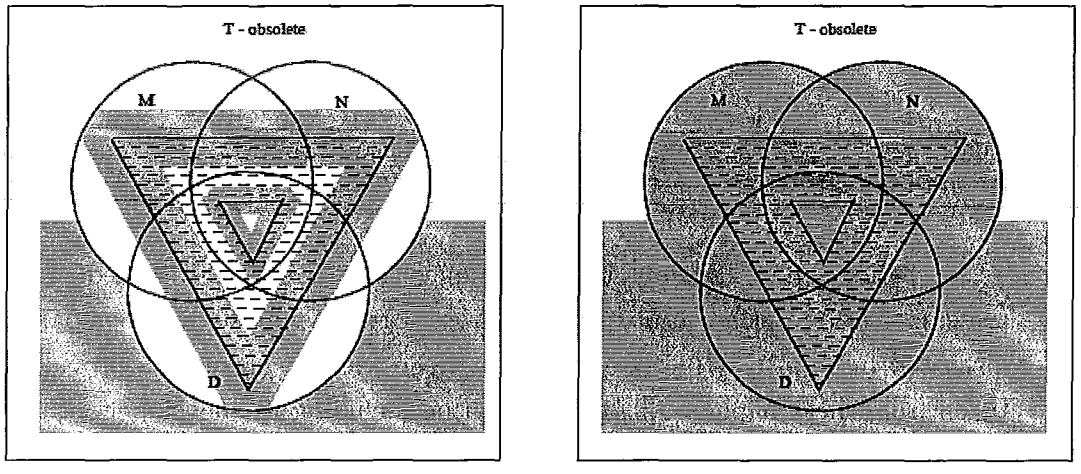
Following are different RTS techniques that have been invented since the regression testing optimization problem arose. Their analysis and comparison were discussed in [YH10; RH96; Rot96]. The description of techniques provided here is taken from those papers. All the information including the developers of techniques can also be found there.

2.3.1 Linear Equation Techniques

The linear equations technique uses systems of linear equations from matrices. These matrices represent relations between test cases and program segments, and mutual relations between program segments. Here segment means single-entry single-exit block of code with sequential execution of its statements. There are two approaches for applying the technique: intraprocedural and interprocedural. Intraprocedural technique uses a 0-1 integer programming algorithm to select the subset T' from T where for each modified segment of the program P' that does not change its control flow at least one test case is selected such that it covers every program segment that statically reaches a modified segment or that are statically reachable from it including

the modified segment itself. One drawback of this approach is that it cannot handle changes in control flow. But the interprocedural approach does not have that problem because it considers every subroutine of the program as a segment, so it essentially deals with subroutine coverage and thus does not depend on the type of modification.

The linear equation technique is essentially a minimization approach because it only selects one test case per a segment under concern as was described earlier. However, as was shown in [RH96; Rot96], it need not deal with only minimizing the set of test cases, but can also be used for selecting test cases. Figure 2.5 illustrates inclusiveness and precision diagrams for the intraprocedural minimization technique (part A) and interprocedural non-minimization technique (part B).



(A) minimization versions (B) non-minimization versions

Figure 2.5: Inclusiveness and precision diagrams for the linear equation technique [RH96]

The minimization approach of the linear equation technique omits modification-revealing test cases and thus is not safe. In both intraprocedural and interprocedural approaches the technique is not precise. The intraprocedural approach does not handle control flow modifications, and in the interprocedural approach the technique might select test cases that traverse modified subroutines but not the modified blocks of the code in those subroutines and thus is also imprecise. Considering the efficiency part of the technique, it can take a while to compute a system of linear equations for

large systems, and the 0-1 integer programming algorithm can take exponential time for computation. As for the generality aspect of the technique, it was initially created for Fortran and later for C. But it can be used for any other procedural languages as well.

2.3.2 The Symbolic Execution Technique

The technique performs test case selection using input partitioning and data-driven symbolic execution. At first a program's code and specifications are statically analyzed to identify input partitions. After that, test cases are selected that execute each input partition at least once, and it also generates test cases if some input partitions are not covered with existing test cases. Given the places where code is modified, the technique identifies edges in the control flow graph from which the modified code is reachable. After that, all test cases are symbolically executed. If some test cases were determined not to reach modified code, then they are halted. Meanwhile those that actually reach modified code are executed until their termination. Thus this technique selects all test cases that reach new and modified parts of the code.

Unfortunately the symbolic execution technique does not select test cases that used to traverse deleted code. Since these test cases can be modification-revealing, the technique is not safe. Although the technique selects all tests that reach new code, there might be an addition of new code that does not introduce a different sequence of statements for a given test case. And therefore this technique also selects non-modification-traversing test cases. These comments are depicted in an inclusion and precision diagram in figure 2.6.

Authors of the technique themselves admitted that it is very costly to run their method. For example the symbolic execution tree that is built by the technique is exponential in the size of the program itself. Therefore it requires much time to build it. Because of this, cost the symbolic execution technique is deficient in generality as well.

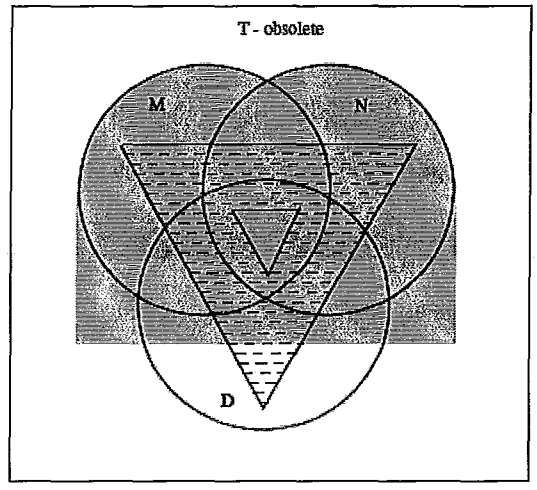


Figure 2.6: Inclusiveness and precision diagram for the symbolic execution technique [RH96]

2.3.3 The Path Analysis Technique

This technique uses a notion of *exemplar* path which represents a path from program entry to program exit without having any cycles. So for both P and P' the technique gets a set of all exemplar paths obtained from existing paths in each version of the program, represented by an algebraic expression. By comparing two sets of exemplar paths it classifies paths as new, cancelled and modified. After that the technique selects only those test cases that traverse modified exemplar paths.

The path analysis technique omits all test cases that traverse new or cancelled paths among which there could be modification-revealing ones. Therefore it is not safe. For modified paths all test cases that traverse them get selected, but the technique does not omit non-modification-traversing test cases. These inclusiveness and precision aspects of the technique are illustrated in figure 2.7.

Path analysis technique is expensive in terms of computation. The amount of exemplar paths for P and P' and for each test case in T can be exponential to the size of the programs. The technique requires programs that are written in languages that can be designed using language-independent algebraic expressions. Also it does not handle modifications where new or deleted code is introduced. Thus it lacks

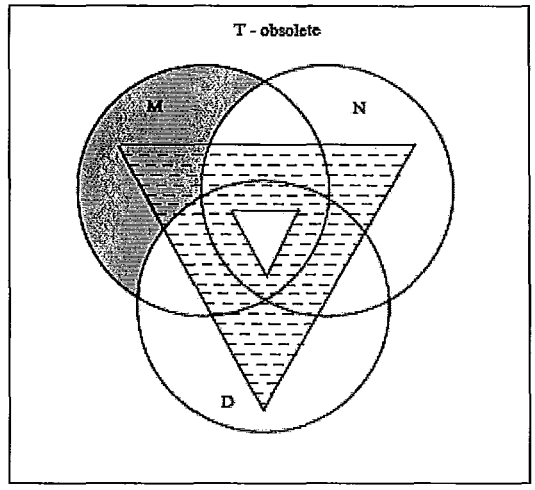


Figure 2.7: Inclusiveness and precision diagram for the path analysis technique [RH96]

generality.

2.3.4 Dataflow Techniques

Dataflow analysis relies on definition-use pairs of the variables in programs. There are several proposed techniques that perform dataflow analysis to select test cases, both on the intraprocedural and interprocedural levels. If there was a modification to a definition-use pair, then test cases that traverse that pair would be selected.

If a statement that is not part of any definition-use pair was deleted then no test case would be selected. Also, if some new or modified code that does not use any variables was introduced then again no test case would be selected. These above mentioned test cases might possibly be modification-revealing and therefore dataflow techniques are not safe. Although the techniques select all tests that reach a new, modified or deleted definition-use pair, there might be an addition of a new definition or use that does not introduce a different sequence of statements or a different definition-use pair for a given test case. And so non-modification-traversing test cases will also be selected. Therefore these techniques are not precise as well. Their inclusiveness and precision diagram is shown in figure 2.8.

When performing dataflow analysis two versions of a program are compared

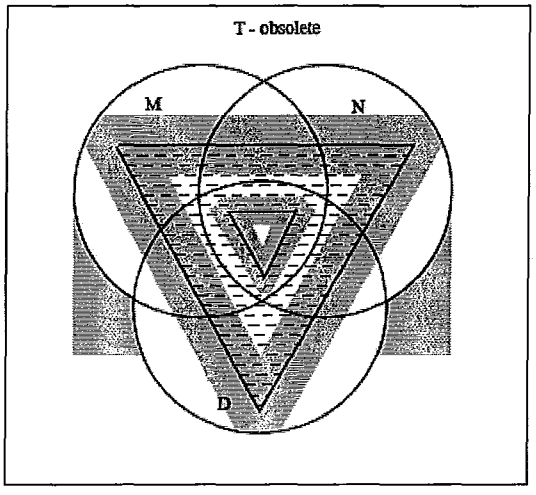


Figure 2.8: Inclusiveness and precision diagram for dataflow techniques [RH96]

for correspondence, then their definition-use pair associations are also compared. And so the time spent on the analysis is quadratic in the size of the largest among two versions of the program times size of the test suite. Dataflow techniques can be applied to any procedural languages. If there is no change in definition-use pairs then it is not the best choice to use these techniques.

2.3.5 Program and System Dependence Graph Techniques

Test selection techniques that use program dependence graph (PDG) consider two different components of a PDG: nodes and edges. Components of two versions of a program are grouped into *execution classes* via slicing so that any test case that executes a given component will certainly execute all other components of the same execution class. After that slices of P and P' are compared to identify *affected* components. Finally, the technique selects all test cases that traverse all components of execution classes that have affected components. This technique is intraprocedural.

There is also an interprocedural extension to the PDG technique that uses a system dependence graph (SDG). It uses a so called *calling context slicing* to identify *common execution patterns* in components of both versions of the program. Then the technique selects all test cases that examine those components in P that have

common execution patterns with new or modified components of P'.

Both PDG and SDG techniques select test cases that traverse new and modified code, but skip those that examine deleted code. Therefore both are not safe. With respect to their precision property, both techniques have the same problem as with symbolic execution and dataflow techniques by selecting non-modification-traversing test cases that examine new code which does not introduce a different sequence of statements for a given test case. The inclusiveness and precision diagram for PDG and SDG techniques is the same and is shown in figure 2.9.

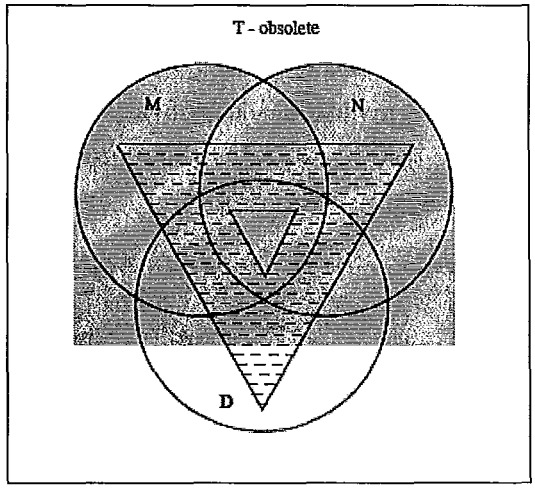


Figure 2.9: Inclusiveness and precision diagram for program dependence graph and system dependence graph techniques [RH96]

Depending on which criterion is used, whether it is PDG-node or PDG-edge, the computational cost of the whole analysis for both PDG and SDG techniques can be up to the power three or even four of the size of the program. It also requires tools for generating program dependence and system dependence graphs for P and P'. Generally these techniques can be used for any procedural languages. The major problem is that they do not deal with code deletions.

2.3.6 The Firewall Technique

The firewall technique identifies where to put a *firewall* around modified modules in the code so that only test cases that for given modified modules are within the firewall get selected. The modules that directly interact with modified modules, i.e., direct ancestors and direct descendants, are included in the firewall with modified modules themselves. The technique specifically deals with unit tests and integration tests. It performs the selection on the interprocedural level.

If all unit and integration test cases of the given test suite are reliable then this technique will select only modification-revealing test cases which makes it safe. However, as authors of the technique note, in practise existing test cases are not reliable and therefore the firewall technique will lack safety. For some modified module in a code, all test cases that examine that module will get selected, even though not all of them actually traverse the modification. Thus the technique selects non-modification-traversing test cases and is therefore not precise. Its inclusiveness and precision diagram is shown in figure 2.10.

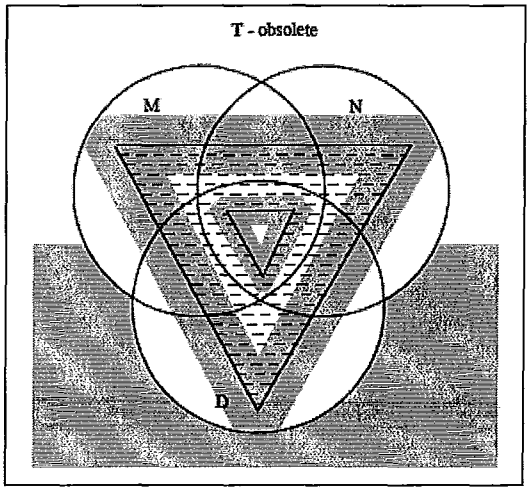


Figure 2.10: Inclusiveness and precision diagram for the firewall technique [RH96]

Performing the firewall technique's selection can take time proportional to the size of the program's call graph times the size of the test suite. This is quite efficient compared with all previous techniques. The only drawback is that it requires

a database setup which might be expensive. But on the other hand that setup is performed only at the initial phase of the analysis and it is done once. The initial firewall technique is applicable to any procedural languages. Later, there were some extensions that also deal with object-oriented languages and even GUI.

2.3.7 The Cluster Identification Technique

The cluster identification technique uses a notion of *cluster*, which is a single-entry single exit subgraph of a control flow graph (CFG) of the program. The technique compares clusters of P and P' for their correspondence and selects those test cases that examine new, deleted or modified clusters. It also selects test cases that execute the code that is control dependent on clusters with modifications.

The cluster identification technique selects all modification-traversing test cases and therefore is safe, but it is not precise. Its precision also suffers from the fact that it can also select test cases that are non-modification-traversing. The reason for this is that the technique selects all test cases that examine identified clusters, but some of those test cases might not traverse the actual modification in the code. Its inclusiveness and precision diagram is illustrated in figure 2.11.

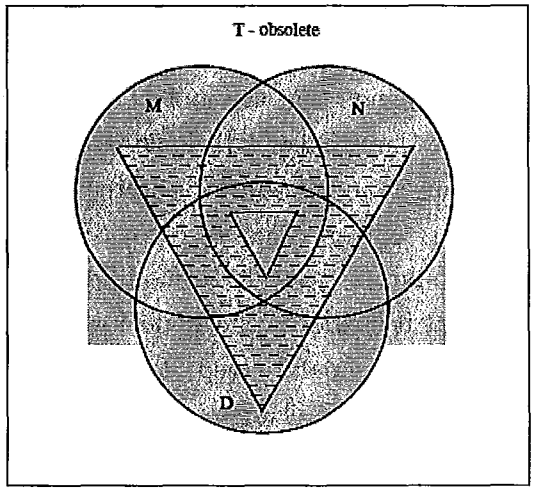


Figure 2.11: Inclusiveness and precision diagram for the cluster identification technique [RH96]

The computational time of the selection part of the cluster identification technique is proportional to the size of the largest of two versions of the program to the power of three times the size of the test suite. Since it deals with control flow graphs, it can be used for any procedural languages.

2.3.8 Slicing Techniques

Slicing techniques have four different slicing criteria for test cases. An *execution slice* includes all statements that were executed by a test case. A *dynamic slice* includes only statements that affect output of the execution slice. Thus a dynamic slice is essentially a subset of the execution slice. A *relevant slice* is a dynamic slice that includes all predicates of the execution slice that if changed could have caused a different output. And there is also an *approximate relevant* slice which is a dynamic slice with all predicates of the execution slice. For a given modified statement, depending on which slicing criterion is chosen, the techniques select those test cases that include that statement in their slice.

Unfortunately, slicing techniques cannot handle situations when there is a modification in control flow of the program. If there is no such change, then the techniques are safe. Also, since the initial variant of the technique could not deal with additions of code, there was an extension which could handle situations when a new statement or a new predicate were introduced. The problem was overcome by means of analyzing all the statements that are control dependent on a new predicate and all the statements that use the value of a variable in a new statement. Still that did not make the techniques safe enough because if a statement that does not use any variables is added, the techniques will not select any test case for that, even though test cases that examine that new statement might be modification-revealing. Another drawback of the techniques is that it cannot properly deal with multiple modifications as it analyzes each one of them in a separate step. This also decreases the slicing technique's inclusiveness. When there is no change in control flow of the program and no new code is introduced slicing techniques select only modification-traversing test cases. Since dynamic, relevant and approximate relevant slicing criteria restrict their test cases to examine the output of the slice, they exclude some modification-traversing but not modification-revealing test cases. With the extension that deals with changes

in control flow the techniques also select some non-modification-traversing test cases as well. The inclusiveness and precision diagrams of slicing techniques for all four different slicing criteria are shown in figure 2.12.

When there is no addition of a new code or change in control flow, slicing techniques take time proportional to the size of the test suite. But if such modification exists then it needs time proportional to the size of the modified program squared times the size of the test suite. And this is the time estimation for a single modification. When there are several changes then the test selection algorithm is not worth running because it would require recalculation of test traces after analyzing each modification. Slicing techniques can be applied to any procedural languages. But its generality is lacking because it deals badly with multiple modifications and changes in control flow.

2.3.9 Graph Walk Techniques

Graph walk techniques use the control flow graph (CFG) of the program to perform test case selection. Given programs P and P' and also a test trace of all test cases in T of P, the techniques compare edges of two CFGs G and G' of respective programs to identify new, modified or deleted nodes. Then techniques select all test cases that traverse those nodes. There is also an extension of the techniques that uses data dependence information for more precise test selection.

Graph walk techniques select all modification-traversing test cases and therefore are safe. As not all modification-traversing test cases are modification-revealing the techniques are not 100% precise. When an extended technique that uses data dependence information is used then some modification-traversing test cases are omitted and thus the technique becomes more precise. The inclusiveness and precision diagrams for both versions of the techniques are illustrated in figure 2.13.

Graph walk techniques can be very efficient in comparison with other techniques. An analysis and selection can take a time proportional to the size of the test suite times the least size among the two versions of the program (when data dependence information is not used). Since control flow graph and data dependence information can be obtained for any procedural language, the graph walk techniques

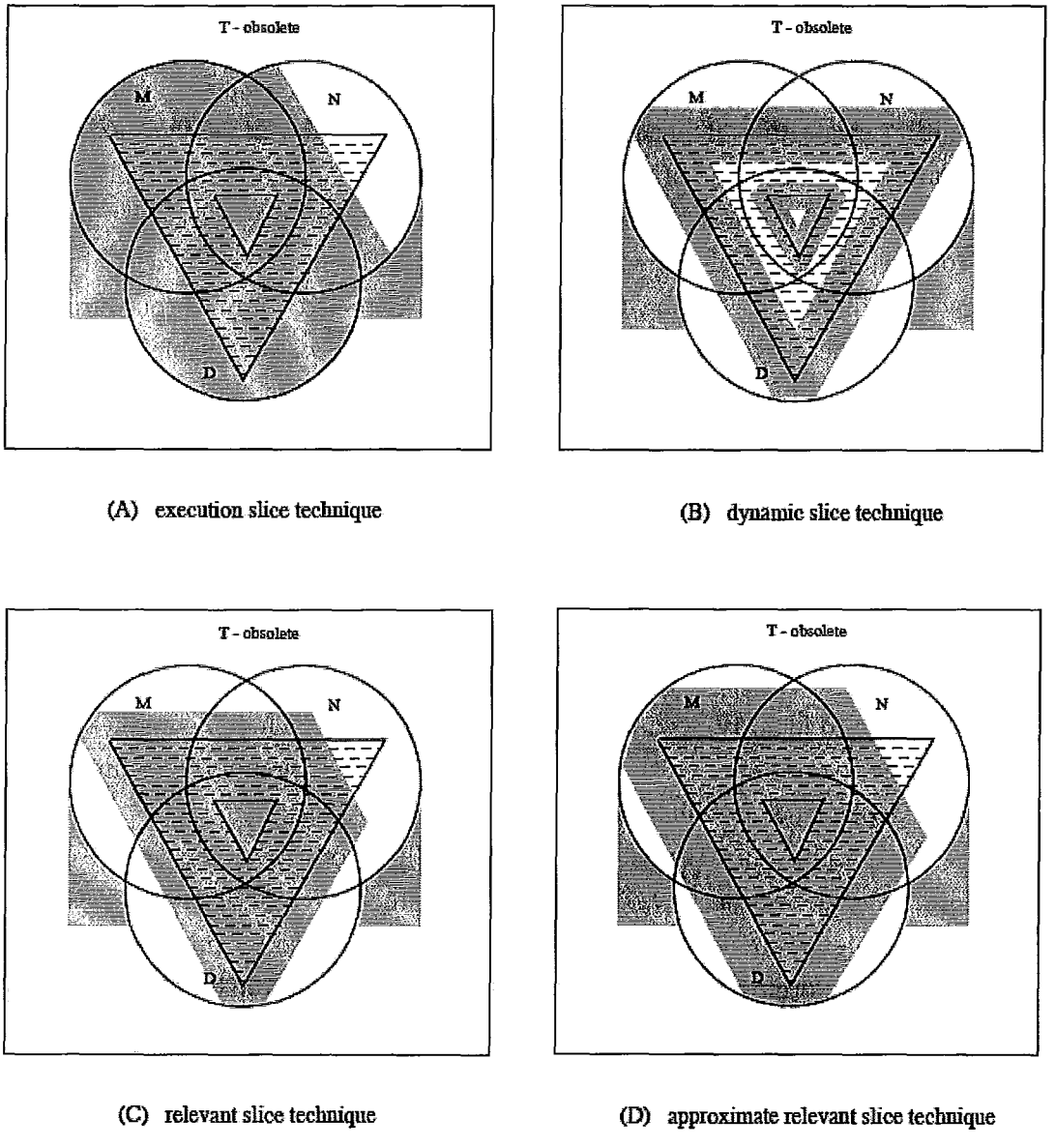


Figure 2.12: Inclusiveness and precision diagram for slicing techniques [RH96]

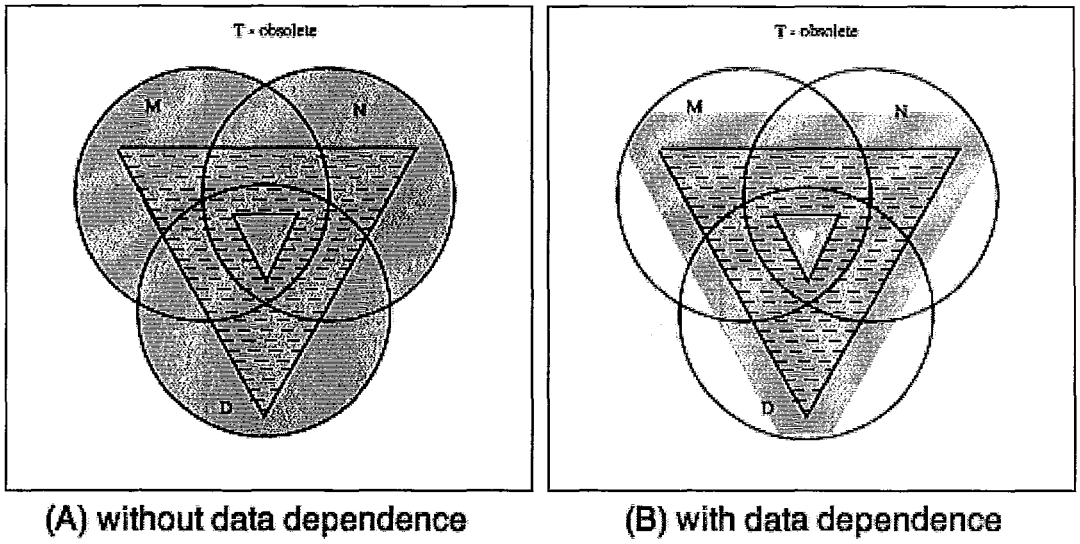


Figure 2.13: Inclusiveness and precision diagram for graph walk techniques [RH96]

can be used for any of them. Also there are extensions that can work with object-oriented languages like Java and C++.

2.3.10 The Modified Entity Technique

The modified entity technique performs test selection through identifying modified code entities and selects tests that examine those entities. A code entity can be a function or some storage locations. It performs test selection on an interprocedural level. Authors of the technique developed a tool called TestTube that implements this technique and works with programs written in C.

The modified entity technique selects all modification-traversing test cases and therefore is safe. It selects test cases that examine modified procedures but which might not examine the modified part of that procedure. That is why the technique allows some non-modification-traversing test cases as well. The inclusiveness and precision properties of the technique are illustrated in the diagram in figure 2.14.

This technique is the most efficient among all of those above mentioned. Its selection can take a time proportional to the size of the test suite times the number of modified entities in the program, which is usually less that the length of the program

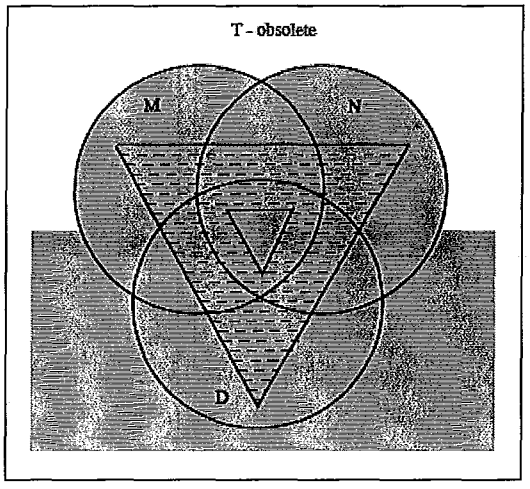


Figure 2.14: Inclusiveness and precision diagram for the modified entity technique [RH96]

itself. It can be used with any procedural language. It also requires the use of a database where it keeps the information about modified entities in the code.

The next section is about software called legacy software. We elaborate on what they are and why they are still in use.

2.4 Legacy Systems

As was mentioned in the first chapter of this work, legacy systems are those that resist modification and evolution [SPL03]. Let us examine a bit what this means.

Software can be considered legacy software for several different reasons: it was written a long time ago, it has poor documentation or does not have it at all, it works on outdated hardware, it is large and monolithic, it was written in an old language which is not in use any more, it does not meet contemporary design requirements, and so on. It might be the case that only one or few of the reasons hold for the system under consideration or it might be a combination of all of them - it is still considered a legacy system. As was stated in [All09] that nowadays there are approximately 200-300 billion lines of legacy code being used in industry. 70% of that amount is

written in COBOL - a programming language that has existed since 1959 and which is currently known by only few programmers. This statistic raises a question - why are there still so many legacy systems in use? There are several reasons for this:

- The system does its job by satisfying users' demands.
- Introducing changes to a system, whether it is a partial change in the code or a complete refactoring of the whole system, could cost a company a large amount of money and might also going to take a long time.
- Making changes to the system might be very challenging because of its monolithic and complex nature.
- It might also going to be expensive both in terms of time and money to teach the company's personnel to work with a new system.
- The system might be working on a real-time basis (like banking systems, air traffic control systems, etc.), and introducing changes to it would require a time delay in its operation which could cost a company a lot of money.
- The system is working on its own and there is nobody who thoroughly understands how it works. It might be the case that it was not fully documented or the documentation was lost.

One good example of an existing legacy system is provided in [OM07] which talks about NASA's [1] flight control system which mostly consists of software that was written in the 1970s. While it was being written, it went through a huge amount of testing and certification before it actually started to be used. So definitely the system is old and it requires a lot of upgrades. But the main problem is that any kind of change to that system would require conducting a huge amount of testing and certification again, and by the time the whole process ends the system would already be considered as being legacy.

The next chapter is about problems associated with using legacy systems.

[1] National Aeronautics and Space Administration

2.5 Challenges of Legacy Systems

M. Feathers [Fea04] defined a legacy system as software without tests. We believe he does not mean that all software that has tests cannot be called legacy. But rather he means that if software has tests, we can make changes and verify if the behaviour changes or not. Whether those tests that come with the software are adequate and thorough is another question. It might be the case that none of the test cases would be able to examine newly introduced changes and therefore there is no warranty that the software will behave as expected. Unfortunately, there are many more difficulties around legacy systems other than just an absence of tests.

In his book Ulrich [Ulr02] presents a list of disadvantages of legacy systems:

- Understanding the system's functionality is hard and time consuming.
- Hard to identify business logic from the system's data logic and presentation.
- Difficult to find technical and functional documentation.
- Difficult to integrate with the legacy system because of software and hardware incompatibilities.
- Hard to perform modifications because of the lack of documentation and regression tests.
- There is no data integrity and consistency because of using an Information Management System (IMS) or Integrated Database Management System (IDMS) which does not support a relational database.

We can add to this list more disadvantages:

- Hard to find spare parts for the hardware which legacy systems run on.
- It might be less expensive to completely change both software and hardware than to keep maintaining the legacy system.
- After having decided to change or modify an existing system, it usually does not meet deadlines and costs much more than what was budgeted in the beginning.

In general, maintaining a legacy system is itself a big challenge. Companies trying to keep up with always emerging business requirements and practises will eventually need to make changes in their legacy system. Seacord mentions Lehman's first law to justify this fact [SPL03]: "A large program that is used undergoes continuing change or becomes progressively less useful". There are four different types of changes that can be made to legacy code [SPL03]:

1. **perfective** – adding new features, performing enhancements to the system.
2. **adaptive** – performing upgrades to the system, e.g., changing the operating system or the database management system.
3. **corrective** – fixing bugs in the system.
4. **preventive** – these changes are made to improve future maintainability and reliability of the system.

Depending on the complexity and source code availability of the legacy system, different types of those changes can be performed. There is one approach that is called the "big bang" approach that implies changing both hardware and software while preserving existing business requirements. For large and complex systems, most of the time this approach is not worth doing. It is very costly both in terms of money and time. What companies have practiced so far and gained certain results in modernizing legacy systems is an incremental modernization approach when the focus is to perform step by step modernization of separate artifacts of the system.

2.6 Summary

In this chapter we provided some background on topics such as software testing, regression testing and legacy systems that this work is mainly focused on. Software testing is a crucial part of the development process of any kind of software. As for the regression testing we showed that there are three popular approaches of optimizing the regression testing process: test suite minimization, test case selection and test case prioritization. In our work we chose to use the test case selection approach.

We provided some details about some aspects of regression test selection and different types of regression test selection techniques. As for legacy systems, we explained what they are and why they are still in use. We also provided challenges that are associated with using legacy systems.

In the next chapter we provide the description of the problem that we faced in our project in detail. Also we are going to discuss the challenges associated with solving that problem.

Chapter 3

Problem Description and Challenges

This chapter is devoted to a description of the problem addressed in this thesis and also the challenges faced tackling the problem.

3.1 System Under Test (SUT)

In this section we will provide the description of the problem by explaining three layers that our system consists of, their mutual interrelations and the patching process that is performed on a system.

3.1.1 Three Layers of the System

Our system consists of three layers which are the Oracle database, the Oracle E-Business Suite and the customer's application. Following are some details about each of these layers.

Oracle Database

In the beginning the company was called Software Development Laboratories which in 1979 produced the first commercial SQL relational database management system (RDBMS) [AACM09]. Now Oracle is one of the biggest computer technology

corporations in the world. Oracle is especially known for its RDBMS of the same name. The latest Oracle database version is 11g, and was introduced a few years ago. Oracle is the most used and sold database in the world [ZRW09].

Oracle is a relational database management system, where *relational* means that it uses tables to store the information. Tables are related with each other through primary keys and each table consists of columns. The relational model fully replaced the formerly used hierarchical data model. Data in Oracle 11g has the following attributes [AACM09]:

- **Data types** – defines the type of the data, e.g., numeric, string, date, etc..
- **Column size** – defines the maximum size of the data in a certain column.
- **Ownership** – defines who is the owner of the information.
- **View and manipulation rights** – defines who has the right to view the data and which users can perform which actions on it.

Other than tables Oracle 11g consists of stored program objects such as [AACM09]:

- **Views** – these are the predefined subsets of the data. SQL statements that define views allow us to see only the necessary amount of information that was filtered by certain constraints.
- **Triggers** – these are the stored objects that are executed after a certain event occurs. Each trigger is bound to only one table and each table can have several triggers.
- **Procedures** – these are stored program units that can accept parameters and interact with Oracle objects.
- **Functions** – these are like procedures but they return a value to the code that they are called from.
- **Packages** – these merge procedures and functions together in one programming unit.

Triggers, procedures and functions are written in the PL/SQL procedural programming language that has been developed by Oracle to operate with the Oracle database.

The letter "g" in the version of the database stands for *grid* and was first introduced with Oracle version 10g. Grid is a kind of distributed system that works with large numbers of files. Grids are normally more loosely coupled, heterogeneous and geographically more spread out, which distinguishes grid computing from other high performance computing systems [ZRW09].

Oracle E-Business Suite

The Oracle E-Business Suite (EBS) is a software package that allows organizations to manage most of their business processes. A big advantage of having such a system deployed is that it provides centralized data storage in conjunction with integrated software applications that have a consistent look and feel.

EBS consists of several product lines such as [PA10]:

- Financials
- Procurement
- Customer Relationship Management (CRM)
- Project Management
- Supply Chain Planning and Management
- Discrete Manufacturing
- Process Manufacturing
- Order Management
- Human Resources Management System (HRMS)
- Applications Technology

Each of the listed product lines consist of several applications, e.g., Oracle Financials consists of General Ledger, Payables, Receivables, Cash Management, iReceivables, etc.

The EBS release that we are going to talk about in this work is R12, the latest release. In R12 there are several different user interfaces, such as forms, an HTML-based interface and also a mobile interface to use from PDA devices. Each of these interfaces have their own features. Forms work better for people that work with the EBS intensively and operate on a lot of data. The HTML-based interface works well for people who infrequently use the system.

The EBS also has development applications like Oracle Forms and Oracle Reports that enable the creation of new objects (forms, reports, concurrent programs) or customizing existing ones. An illustration of components of the EBS is provided in figure 3.1.

R12 is based on a three-tier architecture [PA10] which consists of the:

- **Client tier** – this is the user interface layer, i.e., screen representation of the system.
- **Application tier** – also called middle tier, it contains business logic that operates on data. It is a bridge between the client tier and the database tier.
- **Database tier** – this is the layer that stores and retrieves the data.

The following figure 3.2 illustrates the three-tier architecture of the EBS.

Customer's Application

The customer's application in our case is the legacy system because of the following reasons:

- There is a lack of technical and functional documentation that would help understand its structure and requirements.
- It is currently being used by the customer on a daily basis and represents a real-time system.

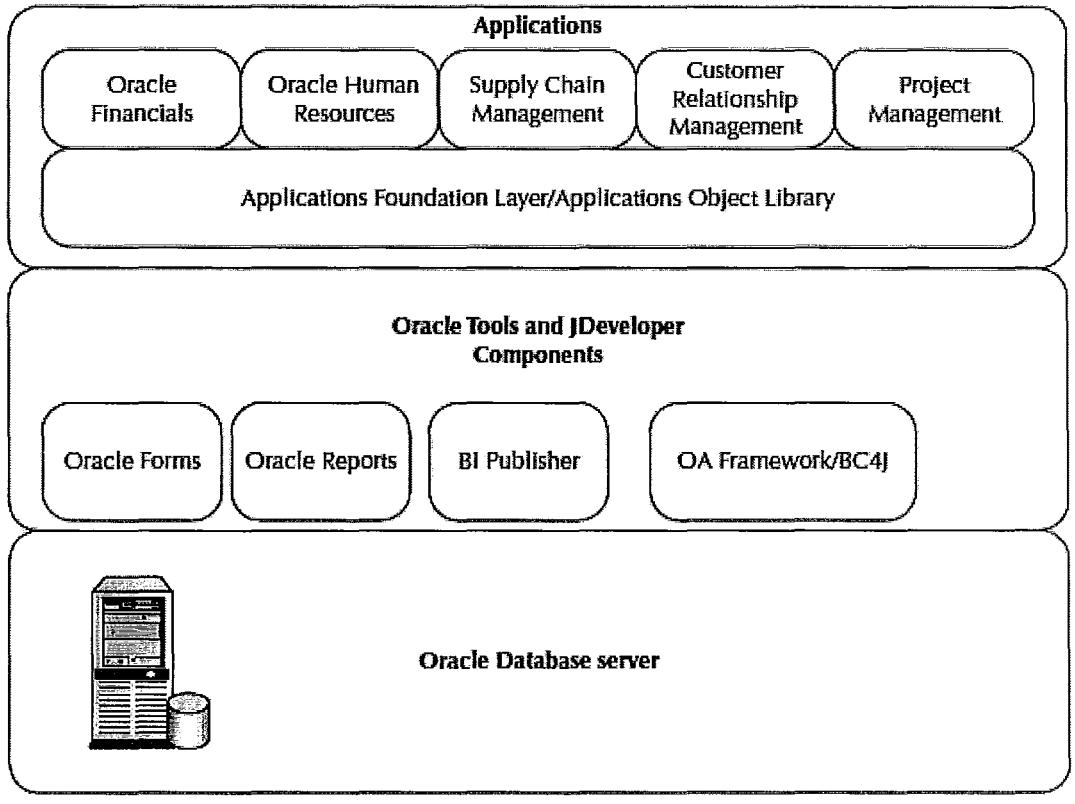


Figure 3.1: Overview of the E-Business Suite applications [PA10]

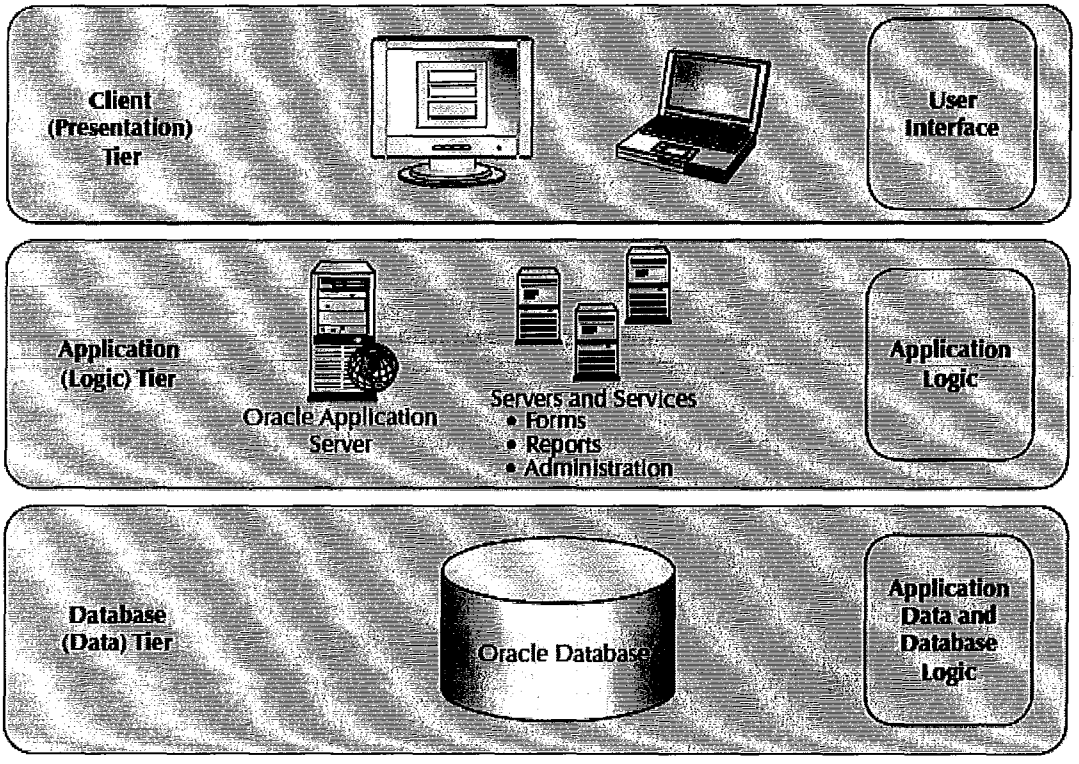


Figure 3.2: Three-tier architecture of the E-Business Suite [PA10]

What is known about the application is that it is written in Java and is built on top of the E-Business Suite and Oracle database. It has a huge test suite that consists of many regression tests. Currently, running the whole test suite after a certain modification is costing the customer a very large amount of money and takes a lot of time. So definitely there is a critical need for efficient and safe ways of testing it.

3.1.2 Interrelations between Layers

The high level representation of our system that consists of three layers is illustrated in figure 1.1. The relation between the Oracle database and the EBS is provided by Oracle itself and is depicted in figure 3.2 where the client tier and application tier together represent our EBS layer. On top of the Oracle database and the EBS layers there is the customer's application layer.

In our three-layer system there is a "top-down" dependency relation which denotes the following:

- The EBS accesses database objects, e.g., Java methods retrieve data from the database through embedded SQL statements.
- The customer's application accesses the EBS stored objects, e.g., method in the customer's application calls a Java method in the EBS.
- The customer's application accesses database objects, e.g., a method in the customer's application stores data in the database through embedded SQL statements.

So essentially our "top-down" dependency relation means that the database objects do not access the EBS stored objects or the customer's application methods or the EBS stored objects do not access the customer's application methods. However, there might exist internal relations in each of the layers, e.g., among the EBS methods.

3.1.3 Patching Process

Periodically throughout the life cycle of the EBS, Oracle releases *patches* to maintain the system. There are several reasons for this [Far09]:

- Fixing an existing error
- Introducing a new functionality or a feature
- Upgrading to a higher maintenance level
- Applying latest enhancements
- Determining the source of an error
- Applying online help

Patches can change both the EBS and the database objects. They differ not only in size but in their format, which can be as follows [Far09]:

- **Individual bug fix** – a patch that fixes an existing error
- **Product family release update pack** – a pack of patches for all products of a specific product family.
- **Release update pack** – a pack of patches for the whole EBS.
- **Pre-upgrade patch** – all upgrade-related high-priority patches for all products of a specific product family.
- **Consolidated upgrade patch** – all upgrade-related patches for all products of a product family. These patches are only available for release R12 to upgrade from one point release [1] to another.

There are several tools provided by Oracle for applying the patches. There are ones that can be launched from command line, also there are tools that are Web-based. Each patch comes with a "readme" file that contains information like what the patch is for and how it should be applied. Some patches require manual steps and some can be automatically run using patch application tools.

[1] Here point release means an upgrade of one specific release of the EBS, e.g., R12.1, R12.2, etc.

3.2 Problem Description

As was mentioned before, the customer's application is dependent on both the EBS and database. Currently after each Oracle patch, the customer runs the whole regression test suite in order to verify that the functionality of the customer's application did not change. As the test suite is very large, it is very expensive both in terms of time and money to perform such regression testing. Considering that patches are being released quite regularly (particularly at the company we were working with, patches were applied each quarter) it becomes incredibly expensive to keep testing the application that way. In this case, regressing testing definitely needs an optimization which if proved to be efficient would save the customer a lot of time and would significantly lower their budget for testing.

3.3 Challenges

This section is about the challenges that are associated with the problem. First we are going to discuss the availability of the system that we were dealing with, then some aspects of the Java language that needed to be taken into account. We are also going to discuss the test suite aspects and some considerations about preciseness and inclusiveness of RTS techniques.

3.3.1 Availability of the SUT

Our SUT is a three-layer system with the customer's application on top. It is necessary to note that as for the EBS and Oracle database layers it was more or less clear how they work and what kind of relations between them might exist. There were enough materials both from Oracle Corporation and other sources about the architecture and functional aspects of those objects. But we did not have any kind of information about what the customer's application looked like. All we knew was that it was built on top of the EBS and the database and serves as a "wrapping" program. The reason for the customer having this kind of application could have been the need for additional functionality or a specific look and feel or both. So needless to say that there was no clear picture of the whole system.

So we did not have any choice other than make assumptions regarding what the customer's application might look like. The assumption was that it is fully written in Java and has only binaries without any source files. That partially reflected the EBS as it also mainly consists of Java binaries.

3.3.2 Existing Test Suite Representation

As we did not know what the customer's application looked like, an additional challenge was to identify what kind of test suite the customer had. All we knew was that they have a regression test suite which they currently run via the "retest-all" method each time modifications are introduced to the system. This is a really important aspect of our work, because it actually defines which method of optimization to use to reduce the cost of regression testing for the customer. Considering the application is written in Java, the test suite might be as follows:

- Test suite that consists of test cases that test requirements of the application.
- Test suite that consists of test cases that test the relations between different classes.
- Test suite that consists of test cases that test the classes.
- Test suite that consists of test cases that test methods of classes.

It was decided to work with unit tests (those that test methods) in our research. After identifying which entities in the EBS and the database changed we need to be able to track back to the customer's application methods using dependency information and then select necessary unit test cases.

3.3.3 Specifics of OOP Languages, Particularly Java

In our work we built an access dependency graph (ADG), which is a method-level control flow graph (CFG), but with the addition of fields of the classes that we have in the EBS. The techniques for building the CFG, like ones that are used for procedural languages like C and Pascal, cannot be used for Java or any other object-oriented programming languages because procedural and OOP languages use

different concepts. OO programs consist of classes that encapsulate not only methods but also fields and there are aspects like inheritance and polymorphism that need to be considered when building a CFG. Let us elaborate on some details of this statement by providing an example.

In the following figure 3.3 two versions of the program are shown. P' here is the modified version of P.

Program P

```
1:  public class SuperA {
2:      int i = 0;
3:      public void dummy() {
4:          System.out.println(i);
5:      }
6:  }

7:  public class A extends SuperA {
8a:     public void dummy() {
8b:         i++;
8c:         System.out.println(i);
8d:     }
8e: }

9:  public class SubA extends A {
10:     public void dummy() {
11:         i = i + 2;
12:         System.out.println(i);
13:     }
14: }

15: public class B {
16:     public void bar() {
17:         A a1 = LibClass.getAnyA();
18:         a1.dummy();
19:     }
20: }
```

Program P'

```
1:  public class SuperA {
2:      int i = 0;
3:      public void dummy() {
4:          System.out.println(i);
5:      }
6:  }

7:  public class A extends SuperA {



8:  }

9:  public class SubA extends A {
10:     public void dummy() {
11:         i = i - 2;
12:         System.out.println(i);
13:     }
14: }

15: public class B {
16:     public void bar() {
17:         A a1 = LibClass.getAnyA();
18:         a1.dummy();
19:     }
20: }
```

Figure 3.3: Example of two versions of the program

There are two changes in P', namely the statement on line 11 and a deleted method in class A on lines 8a-8e. Suppose now that we only have a change on line 11. We are safe with this change (known as statement-level modification [OSH04]) because every test case that traverses method dummy() in class SubA will reveal a fault if there is any. But the second change is a more interesting case (also known as a declaration-level modification [OSH04]). The statement on line 17 represents an initialization of a variable "a1" to the result of static method getAnyA() of class LibClass where LibClass is a library class which has a static method getAnyA() that

randomly returns an instance of either SuperA, A or SubA. So if we choose only the test cases that examine method dummy() of class A we might omit the situations when method getAnyA() returns an instance of class SubA or SuperA since these classes also have method dummy(). SubA overrides this method and SuperA is the superclass that defines method dummy(). Therefore test cases that instantiate variable "a1" to instances of SubA and SuperA would not be selected and thus the technique would be unsafe. This is a dynamic binding problem and the solution would be to select all test cases that examine method dummy() of SubA, A and SuperA.

Assume now if the initialization of variable "a1" was: "A a1 = new A()". In this case the type of the variable is always A. But in P' class A does not define method dummy() explicitly any more, it only inherits it from SuperA. Thus selecting only the test cases that examine method dummy() in A would miss possible errors, because method dummy() of SuperA would be called. This is an inheritance issue and it can be solved by selecting test cases that examine method dummy() of both A and SuperA.

In order to overcome these polymorphism and inheritance issues of OOP languages when building an ADG, we need to analyze the whole hierarchy of each class or interface. So in the example above all test cases that examine classes SuperA, A and SubA should be selected. Also we need to select test cases that examine method bar() of class B because it has an explicit call to the modified method. In this way we ensure safety of the selection technique that we are about to propose. This method is also known as *static analysis* of the code under examination, i.e., we are analyzing the worst case scenario of the code execution and therefore do not miss any necessary test cases.

3.3.4 Trade-Off between Inclusiveness, Preciseness and Efficiency of RTS Techniques

As was shown in Chapter 2 none of the existing RTS techniques are safe and 100% precise. In order to be like these, the technique needs to select only modification-revealing test cases. Although this might be a possible aim, there is an issue of efficiency of the technique. For instance, for safe techniques, increasing their precision requires more time for analysis which decreases efficiency. For the techniques that are

not safe, increasing either precision or inclusiveness also decreases efficiency of those techniques. It might be the case that by means of increasing a certain technique's precision or inclusiveness the time that is spent on analysis gets bigger than the time spent to perform the "retest-all" method of regression testing. And that is not worth it [RH96].

In our work we came up with a safe technique of selecting test cases (shown in the next chapter) that selects all modification-traversing test cases. It might seem to be inefficient for large systems, but for our three-layer system where regular patches usually change only a bit of code in the EBS, our technique is actually much more efficient than the "retest-all" method. Some empirical data will also be provided in the following chapter.

3.4 Summary

In this chapter we elaborated on the details of our system of concern which is a three-layer system consisting of the Oracle database, Oracle E-Business Suite and the customer's application. The legacy code in this case is the customer's application which is dependent on the other two layers. We also showed how patches are being applied to EBS that can affect the customer's application functionality. Before tackling the problem in the next chapter, we went through several different challenges that one would face in solving the problem. We are going to go through them again explaining how we handled those challenges in our solution in the next chapter.

So the next chapter is about the details of our solution to a given problem. It consists of several steps and we are going to present and explain each one of them in detail.

Chapter 4

Proposed Solution Description

In this chapter we present the details of how we solved the problem. As was indicated in Chapter 1 our solution is partitioned into 5 consecutive steps. First we build an access dependency graph (ADG). Then we analyze a patch to identify what parts of the system will be changed. From that we obtain information about which entities (methods and fields) were modified in the changed files. By knowing this we can trace back to the customer's application methods via our impact analysis. And for our last step we select only the test cases that examine those methods and also output names of the methods that do not have tests. All these steps are described below.

4.1 Building Access Dependency Graph

To determine if patches affect functionality of the customer's application we need to know the dependencies between the layers in our three-layer system. Because in this work we are not considering the database layer, we are only concerned about relations between the EBS and the customer's application. We also mentioned that we are specifically concerned about class files, i.e., Java binaries. So we obviously need a tool that would parse all those binaries to retrieve their dependency information.

In [Iqb11] Iqbal proposes to build an access dependency graph (ADG) which is basically a control flow graph, but with the addition of the fields of the classes.

The reason for including the fields is because in classes methods can access fields of the classes and changes can occur in fields as well. This procedure for building ADG is explained in detail in [Iqb11] and here we provide only a brief explanation of its steps.

The idea is to build ADG for all classes of the EBS and the customer's application and save it in a database for later use. It is the most expensive procedure in our overall analysis, but it need be done only once for the first application of regression testing. Following testing processes would require only a modified ADG with corrections after the last analysis. So the process begins by (1) finding class files and converting them to an appropriate XML format; (2) parsing XML files to identify all existing entities (methods and fields) and the relations between them. The following are some details about each of these steps.

4.1.1 Converting Binaries into XML

In [Iqb11] Iqbal uses a tool called *ClassReader* of the *Dependency Finder* toolset [Tes10] that takes any class file (Java binary) as an input and outputs an XML file that consists of information about all attributes of that class. That information is essentially the same as for the Java bytecode. ClassReader is called from our program for each class file in the system, even for those that are contained in jar and zip files specified in "classpath" environmental variables. All XML files are saved into a user specified directory under the same path hierarchy that corresponds to ones that the class files have in the system.

4.1.2 Building ADG

Building the ADG is accomplished using a tool written by Iqbal [Iqb11] which is called the *Dependency Analyzer*. This tool builds the graph by parsing all the XML files that were created in the previous step. First, it creates nodes by filling in an ENTITY database table, which consists of all methods and fields of all classes. Secondly, it creates edges by filling in a DEPENDENCY_RELATION database table which consists of all dependency relations between the nodes. These dependency relations are ones that satisfy the following conditions [Iqb11]:

- For classes A and B (B can also be an interface), if method *a()* of class A calls method *b()* of class B using *invokeinterface, invokestatic, invokespecial* or *invokevirtual* [1], then the following edge is added to ADG:
  $A.a() \rightarrow B.b()$

- For classes A and B (B can also be an interface), if method *a()* of class A calls method *b()* of class B using *invokeinterface* or *invokevirtual* and B has transitive subclasses or implementations (in the case when B is an interface) $B_1, B_2, ..., B_n$ that override or implement *b()*, then the following edges are added to ADG:
  $B.b() \rightarrow B_1.b()$
  $B.b() \rightarrow B_2.b()$
  ...
  $B.b() \rightarrow B_n.b()$
  If method *b()* is inherited by class B and there is no overriding of *b()* in B then the following edge is also added to ADG:
  $B.b() \rightarrow S.b()$
  where S is a direct superclass of B.

- For classes A and B, if method *a()* of class A accesses field b of class B then the following edge is added to ADG:
  $A.a() \rightarrow B.b$
  If b is a static field then the following edge is also added to ADG:
  $A.a() \rightarrow B. < clinit > ()$
  where $< clinit > ()$ is a bytecode method that contains static initializers of the given class.

By considering all above mentioned cases of relations and via obtaining the nodes for every single entity in every class file (even those that are stored in jar and zip files) we statically build a graph using conservative analysis. This graph will be used later for impact analysis and safe regression test selection. To show what ADG looks like using an example, let us take already used code snippets from figure 3.3

[1] Java Virtual Machine specifies that a method can be invoked in these four different ways; which way is certain method invoked would be specified in our XML files

and build ADGs for both versions of the program.

For P, the ADG is shown in figure 4.1. And for P', the ADG is shown in figure 4.2.

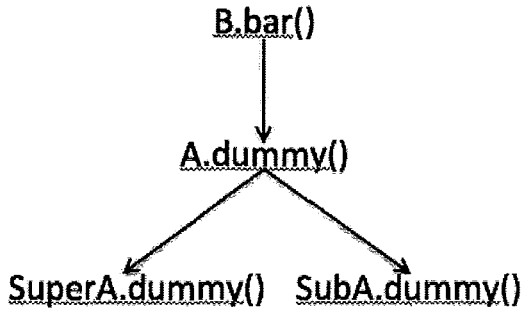


Figure 4.1: ADG for program P from figure 3.3

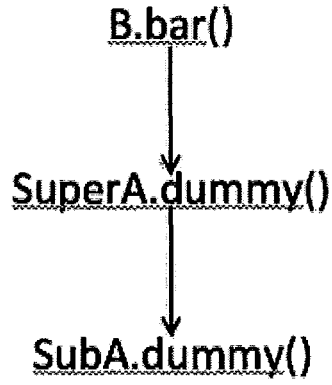


Figure 4.2: ADG for program P' from figure 3.3

The ENTITY table assigns each method and field a unique number which is then used in the DEPENDENCY_RELATION table. So instead of using names with paths of methods and fields, we use identification numbers for them to describe their interrelations (edges). By doing so, we simplify and also speed up our impact analysis process that is going to be described in further steps.

4.2 Patch Analysis

Oracle patches can introduce new files or change existing files or both. This can be observed using such interactive tools as Patch Wizard that comes with the EBS and allows us to see the details of what kind of changes the patches introduce. Because the goal of this research is to select test cases from an existing test suite, it is out of the scope of this work to analyze newly introduced files of patches. We are only concerned about changed files of the EBS after patching. As far as we know, Oracle patches do not delete their existing files, so we do not have to consider that scenario.

There is an Oracle tool called *Auto Patch* [Tuc00] that can be used from the command line to apply patches. Its advantage over other tools is that it has two modes of execution:

1. Normal mode
2. Test mode

When used in normal mode the Auto Patch actually applies the patch. But in the test mode, a patch does not get applied, rather it is analyzed to retrieve information on what modifications would have been done to the system if the patch was applied. Both in normal mode and test mode Auto Patch outputs two log files:

- The one that has the **"log"** extension contains information about the process of running the tool. It shows all the steps that were taken during the execution, such as the user's answers to the Auto Patch's prompts.
- The one that has the **"lgi"** extension contains information on which files from the patch were applied and which were not.

The latter log file with the "lgi" extension is the one that we are interested in. It contains information on which existing files changed, as shown in figure 4.3.

For each changed existing file, the "lgi" log file has two lines: one saying "Backing up ..." shows that the existing file in the system is being backed up for cases when we need to roll back; and the other saying "Copying file ..." shows that the new file from the patch is going to overwrite the existing file. This log file can be parsed

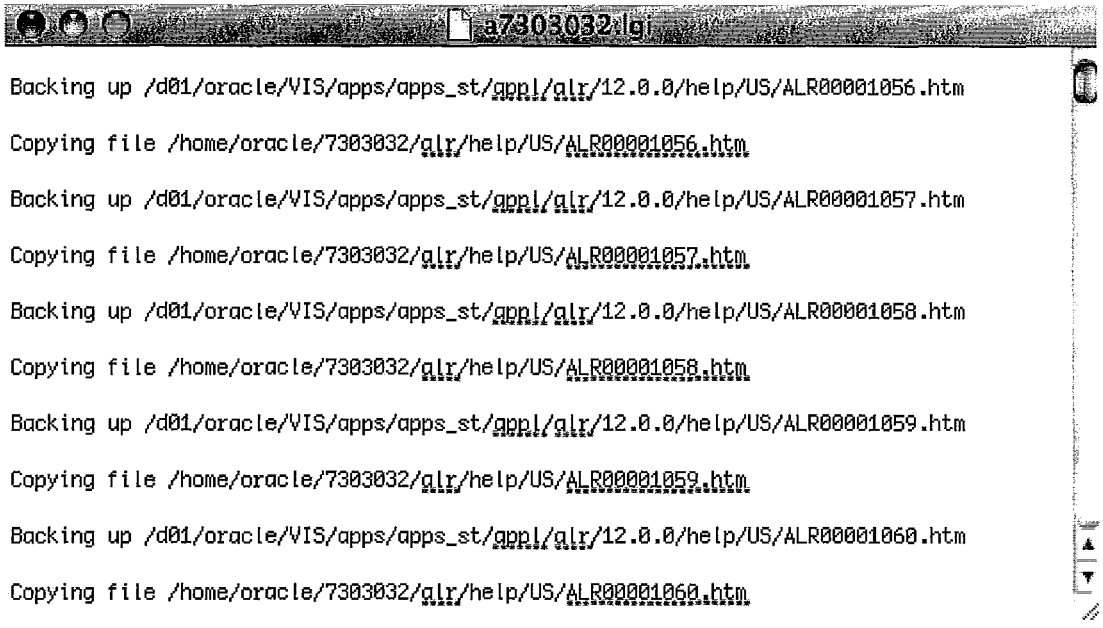


Figure 4.3: Snapshot of sample "lgi" file contents

to get the list of all files that were changed. The tool called LGIParser [Abd10b] does this job. It parses a certain "lgi" file and outputs the names of the files and their paths in the EBS and the patch to the database table called AA_CHANGED_FILES. This table will be used to identify changed entities in the following step.

4.3 Identifying Changed Entities

As for the first step where we built our ADG, we used only one version of the system, which is before applying a patch. Now we need to identify those methods and fields that changed after applying a patch. For that we parse the AA_CHANGED_FILES database table that was filled in during the previous step to get names and locations of the files that the patch changed. There are two different locations (paths): one is the location in the EBS, the second is the location in the patch. By knowing this information we can get both versions of all changed files without applying a patch and using additional resources for keeping old files.

There is a tool that was written by Iqbal [Iqb11] called ClassDiff that takes

as an input two versions of the class file, analyzes them to identify changed methods and fields and then outputs those entities to an XML file for that specific class file. Changes can be of the following type [Iqb11]:

- Changes in a super class that the class inherits from;
- Changes in interfaces that the class implements;
- Changes in an access flag of the class;
- Changes in methods;
- Changes in fields.

By using the tool we obtain XML files for all changed class files in the EBS that contain information on what entities changed. Knowing that helps us to further analyze using the ADG what entities are affected by the changes.

4.4 Impact Analysis

The goal of impact analysis is to trace back through the edges of the ADG and identify all methods that are affected by the changes in the EBS. It is worth noting that fields of the classes can only be leaves in our ADG, i.e., they have no edges from them to other entities. The reason is that changes to a field can be changes in access flags, changes of its type or deletion of a field. If changes are in their initialization then those changes would appear in a <clinit>() method if the field is static and in a <init>() method if not. Those two methods are obtained from the Java bytecode and will appear in our ADG as separate methods for a given class [Iqb11]. Therefore, all affected entities will be methods.

Chen [Che11] built a tool called the InversePathGenerator that takes as an input XML files that were obtained from the previous step, which contain information about changed entities in the EBS, parses them and, using a depth-first search algorithm, identifies what methods are dependent on those changed entities, up to the methods in the customer's application. The next figure 4.4 illustrates the process of identifying affected methods in the ADG.

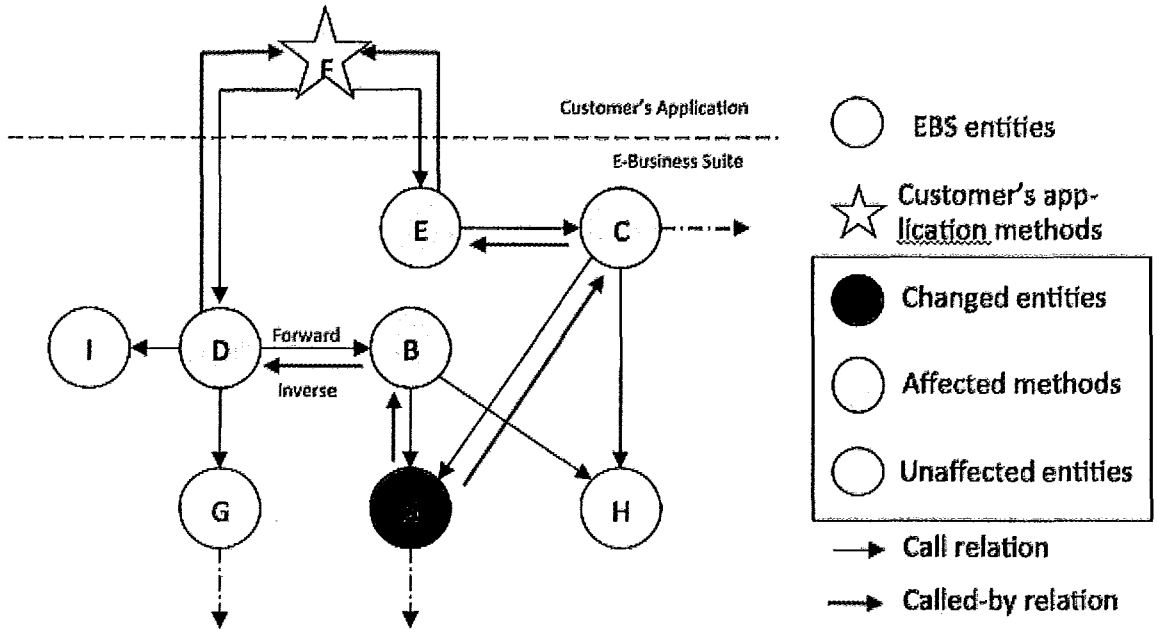


Figure 4.4: Impact analysis in ADG

In the figure 4.4 *A, B, C, D, E, F, G, H, I* are entities of the ADG and black arrows between them are dependencies. So if the arrow goes from *B* to *A* it means that entity *B* calls or accesses entity *A*. A red circle shows that it is a changed entity of the EBS which was modified after applying a patch. Yellow circles show those methods that are transitively dependent on a red circle, which is *A* in our case. Red arrows illustrate how the dependencies are revealed, leading to method *F* which belongs to the customer's application.

The InversePathGenerator outputs information about all dependent methods to the DFS_n database table where n is the number of the patch. The table consists of rows that include two identification numbers [2]: one for the entity that is accessed or called, the second for the entity that accesses or calls. In the next step this table is used for identifying methods that belong to the customer's application in order to perform test selection.

[2]Identification numbers here represent unique numbers that were assigned to each entity in ENTITY database table

4.5 Test Selection

Now after we have identified which entities changed and which were affected by those changes, it is time to determine the affected customer's application methods and perform regression test selection.

Our tool called TestSelect does two things:

- Identifies affected methods in the customer's application by parsing the DFS_n database table that was filled in in the previous step. Identification is performed through mapping caller methods to those methods that belong to the customer's application.
- Selects the test cases that examine those affected methods.

Here we need to point out one more time that we knew nothing about what the customer's application looked like, nor we know how its test suite was represented. In order to perform test selection for the customer's application, we needed test cases for it. It was decided to create them as well, as we did when we created the application itself. Rather than creating every single test case manually it was decided to find a tool that could generate test cases for us. The idea was that even if the customer already has test cases that examine, for instance, requirements, it would still be useful to have a tool that could generate unit tests and, as such, provide additional coverage which is an examination of the code. There are several tools for Java, most of which are proprietary, that can generate test cases for Java.

4.5.1 Test Case Generation

Among those tools that were open source, we found three tools, namely JUB (JUnit test case Builder), TestGen4J and JCrasher. An empirical comparison of these three tools is provided in [WO09] where authors introduce mutants into code and then let each tool generate test cases and at the end compare which tool has revealed the greatest number of mutants. Manually written test cases that provide edge coverage and a set of random test cases were also subject to comparison. According to the empirical results, edge coverage test cases revealed the highest number of mutations in the code. The worst results were shown by JUB. These results are illustrated in

the figure 4.5 by means of the total percent of mutants that were revealed by a certain set of test cases.

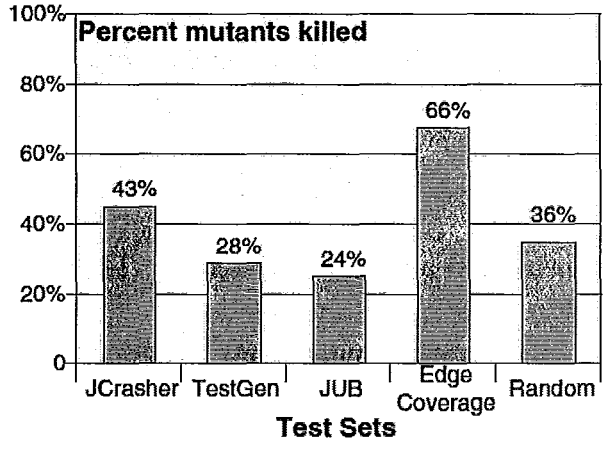


Figure 4.5: Total percent of mutants killed by each test set [WO09]

Because among other open source test generation tools the JCrasher showed better results according to [WO09] and also because of lack of documentation for the other two tools, it was decided to use JCrasher in this work.

JCrasher is an automatic test case generation tool for Java that examines the type information of classes. It generates random test cases that pass the type-correct inputs to the program's public methods so that the program would eventually "crash", i.e., output an unexpected exception. Its main advantages among other tools are [CS04]:

- it transitively analyzes methods, calculates the size of each method's parameter space and randomly selects parameters in accordance with time allocated for testing.
- it heuristically decides whether the program's crash is a bug or it is a violation of the program's preconditions.
- it returns a program to its initial state after each test case in an efficient way.
- test cases it produces support JUnit.

- it can be integrated in the Eclipse IDE.

JCrasher takes as an input the folder where all class files or source files are located and outputs a set of Java source files with test cases. Each such Java source file examines one public method of the application. An example of such a file is presented in Appendix A.

4.5.2 TestSelect

As was mentioned above, we created the TestSelect tool that finds affected customer's application methods and performs test case selection. Additionally, it indicates which methods of the customer's application are missing tests, i.e., in the existing regression test suite there is no test case that examines it. The latter point is especially interesting because it gives a hint to the customer about which methods were affected and might possibly introduce a fault after a patch has been applied. So there will be a target on which further test suite augmentation process should be focusing.

See pseudocode 1 for the TestSelect tool that shows the basic steps of how the test selection is performed.

4.6 Inclusiveness, Preciseness, Efficiency and Generality of the Proposed RTS Technique

Now it is time to perform an analysis of our technique in regards to its properties, which are inclusiveness, precision, efficiency and generality, described in Chapter 2.

As was mentioned in Chapter 2, the RTS technique's inclusiveness shows to which extent the technique selects modification-revealing test cases. There is a theorem and corresponding proof in [Rot96] that there is no algorithm that for given P, P' and T solves the modification-revealing test identification problem. Therefore we reduce the problem to identifying modification-traversing test cases such that any technique that selects all such test cases is 100% inclusive or safe.

Since when patches are applied, they only change the code in the EBS and the

Input: patch number n
Set of test cases in existing test suite T
The location of test cases *path*
Output: Set of selected test cases S
Set of methods that miss tests M

```
1  begin
2     get methods from table DFS_n that belong to customer's application
3     foreach m ∈ methods do
4        foreach file f in path do
5           if f tests m then
6              mark m that it has tests
7              test_cases = test_cases ∪ f
8           end
9        end
10    end
11    foreach m that is not marked to have tests do
12       M = M ∪ m
13    end
14    foreach t ∈ test_cases do
15       S = S ∪ t
16    end
17    return S, M
18 end
```

Algorithm 1: Pseudocode for TestSelect tool

database layer, the specification for the customer's application does not change, and therefore all the existing test cases still specify valid inputs and valid input-output relations. This means that test cases are not obsolete. Thereby there is no need to check if the test cases are obsolete. Also there are no obsolete fault-revealing test cases such as those that are shown in figure 2.2.

The modification can be either an addition of new code, modifying existing code or deletion of code from P in P'. Our technique can deal with all of these types of changes. Our ClassDiff tool compares two versions of all changed class files and identifies modifications at an entity-level granularity. So coming back to our figure 1.4, we identify all red circles first. After that our impact analysis, implemented in the InversePathGenerator tool, finds all methods that are affected by those changed entities, which corresponds to yellow objects in the figure 1.4. Now the TestSelect tool determines which affected methods belong to the customer's application, which corresponds to identifying yellow stars. Finally, the TestSelect tool selects all test cases that examine affected customer's application methods. Considering that existing regression test cases are written for examining only the customer's application methods and the analysis that we perform is static, as it considers all possible access dependencies among entities in both the customer's application and the EBS layers, we can conclude that our proposed technique is therefore safe as it selects all modification-traversing test cases.

In relation to our technique's precision, it is obvious that it is not 100% precise as it selects all modification-traversing test cases, which may not be modification-revealing. Just to remind the reader, a technique is 100% precise if it omits all non-modification-revealing test cases. Our technique admits some non-modification-traversing test cases as well because, even though we choose all test cases that examine a given affected method in the customer's application, those test cases may not reach the modified code and so would not reveal a fault. Our technique's inclusiveness and precision diagram is exactly like the one for the cluster identification technique that was shown in Chapter 2. It is also illustrated here in figure 4.6. Since it is safe, all circles are shaded, meaning that all modification-traversing test cases get selected. The shaded areas outside the circles illustrate admitting some non-modification-traversing

test cases that can happen when existing test cases do not actually examine the modified part of the code.

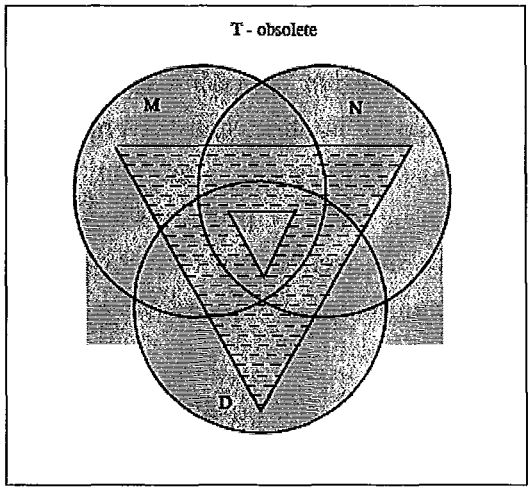


Figure 4.6: Inclusiveness and precision diagram for proposed RTS technique

Even though our technique is not 100% precise it is still efficient for a given SUT [3]. It is because the EBS layer consists of a huge amount of classes. The one that we used in our project consists of 230,000 classes and almost 4 million methods. Patches that are released by Oracle normally change only a tiny bit of the system. And so the number of affected customer's application methods can be really small. The time spent on analysis in our technique is also small compared to the time spent by the customer to perform the "retest-all" technique after each patch. Building the ADG took us approximately 7 hours on a Quad Core 3.2 GHz machine with 32 GB RAM running 64-bit Linux and the impact analysis in the worst case took about 30 minutes, while typically taking only 3 seconds. The time spent on test selection is proportional to the number of affected customer's application methods times the number of test cases. With the application that has one affected function and 21 test cases it took 2 seconds to perform test selection by launching the TestSelect tool on a local machine [4]. The technique is quite efficient, especially considering that the ADG can be built just for the first instance of regression testing, as the following

[3]System Under Test

[4]Also takes into consideration the time to connect to the server.

testing processes would use a modified ADG that is already built with consideration of changes from the last patch.

Our technique can be applied in cases when nothing except the code of the SUT changes, which means that it does not consider cases when there is a change in platform or migrating to another operating system and so on. So, basically, the assumption that should hold for modification-traversing test cases, which was mentioned in Chapter 2, should also hold in our case. Our technique was applied specifically to Java classes, but can be used for programs written on any OOP language. Even though the architecture we used was a three-layer system, it does not need to be as such, and that makes the approach applicable to many kinds of applications and increases its generality.

4.7 Summary

In this chapter we showed how we solved the problem by providing details about each step that was mentioned in Chapter 1. We explained how we build the access dependency graph (ADG) for all entities in the EBS and the customer's application using our own DependencyAnalyzer tool which builds the graph via conservative analysis. It then serves as a knowledge base of dependencies in further steps. The ADG does not need to be rebuilt for each regression testing process. There is a possibility to change some parts of it after finishing analysis in correspondence with changes that a patch introduced. Each patch can be analyzed for which files it changes using an existing Oracle tool and our own tool called LGIParser. Using our ClassDiff tool, we are able to identify which entities are changed by comparing two versions of every class file. We then perform our impact analysis using our InversePathGenerator tool that detects all methods that are affected by changed entities. Its output is an input for our TestSelect tool that identifies affected customer's application methods and selects corresponding test cases that examine them and also outputs information about which affected customer's application methods do not have tests. At the high level the whole solution is illustrated in figure 4.7.

In this chapter we also provided an analysis of our technique's properties like inclusiveness, preciseness, efficiency and generality to show that (1) it is cost-effective

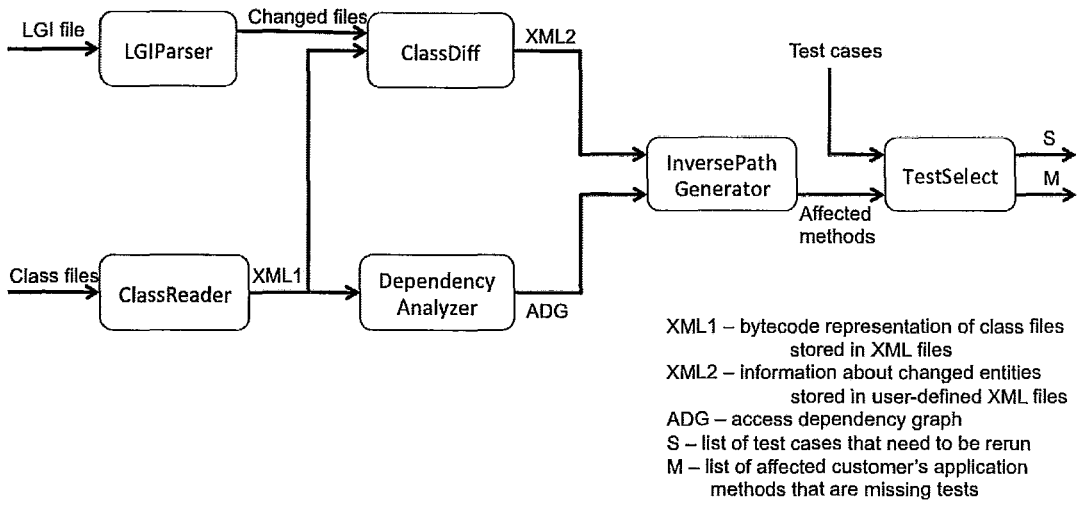


Figure 4.7: Solution diagram

for the system that we worked on in our project and that (2) it can be applied to other projects and can be a good alternative to existing RTS techniques.

The next chapter is about some aspects that need to be considered to make our proposed RTS technique even better and further steps in dealing with the problem.

Chapter 5

Future Work

In this chapter we are discuss some aspects of our research that were left out because of the overall burden and lack of time but still need to be done. One of them is adding the database layer to our analysis. Even though we considered only Java classes from class files, there are also several other file extensions that come with Oracle patches and which might introduce changes to either the EBS or the database and thus potentially introduce a fault to our system. Also performing only regression test selection can not reveal all the faults that were introduced with modification. It might happen that we do not have tests for some affected customer's application methods. Test suite augmentation deals with that kind of a problem and we are also going to discuss that in this chapter.

5.1 Adding the Database Layer

As was shown in figure 1.1 of Chapter 1 our system consists of three layers, namely the Oracle database, Oracle E-Business Suite (EBS) and the customer's application. However in this work we only considered interaction between two of them: the EBS and the customer's application.

Oracle patches can also introduce modifications to the Oracle database layer. Changes can be as follows:

- Changes in data

- Changes in table structure like an addition of a new column or a modification of the type of a column and so on
- Changes in triggers
- Changes in PL/SQL functions or procedures

If modifications are introduced to the database there may exist dependencies between different entities in the database, the EBS and the customer's application that can get affected from those modifications and therefore may affect the customer's application methods and introduce faults to our system. Some of the dependencies that may exist between our three layers (excluding dependencies between the EBS and the customer's application):

- Dependencies between two tables through foreign keys
- Dependencies between PL/SQL functions/procedures
- Dependencies between tables and triggers
- Dependencies between triggers and PL/SQL functions/procedures
- Dependencies between the customer's application methods and PL/SQL functions/procedures
- Dependencies between the customer's application methods and the database tables
- Dependencies between the EBS entities and PL/SQL functions/procedures
- Dependencies between the EBS entities and the database tables

All above mentioned kinds of dependencies need to be included in our access dependency graph (ADG) so that when changes are introduced in either the EBS or the database layer we can safely trace back to all affected objects up to the customer's application methods taking into an account all possible dependencies between all three layers of the system.

5.2 Analyzing Other Kinds of Files that Come with Oracle Patches

In this work we only considered class files and performed our analysis only on Java classes. Even though Oracle E-Business Suite is mainly written in Java, the patches come with variety of kinds of files that need to be analyzed for they might also introduce certain modifications to the system and therefore it is potentially dangerous not to consider them.

We surveyed which kinds of files may be patched, and checked over 40 different patches of different size and format [Abd10a]. We found 32 different kinds of files that we think could potentially change the EBS or the database functionality. We intentionally excluded from the list documentation files such as those with extension "doc", "htm", "rtf", because it is quite obvious that they can not change functionality of the system and introduce a different behaviour. Among those 32 file extensions we distinguished ones that can potentially change the EBS or the database or both. Through thorough analysis of real samples of the files and searching information about them on the Internet led us to the results that 6 file extensions potentially change the database layer ("ctl", "lct", "odf", "pls", "sql", "xdf"), 6 file extensions potentially change the EBS layer ("drv", "fmb", "jsp", "rdf", "wft", "wfx") and 4 file extensions potentially change both ("pkb", "pl", "pll", "xml"). The details can be obtained from [Abd10a].

Some work has been done towards analyzing some file extensions, namely we took "ctl" file extension under consideration which was shown to potentially change the database layer. CTL files are control files for Oracle's SQL Loader tool that consist of information about a structure of a certain table in the database and might also include data that needs to be loaded into the database. Our tool called CTLParser gets the list of all CTL files from the database table that changed after the patch (database table is filled in by our LGIParser tool that was used in the second step of our solution) and parses them to obtain names of the tables. From there, it is possible to add those table names to ENTITY table and also search for dependent entities to put information about dependencies in DEPENDENCY_RELATION table to extend our ADG.

5.3 Test Suite Augmentation

After certain changes to the system have been introduced, performing regression test selection and running selected existing test cases do not guarantee that there would be no fault in the system. It might happen that the changes introduced a part of the code that was never examined before and therefore does not have any tests. Such modifications can give a birth to unexpected faults and cause a lot of problems, because it would be very hard to identify where exactly those faults came from afterwards. It is much more effective to be able to create new test cases that would complement the existing ones. So after each modification running the selected test cases and new ones would provide an assurance that every piece of code that could potentially introduce a fault was examined.

In our particular case, where patches change the EBS or the database layer entities, it might happen that from tracing back up to the customer's application methods we might reveal that these methods were never examined before and therefore do not have any tests. Another possible case is when test cases that examine affected customer's application methods do not reach changed parts of the EBS and the database. This is an interesting scenario, because it would mean that the existing test cases are not adequate any more for that particular case or that particular patch. And, because we are essentially performing static analysis on what kind of dependencies we have in our system, it is possible to eventually get to those changed parts from the customer's application methods and end up with faults. Whether that kind of scenario would be realistic is another question. It might be the case that the customer does not even use that particular functionality of the EBS or the database where the changes had been introduced and so there is no need to examine it. But knowing what has changed and where the effects are is definitely useful as it can leave the customer a choice what to do next. That is why in our work we included the part where, after applying our tools, the customer would know which customer's application methods are affected and also which miss tests.

In order to know whether a certain test case reaches some part of the code in a system the code needs to be instrumented. Instrumentation is a process that allows to record the steps overtaken by a certain program (in our case it is a test case) during its execution. There are lots of ways and tools to instrument the code. One of

such tools is called InsECT [CO04] that works with Java programs and instruments the code on Java bytecode level using Byte-Code Engineering Library (BCEL). It can instrument the code either statically or dynamically. Instrumentation is very important for test suite augmentation, because it gives us details about execution of each test case. And so, if we create a new test case, we will know if it reaches a certain point in the code after we execute it.

Harrold et al. [TSC+06; SCT+08] propose their own approach and tools for test suite augmentation problem which works with Java. Their first tool called Matrix for given two versions of the program and requested distance of dependence from changes outputs set of testing requirements that can be used for creating new test cases. This tool uses data and control dependence information, generates testing requirements only when the execution follows different paths in P and P' and works only on a single change. The upgraded version of the tool called MatrixReloaded is able to handle multiple changes at once and performs state analysis of statements that are data or control dependent on changed parts based on symbolic execution. The tool also checks if existing test cases satisfy generated testing requirements by instrumenting the code and running all test cases of the test suite and leaves only those requirements that were not satisfied by any test case. Both tools use InsECT to instrument the code and obtain coverage of the requirements produced by instrumented program. Applying similar approaches to our work would be beneficial as it would not only check if existing test cases are adequate for a given modification, but also give us an idea of how to write new test cases if there is a need.

5.4 Summary

In this chapter we discussed different aspects that were either left out during implementation of our solution because of lack of time and lack of information about the system, or would be beneficial for our project to make the solution of the problem better. The database layer is a part of the whole system and Oracle patches possibly introduce modifications to it. So adding it to our analysis is thus very important, as there are dependencies between all three layers. Patches come with several different file extensions aimed to different functionality of the EBS and the database. Omitting

some of them can be dangerous as it might potentially change the system's behaviour. Our analysis of different file extensions showed those that can introduce a change to the EBS or the database or both. Performing effective regression test selection certainly saves the customer a time when performing regression testing. But existing test cases may not be adequate for a given modification and thus many faults can go undetected. The test suite augmentation concept deals with that kind of a problem. We mentioned few recent papers on how it is performed and what is the rationale behind it.

Chapter 6

Conclusion

In this work we presented a regression test selection technique that is applied to three-layer system consisting of the Oracle database, Oracle E-Business Suite (EBS) and the customer's application. The customer's application is a wrapper program that is dependent on the database and the EBS. It represents a legacy system because it was written long time ago and lacks an appropriate technical and functional documentation. Patches that Oracle introduces to its database and ERP system can change the customer's application behaviour and introduce faults. But running the huge test suite during regression testing process is currently costing the customer a lot - both in terms of time and money. The customer's application is a real-time system that is being operated on a daily basis and so holding it for a long time to perform testing is a big pain. Obviously there is a need in optimizing the regression testing and this work presents a partial solution to the problem.

The three-layer system that we were dealing with represents a huge amount of code. To reach a feasible stage in our solution we had to limit ourselves to only two layers, namely the EBS and the customer's application. Also we assumed that the customer's application is written in Java and so we analyzed only Java classes in the EBS and the customer's application.

The solution that we propose consists of five consecutive steps:

- Converting all class files of the system to their bytecode representation in XML files and from parsing those files building an access dependency graph (ADG) that consists of information about what kind of entities (methods and fields of

the classes) we have and what are dependencies between them.

- Analyzing Oracle patches without applying them to identify which existing files change.
- By knowing which files change, we parse bytecode representation in XML files of both versions of each file to identify which entities have changed.
- Changed entities are the input to our impact analysis part that identifies which methods are affected by those changes in the system using depth-first search algorithm on information from the ADG.
- From affected methods we identify those that belong to the customer's application and run the test selection program for them that will identify which test cases need to be executed and which affected methods of the customer's application did not have tests.

In our analysis we built an access dependency graph (ADG) which is basically a method-level control flow graph but with addition of fields of the classes. Building the ADG represents a static analysis for Java as it counts for every possible scenario that can happen during execution. Because of that the test selection algorithm that we propose is *safe*, i.e., it selects all test cases from the existing test suite that can reveal a fault in the system after modification. We also showed that our technique is very efficient. It only takes a while to generate the ADG, but that can be done just once, before the first regression testing process. After each regression testing ADG can be updated with regards to changes that a patch introduced and so updated ADG can be used for further regression testing processes. Oracle patches usually do not introduce a lot of changes to the EBS and the database and so the amount of affected customer's application methods would be relatively small. Thus selecting those test cases that examine only the affected customer's application methods would dramatically reduce the time spent on regression testing, especially comparing to "retest-all" approach that is currently being used by the customer. The time spent on analysis and selection is also minor and thus lets us deduce that our technique is efficient. Empirical results were provided in Chapter 4.

In Chapter 2 we showed several existing regression test selection techniques.

Our technique is unique in that it specifically deals with three-layer system and can be very efficient. There is no restriction on using our technique only on that kind of architecture.

In Chapter 5 we discussed some aspects that need to be taken into consideration to make our technique better and more suitable for the whole problem. Namely, we talked about a need to add the third layer of our three-layer system which is the database layer. We also showed that Oracle patches can include not only class files but several different file types that potentially change either the EBS or the database functionality or both. Performing regression test selection and thus reducing the cost of regression testing is very useful, but it might not be enough because existing test cases may not reveal a fault introduced by a change from the patch, or it might be the case that the existing test suite does not have a test for a given affected customer's application method. These kinds of issues are handled by a test suite augmentation problem that was also discussed in Chapter 5.

Let us conclude this work with quotation from Barbara Hall: "The path to our destination is not always a straight one. We go down the wrong road, we get lost, we turn back. Maybe it does not matter which road we embark on. Maybe what matters is that we embark."

Appendix A

Sample of JUnit File Generated by JCrasher

```
1  package codeundertest;
2
3  /**
4   * Test cases for compareTo
5   */
6  public class MyCodeTest3 extends edu.gatech.cc.junit.
       FilteringTestCase {
7
8    /**
9     * Executed before each testXXX().
10    */
11   protected void setUp() {
12     /* Re-initialize static fields of loaded classes. */
13     edu.gatech.cc.junit.reinit.ClassRegistry.resetClasses();
14     //TODO: my setup code goes here.
15   }
16
17   /**
18    * Executed after each testXXX().
19    */
```

```
20    protected void tearDown() throws Exception {
21      super.tearDown();
22      //TODO: my tear down code goes here.
23    }
24
25    /**
26     * JCrasher-generated test case.
27     */
28    public void test0() throws Throwable {
29      try{
30        wps.gantt.common.data.JobPrimaryKey j1 = (wps.gantt.common.
              data.JobPrimaryKey)null;
31        MyCode m2 = new MyCode();
32        m2.compareTo(j1);
33      }
34      catch (Throwable throwable) {throwIf(throwable);}
35    }
36
37    /**
38     * JCrasher-generated test case.
39     */
40    public void test1() throws Throwable {
41      try{
42        wps.gantt.common.data.JobPrimaryKey j1 = new wps.gantt.
              common.data.JobPrimaryKey(-1);
43        MyCode m2 = new MyCode();
44        m2.compareTo(j1);
45      }
46      catch (Throwable throwable) {throwIf(throwable);}
47    }
48
49    /**
50     * JCrasher-generated test case.
51     */
```

```
52  public void test2() throws Throwable {
53    try{
54      wps.gantt.common.data.JobPrimaryKey j1 = new wps.gantt.
          common.data.JobPrimaryKey(0);
55      MyCode m2 = new MyCode();
56      m2.compareTo(j1);
57    }
58    catch (Throwable throwable) {throwIf(throwable);}
59  }
60
61  /**
62   * JCrasher-generated test case.
63   */
64  public void test3() throws Throwable {
65    try{
66      wps.gantt.common.data.JobPrimaryKey j1 = new wps.gantt.
          common.data.JobPrimaryKey((wps.gantt.common.data.
          JobPrimaryKey)null);
67      MyCode m2 = new MyCode();
68      m2.compareTo(j1);
69    }
70    catch (Throwable throwable) {throwIf(throwable);}
71  }
72
73
74  protected String getNameOfTestedMeth() {
75    return "codeundertest.MyCode.compareTo";
76  }
77
78  /**
79   * Constructor
80   */
81  public MyCodeTest3(String pName) {
82    super(pName);
```

```
83    }
84
85    /**
86     * Easy access for aggregating test suite.
87     */
88    public static junit.framework.Test suite() {
89      return new junit.framework.TestSuite(MyCodeTest3.class);
90    }
91
92    /**
93     * Main
94     */
95    public static void main(String[] args) {
96      junit.textui.TestRunner.run(MyCodeTest3.class);
97    }
98  }
```

Bibliography

[AACM09] Ian Abramson, Michael Abbey, Michael J. Corey, and Michelle Malcher. *Oracle Database 11g: A Beginner's Guide.* The McGraw-Hill Companies, Inc., 2009.

[Abd10a] Akbar Abdrakhmanov. Analyzing file extensions from the patches. Technical report, McMaster University, 2010.

[Abd10b] Akbar Abdrakhmanov. Parsing adpatch LGI log files. Technical report, McMaster University, 2010.

[AIC10] Akbar Abdrakhmanov, Asif Iqbal, and Wen Chen. Reducing the risk of patching. Technical report, McMaster University, 2010.

[All09] Tyler Allman. How should government IT professionals manage legacy code. *Enterprise Innovation (http://www.enterpriseinnovation.net)*, 2009.

[AO08] Paul Ammann and Jeff Offutt. *Introduction on Software Testing.* Cambridge University Press, 2008.

[Bei90] Boris Beizer. *Software Testing Techniques.* International Thomson Computer Press, 2 edition, 1990.

[Che11] Wen Chen. Private communication. 2011.

[CO04] Anil Chawla and Alessandro Orso. A generic instrumentation framework for collecting dynamic information. *ACM SIGSOFT Software Engineering Notes*, 29(5):1–4, September 2004.

[CS04] Christoph Csallner and Yannis Smaragdakis. Jcrasher: an automatic robustness tester for java. *Software - Practise & Experience*, 34(11):1025–1050, 2004.

[Far09] Robert Farrington. Oracle E-Business Suite Patching Procedures, Release 12.1. Technical report, Oracle Corporation, 2009.

[Fea04] Michael C. Feathers. *Working Effectively with Legacy Code.* Prentice Hall PTR, September 2004.

[Iqb11] Asif Iqbal. Identifying modifications and generating dependency graphs for impact analysis in a legacy environment. Master's thesis, McMaster University, 2011.

[LH08] Wanchun Li and Mary Jean Harrold. Using random test selection to gain confidence in modified software. In *Software Maintenance, 2008. ICSM 2008.*, pages 267–276, 2008.

[Mye04] Glenford J. Myers. *The Art of Software Testing.* John Wiley & Sons, Inc., 2 edition, 2004.

[OM07] Andres S. Orrego and Gregory E. Mundy. A study of software reuse in NASA legacy systems. *Innovations in Systems and Software Engineering*, 3(3):167–180, July 2007.

[OSH04] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. Scaling regression testing to large software systems. In *ACM SIGSOFT Symposium on the Foundations of Software Engneering*, pages 241–252, Newport Beach, CA, November 2004.

[PA10] Anil Passi and Vladimir Ajvaz. *Oracle E-Business Suite Development and Extensibility Handbook.* The McGraw-Hill Companies, Inc., 2010.

[RH96] Gregg Rothermel and Mary Jean Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, August 1996.

[Rot96] Gregg Rothermel. *Efficient, Effective Regression Testing Using Safe Test Selection Techniques.* PhD thesis, Clemson University, May 1996.

[SCT+08] Raul Santelices, Pavan Kumar Chittimalli, Apiwattanapong Taweesup, Alessandro Orso, and Mary Jean Harrold. Test-suite augmentation for evolving software. *Automated Software Engineering, International Conference on Automated Software Engineering*, pages 218–227, 2008.

[SPL03] Robert C. Seacord, Daniel Plakosh, and Grace A. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practises.* Addison Wesley, 2003.

[Tes10] Jean Tessier. *The Dependency Finder User Manual.* http://depfind.sourceforge.net/Manual.html, 2010.

[TSC+06] Apiwattanapong Taweesup, Raul Santelices, Pavan Kumar Chittimalli, Alessandro Orso, and Mary Jean Harrold. Matrix: Maintenance-oriented testing requirements identifier and examiner. In *Testing: Academic and Industrial Conference - Practise and Research Techniques*, pages 137–146, 2006.

[Tuc00] Sean Tuck. *Adpatch Basics*, 2000.

[Ulr02] William M. Ulrich. *Legacy Systems: Transformation Strategies.* Pearson Education, Inc., 2002.

[WO09] Shuang Wang and Jeff Offutt. Comparison of unit-level automated test generation tools. *Software Testing Verification and Validation Workshop*, pages 210–219, 2009.

[YH10] Shin Yoo and Mark Harman. Regression testing minimisation, selection and prioritisation: A survey. *Software Testing, Verification and Reliability*, 2010.

[ZRW09] Chris Zeis, Chris Ruel, and Michael Wessler. *Oracle 11g for Dummies.* Wiley Publishing, Inc., 2009.