Bo Xie
AUTEUR DE LA THÈSE / AUTHOR OF THESIS

Master of Computer Science
GRADE / DEGREE

School of Information Technology and Engineering
FACULTÉ, ÉCOLE, DÉPARTEMENT / FACULTY, SCHOOL, DEPARTMENT

Requirement Based Regression Test Suite Reduction Using Dependence Analysis
TITRE DE LA THÈSE / TITLE OF THESIS

H. Ural
DIRECTEUR (DIRECTRICE) DE LA THÈSE / THESIS SUPERVISOR

CO-DIRECTEUR (CO-DIRECTRICE) DE LA THÈSE / THESIS CO-SUPERVISOR

EXAMINATEURS (EXAMINATRICES) DE LA THÈSE / THESIS EXAMINERS

L. Labiche R. L. Probert

Gary W. Slater
LE DOYEN DE LA FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTDOCTORALES / DEAN OF THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES
Requirement Based Regression Test Suite Reduction Using Dependence Analysis

Bo Xie

A thesis submitted to
The Faculty of Graduate and Postdoctoral Studies of the University of Ottawa
in partial fulfillment of the requirements for the degree of
Masters in Computer Science *

School of Information Technology and Engineering
University of Ottawa
Ottawa, Ontario, Canada

* The Masters program in Computer Science is a joint program with Carleton University, administered by the Ottawa Carleton Institute for Computer Science
NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.
Abstract

During maintenance of evolving software systems, regression testing is a crucial activity in confirming that the unchanged parts of the system have not been adversely affected by the modifications on the system specification and implementation. It is time-and computing-resource-consuming, especially for large software systems. Therefore, regression test suite reduction is important. Most regression testing techniques are code-based. Requirement-based (specification-based) regression testing is a system testing technique used to test software systems modeled by formal description languages, e.g., an Extended Finite State Machine (EFSM). There exists limited research on requirement-based regression testing techniques, and most of these techniques select regression tests using only the modified models of systems. Since the original model is not used to select regression test suites, the quality of regression test suites is questionable.

In this research, we present an extension to an existing requirement-based regression test suite reduction approach that uses EFSM model dependence analysis to reduce a given regression test suite [19]. The approach is based on the difference between the original model and the modified model expressed as a set of elementary model modifications: elementary addition of a transition and elementary deletion of a transition. For each elementary modification, the data and control dependencies are used to capture potential interactions between EFSM transitions. The potential interactions are used to reduce an existing regression test suite by eliminating repetitive tests.

In this thesis, based on [19], we have defined some new dependencies introduced by elementary modifications of the EFSM model. We proposed algorithms to obtain these
dependencies; to generate potential interactions with respect to (wrt) an elementary modification; and to reduce the regression test suite. We have also developed a Regression Test Suite Reduction tool, called RTSR based on these algorithms to be used for reducing the size of existing regression test suites. RTSR has been tested and validated.
Acknowledgments

First and foremost, I would like to express my deepest gratitude to my supervisor Dr. Hasan Ural for his invaluable guidance, continuous supports, and numerous encouragements, especially for giving me feedback at amazing speed.

I would like to thank everyone from the Advanced Software Engineering Research and Training Group (ASERT), especially to Dr. Robert L. Probert, Tuong Nguyen, Panitee Ritthiruangdech, and Olfa Chemli for their useful suggestions and help.

I would also like to thank my husband and my parents for their love, support, and encouragement during the whole period that went into pursuing my graduate studies.

Finally, I would like to acknowledge financial support from the Ontario Graduate Scholarship Program (OGS), Communications and Information Technology Ontario (CITO, now Ontario Centres of Excellence, OCE), and the University of Ottawa.
Table of Contents

Abstract ......................................................................................... i
Acknowledgments ........................................................................ iii
Table of Contents .......................................................................... iv
List of Figures ............................................................................... vi
List of Tables ................................................................................ vii
List of Tables ................................................................................ vii
Chapter 1 Introduction .................................................................. 1
  1.1 Background ............................................................................ 1
  1.2 Contributions of the Thesis ..................................................... 2
  1.3 Organization of the Thesis ...................................................... 3
Chapter 2 Formal Description Languages, Test Construction Methods, and Regression Testing Techniques .................................................. 4
  2.1 EFSM ....................................................................................... 4
  2.2 Test Construction Methods ..................................................... 7
     2.2.1 Control Flow Oriented Test Selection Criteria .................... 8
     2.2.2 Data Flow Oriented Test Selection Criteria ....................... 11
     2.2.2.1 Data Flow Related Concepts ........................................ 11
     2.2.2.2 Data Flow Oriented Test Selection Criteria .................... 16
  2.3 Regression Testing Techniques .............................................. 18
     2.3.1 Requirement-Based Regression Testing Strategies ............. 23
     2.3.2 Requirement-Based Regression Test Suite Reduction Techniques .......... 27
Chapter 3 Dependence Analysis in the EFSM Model ....................... 29
  3.1 Data Dependence ................................................................... 29
  3.2 Control Dependence ............................................................. 30
  3.3 Static Dependence Graph ....................................................... 32
  3.4 Algorithm for Generating the SDG of the Modified EFSM ............ 34
     3.4.1 Algorithm for Affecting Data Dependence ....................... 38
     3.4.2 Algorithm for Affecting Control Dependence ................... 40
     3.4.3 Algorithm for Affected Data Dependence ....................... 41
     3.4.4 Algorithm for Affected Control Dependence ................... 42
     3.4.5 Algorithm for Activation Dependence ............................. 43
     3.4.6 Algorithm for Affecting Ghost Data Dependence .............. 45
     3.4.7 Algorithm for Affected Ghost Data Dependence .............. 47
     3.4.8 Algorithm for Ghost Activation Dependence ................... 49
  3.5 SDG of the Simplified ATM System with Modified Transactions .... 51
Chapter 4 Dependence Analysis for Regression Test Suite Reduction .... 53
  4.1 Testing the Addition of a Transition ....................................... 54
  4.2 Testing the Deletion of a Transition ....................................... 56
  4.3 Algorithm for Reducing Regression Test Suite ....................... 60
  4.4 Algorithm for Generating Interaction Patterns for a Given Test Sequence .... 62
List of Figures

Figure 2.1 An EFSM Model for a Simplified ATM System [19]................................. 7
Figure 2.2 Flow Graph Generated from the EFSM of the Simplified ATM System........ 12
Figure 2.3 Relationships Among Test Selection Criteria ..................................... 18
Figure 2.4 EFSM Model of a Simplified ATM System with Added Transition T9............. 25
Figure 2.5 EFSM Model of a Simplified ATM System with Deleted Transition T6........... 26
Figure 3.1 Data Dependence ................................................................................. 30
Figure 3.2 Data Dependence in the Example EFSM in Figure 2.1.............................. 30
Figure 3.3 Post-Dominance ................................................................................. 31
Figure 3.4 Control Dependence ............................................................................ 31
Figure 3.5 Control Dependence in the Example EFSM in Figure 2.1......................... 31
Figure 3.6 Static Dependence Graph (SDG) for the Example EFSM in Figure 2.1....... 32
Figure 3.7 Affecting Data Dependence .................................................................. 39
Figure 3.8 Affecting Control Dependence ............................................................. 40
Figure 3.9 Affected Data Dependence ................................................................... 41
Figure 3.10 Affected Control Dependence ............................................................. 42
Figure 3.11 Activation Dependence ....................................................................... 44
Figure 3.12 Affecting Ghost Data Dependence ...................................................... 46
Figure 3.13 Affected Ghost Data Dependence ....................................................... 48
Figure 3.14 Ghost Activation Dependence ............................................................. 49
Figure 3.15 EFSM Model of the Simplified ATM System with Added Transition T9 and Deleted Transition T6................................................................. 51
Figure 3.16 SDG of the Simplified ATM System with Modified Transactions ............. 52
Figure 4.1(a) Sub-SDG of Test_1 with Added Transition T9 Marked in Bold................. 55
Figure 4.1(b) Sub-SDG of Test_2 with Added Transition T9 Marked in Bold............... 55
Figure 4.2 The Affecting Interaction Pattern for Test_1 and Test_2............................ 56
Figure 4.3 The Affected Interaction Pattern for Test_1 and Test_2............................. 56
Figure 4.4 The Side-Effect Interaction Pattern for Test_1 and Test_2......................... 56
Figure 4.5(a) Sub-SDG of Test_1 with Deleted Transition T6 Marked in Dashed Cycle 58
Figure 4.5(b) Sub-SDG of Test_2 with Deleted Transition T6 Marked in Dashed Cycle 58
Figure 4.6 the Affecting Interaction Pattern for Test_1 and Test_2............................ 58
Figure 4.7 the Affected Interaction Pattern for Test_1 and Test_2............................. 58
Figure 5.1 Structure of RTSR .............................................................................. 76
List of Tables

Table 2.1 def sets and c-use sets in Figure 2.2 ........................................ 13
Table 2.2 p-use sets in Figure 2.2 ............................................................. 14
Table 2.3 Def-use Associations in Figure 2.1 ............................................. 15
Table 2.4 Some Data Flow Oriented Test Selection Criteria .................... 16
Table 5.1 BNF Definition of an EFSM Input File ...................................... 77
Table 5.2 BNF Definition of a MOD Input File .......................................... 80
Table 5.3 BNF Definition of a TS Input File .............................................. 82
Table 5.4 BNF Definition of an IP Output File ......................................... 83
Table 5.5 An Example of Dependencies from T1 to T2 ............................. 85
Table 5.6 An Example Internal Data Structure of Modified EFSM with Added Transition T9 ................................................................. 86
Table 5.7: An Example of Internal Data Structure of $S_M$ ........................ 87
Table 5.8: An Example of Reduced Regression Test Suite in RRTS File (.rr.ts file) ..... 89
Table 5.9: An Example of Affecting Interaction Pattern for T9 in IP Output File (.rip file) ................................................................. 89
Table 5.10 The Interaction Patterns and the Equivalent Test Cases wrt a Certain Interaction Pattern for T9 ....................................................... 90
Table 5.11 The Interaction Patterns and the Equivalent Test Cases wrt a Certain Interaction Pattern for T6dummy ........................................ 92
Table 5.12 Regression Test Suite Reduction of the Simplified ATM System .......................... 93
Table 5.13 Possible Number of Interaction Patterns wrt T9 ........................ 94
Table 5.14 Possible Number of Interaction Patterns wrt T6dummy ............... 94
Chapter 1

Introduction

1.1 Background

Software maintenance activities account for as much as two-thirds of the cost of software production. One necessary but expensive maintenance task is regression testing [30]. Regression testing is the process of validating that the changes introduced in a system are correct and do not adversely affect the unchanged portion of the system. During maintenance of evolving software systems, regression testing is a crucial activity in confirming our confidence of software in the presence of continuous changes of specification and implementation.

There has been a significant amount of research on the design of effective regression testing techniques to reduce the cost of regression testing [29]. There are two types of regression testing: code-based and requirement-based (specification-based) regression testing which complement each other [8]. Most regression testing techniques are code-based. There exists limited research on requirement-based regression testing techniques [4, 10, 24, 35]. This thesis deals with requirement-based regression testing.

Requirement-based testing is a model-based testing technique [5, 7, 11] that can be used on the system level. Model-based test generation techniques have been developed recently which can be applied to requirement-based testing if requirements can be modeled in some formalism. These techniques are appropriate for state-based systems that can be modeled using formal description languages like Extended Finite State Machine (EFSM), Specification Description Language (SDL), or ESTELLE, as shown in
[36, 16, 37], and are used to automatically generate system-level test suites even for large software systems. System models are frequently modified to reflect changes in specifications. When the system model is changed, we can apply model-based test generation techniques on the modified model, and partially test the system with respect to (wrt) a set of selected requirements. The size of these regression test suites may be very large even for relatively small systems. In addition, since the original model is often not used to select regression test suites, the quality of regression test suites is questionable [19].

We present an extension to an existing approach [19] of model-based regression test reduction that may significantly reduce the size of regression test suites. This approach is based on the difference between the original and modified model expressed as a set of elementary modifications: elementary addition of a transition and elementary deletion of a transition. For each elementary modification, regression test reduction strategies that use EFSM dependence analysis are used to reduce the regression test suites by eliminating repetitive tests.

1.2 Contributions of the Thesis

In this research, the reduction of model-based regression test suites using EFSM dependence analysis is studied. Given an EFSM model and a set of elementary modifications, EFSM dependence analysis is used to reduce a given regression test suite. In this research, the approach of [19] is extended:

- We have defined some new dependencies introduced by elementary modifications of the EFSM model, namely: affecting data dependence, affected data dependence, affecting control dependence, and affected control dependence.
• We proposed algorithms for obtaining all dependencies introduced by elementary modifications; for generating three types of interaction patterns wrt an elementary modification; and for regression test suite reduction.

• We also developed a regression test suite reduction tool (RTSR) based on the approach using C++. This tool automatically reduces a given regression test suite using system models in EFSM. Consequentially, the regression testing cost may be reduced significantly. An example has been provided to show the expected reduction in the size of test suites.

1.3 Organization of the Thesis

The remainder of this thesis is outlined as follows: Chapter 2 introduces regression testing techniques and requirement-based regression test suite reduction techniques. Chapter 3 introduces the concepts related to the dependence analysis on the EFSM model, and presents algorithms for discovering all dependencies introduced by elementary modifications of the EFSM model. Chapter 4 presents an approach that uses the dependence analysis for regression test suite reduction, and the algorithms based on this approach. Chapter 5 presents the Regression Test Suite Reduction tool, which has been developed based on the algorithms introduced in Chapter 3 and Chapter 4. Chapter 5 also presents the application of RTSR to an example. Chapter 6 concludes the thesis, and presents recommendations for further research.
Chapter 2

Formal Description Languages, Test Construction Methods, and Regression Testing Techniques

Generally, software system specifications consist of individual requirements, which are expressed informally in textual format – e.g., English – that may be ambiguous, inconsistent and incomplete. Hence, models and formal description techniques such as EFSM, SDL, and Estelle are used to describe requirements in order to eliminate problems associated with informal specifications. In this thesis, the EFSM models are used for the representation of system requirements.

2.1 EFSM

Extended Finite State Machines (EFSM) [39] have been widely used to model many types of systems, especially state-based systems found in telecommunications and computer communication networks. EFSM is an extension of the classical Finite State Machine (FSM), which adds variables, enabling predicate, and actions into a transition. An EFSM can be formally represented [28] as a seven-tuple \((S, s_{sr}, s_{ex}, I, O, V, T)\), where:

\[
\begin{align*}
S & \quad \text{finite set of states} \\
s_{sr} & \quad \text{start state} \\
s_{ex} & \quad \text{exit state} \\
I & \quad \text{finite set of input interactions} \\
O & \quad \text{finite set of output interactions} \\
V & \quad \text{finite set of variables} \\
T & \quad \text{finite set of transitions}
\end{align*}
\]
Each element of $T$ is a 5-tuple $t = (s_i, s_o, i, g, a)$ where $s_i$ and $s_o$ are states in $S$ representing the state from which $t$ is outgoing and the state to which $t$ is incoming, respectively. $i$ is an input interaction in $I$ that triggers $t$, $g$ is a Boolean condition that must be evaluated to be true for $t$ to be executed, and $a$ is a sequence of actions that takes place when $t$ is executed. An action may be an assignment or an output. Note that, in general, an EF$S$M model may include, in actions part of a transition, conditional statements such as if-then-else or case statements as well as repetitive statements such as for or while statements. In our work, we do not consider EF$S$Ms where transitions have conditional and repetitive statements. Interested reader may refer to [36] for handling of such statements within the context of the EF$S$M model.

An EF$S$M can also be graphically represented by a digraph $G = (V, E)$ where $V$ is a set of nodes, each representing a state in $S$ and $E$ is a set of edges, each representing a transition in $T$. For instance, Figure 2.1 shows an EF$S$M describing the requirements for a simplified Automated Teller Machine (ATM) System [19]. “Requirements” are traditionally non-executable. They are usually expressed in a natural language, declarative (non-imperative) form. In this thesis, we assume that requirements can be represented as a single EF$S$M and each requirement can be adequately represented by a single transition in this EF$S$M.

For example, the set of requirements of the simplified ATM system is $R = \{r1, r2, r3, r4\}$: where each $r$ is represented by a single transition in the corresponding EF$S$M:

- **r1**: User fails to enter a correct pin that matches with the PIN stored in the ATM card in less than four attempts. If pins don’t match, and less than four attempts were performed, the system displays an error message, increments the number of attempts
and prompts for pin. At the fourth attempt, if user still fails to enter a correct pin, the
system prints an error message and ejects the user’s ATM card.

\[ r1 = T2 \ T2 \ T2 \ T3 \]

\( r2 \): After entering a correct pin in less than four attempts, user selects withdrawal
function. The system adjusts the balance, and displays a menu with withdrawal, deposit, and exit functions.

\[ r2 = x \ T4 \ T5 \]

\( r3 \): After entering a correct pin in less than four attempts, user selects deposit function.
The system adjusts the balance, and displays a menu with withdrawal, deposit, and exit functions.

\[ r3 = x \ T4 \ T6 \]

\( r4 \): After entering a correct pin in less than four attempts, user selects exit function and
the system ejects the user’s ATM card.

\[ r4 = x \ T4 \ T8 \]

\( x \):  \textit{T2 may be inserted 0, 1, 2 or 3 times}  

The transition shown in bold in each sequence of transitions given above is the
\textit{transition under test (tut)} which represents the requirement \( r \).
2.2 Test Construction Methods

Control-flow and data-flow oriented testing are frequently used in white box testing to construct test cases. They have been adapted to black-box (requirement-based) testing for systems specified in SDL [36, 16], Estelle [37], and Lotos [34]. Since our work aims to reduce regression test suites produced from requirements of systems for both control flow oriented testing and data flow oriented testing, the control and data flow oriented test selection criteria and related concepts are discussed in the following sections. These criteria are based on single entry, single exit directed graphs (digraphs) representing the source code or the specification of the system under test. Here, we review these methods within the context of a single entry single exit digraph $G$ representing an EFSM where the entry node of $G$ is the start state of the EFSM and the exit node of $G$ is the exit state of the EFSM.

A path $(n_1, n_2, \ldots, n_m)$ in a directed graph $G$ is a sequence of nodes in $G$, such that for all $j$, $1 \leq j \leq m-1$, $m \geq 2$, $(n_j, n_{j+1}) \in E$. A sub-path of a path $(n_1, \ldots, n_m)$ is a path $(i_1, \ldots, i_k)$ if there exists a $\Delta$, $0 \leq \Delta \leq m-k$, such that for all $j$, $1 \leq j \leq k$, $i_j = n_{j+\Delta}$. A loop-free path is a
path in which all nodes are distinct. A complete path in a single entry, single exit directed graph $G$ is a path whose first node is the entry node and whose last node is the exit node. A complete path is executable or feasible if a set of input data which causes its execution exists, and un-executable (infeasible) otherwise. A path is executable if it is a sub-path of an executable complete path. Whether a path is executable depends on the semantics of the system itself, not just on the underlying graph structure. It is well-known that determining feasibility of a path is an undecidable problem.

Let $p = (n_1, \ldots, n_m)$ be a complete path in a directed graph $G$. We say that a node $n$ is covered by $p$ if $n = n_i$ for some $i$, $1 \leq i \leq m$. Similarly, an edge $(s, t)$ is covered by $p$ if $s = n_i$ and $t = n_{i+1}$ for some $i$, $1 \leq i \leq m-1$. A path $r$ is covered by $p$ if path $r$ is a sub-path of $p$. A node, edge, or path is covered by a set $P$ of complete paths of $G$ if the node, edge, or path, respectively is covered by a complete path in $P$.

Consider as an example, the EFSM of the simplified ATM system shown in Figure 2.1. In this graph, let $P$ be a set of complete paths starting at the start state and terminating at the exit state. For example, a complete path (T1, T4, T5, T7, T8) is in $P$ and covers nodes: start, S1, S2, S3 and exit states, covers edges: T1, T4, T5, T7 and T8 and covers a path: T1, T4, T5, T7, T8.

2.2.1 Control Flow Oriented Test Selection Criteria

Each control flow oriented test selection criterion aims at generating a set of tests that covers all occurrences of control structures (e.g. node, edge, etc.) in a source code or a specification. There are various control flow oriented test selection criteria: all-nodes, all-edges, all-paths, decision coverage (branch coverage), condition coverage, decision/condition coverage and multiple condition coverage [25].
Let $P$ be a set of complete paths of a directed graph $G$,

- **All-nodes** (sometimes called statement coverage)
  
  $P$ satisfies the all-nodes criterion in $G$ if every node of $G$ is covered by $P$ at least once.

- **All-edges**
  
  $P$ satisfies the all-edges criterion in $G$ if every edge of $G$ is covered by $P$ at least once.

- **All-paths**
  
  $P$ satisfies the all-paths criterion in $G$ if every complete path of $G$ is included in $P$.

This criterion is not generally practical or feasible due to the existence of loops in $G$.

- **Decision Coverage** (sometimes called branch coverage)
  
  A decision is a Boolean expression which is composed of one or more conditions.

  $P$ satisfies the decision coverage criterion [25] (or branch coverage criterion) in $G$ if every possible outcome (i.e. branch) of each decision in $G$ is covered by $P$ at least once.

However, this criterion does not imply that all combinations of conditions in each decision are covered; thus some faults present in a particular condition may not be exposed. For example, for testing the decision ($X$ and $Y$), the decision coverage criterion can be satisfied by two tests: ($X$ = true, $Y$ = true) and ($X$ = true, $Y$ = false); however, neither tests causes $X$ to be false. This could mask a possible fault in condition $X$. Hence, a shortcoming of decision coverage is that it does not guarantee coverage of each condition within each decision.

- **Condition Coverage**

  $P$ satisfies the condition coverage criterion [25] in $G$ if each possible outcome of every condition of each decision in $G$ is covered by $P$ at least once.
This criterion does not guarantee that every possible outcome of each decision will be taken; therefore, it does not guarantee the coverage of all statements or all branches in the graph. For example, for testing the decision ($X$ or $Y$), two tests: ($X = \text{true}$, $Y = \text{false}$) and ($X = \text{false}$, $Y = \text{true}$) would satisfy the condition coverage criteria, but it would not satisfy the decision coverage criterion because it would fail to test the false outcome of the decision.

- Decision/Condition Coverage

$P$ satisfies the decision/condition coverage criterion [25] in $G$, if each outcome of each decision and each outcome of each condition in $G$ is covered by $P$ at least once.

Decision/condition coverage has been developed to overcome the deficiencies of decision coverage and condition coverage alone. This criterion ensures that each outcome of each decision and each outcome of each condition will be executed. However, the fault of using “and” instead of “or” in a decision may still not be revealed. For example, for testing the decision ($X$ and $Y$), two tests: ($X = \text{true}$, $Y = \text{true}$) and ($X = \text{false}$, $Y = \text{false}$) would satisfy decision, condition and decision/condition coverage, but it would be possible to replace the “and” with an “or” without affecting the result of the tests.

- Multiple Condition Coverage

$P$ satisfies the multiple condition coverage criterion [25] in $G$, if all possible combinations of condition outcomes in each decision in $G$ is covered by $P$ at least once.

The deficiencies of the previous criteria are eliminated by multiple condition coverage, since tests must be generated to exercise all possible combinations of all condition outcomes in each decision.
It is commonly agreed that all-nodes coverage is quite weak, and that the branch coverage criterion is a minimal standard of coverage in white box testing. Other criteria, such as multiple condition coverage, decision/condition coverage, and condition coverage, are difficult or even impossible to achieve in software of high complexity [2].

2.2.2 Data Flow Oriented Test Selection Criteria

Data flow oriented testing is based on data flow analysis, which establishes certain associations between definitions and uses of variables. Such associations are identified by tracking variables as they are defined and modified until they are used to compute values for other variables or generate output values. Data flow oriented test selection criteria require that each association between the definition of a variable and its uses to be exercised at least once during testing. The motivation behind the selection of tests based on the coverage of data flow associations is that faults in a system may lead to incorrect values and as the result of propagation through computations, an erroneous result may show up at the system’s output [27].

2.2.2.1 Data Flow Related Concepts

Data flow analysis was originally used for compiler optimization [14]. Its aim is to trace the variables of a given program as they are defined and used during the execution of the program. Data flow analysis classifies an occurrence of a variable as part of a statement associated with a node or as part of a Boolean expression associated with an edge in a digraph as a definition or a use [27, 15]. Data flow analysis techniques can be applied to the selection of test sequences from an EFSM by transforming the EFSM into a special digraph $G$, often called flowgraph. Details of this transformation can be found in [36].
applying this transformation to the EFSM of the simplified ATM system, one can obtain the flowgraph shown in Figure 2.2, where for each transition $t$, the input of $t$ is in a circled node, enabling predicate of $t$ is on the edge leading to the actions of the transition $t$ which are in a rectangular node.

![Figure 2.2 Flow Graph Generated from the EFSM of the Simplified ATM System](image)

A **definition** (called *def*) of a variable $v$ at node $n$ (denoted $d_{vn}$) is an occurrence of $v$ by which a value is assigned to $v$ (e.g., an occurrence of $v$ on the left hand side of an assignment statement, for instance, the definition of “*attempts*” at T1 in Figure 2.1, or an occurrence of $v$ in an input, for instance, the definition of “pin” and “b” at T1 in Figure 2.1). A use of a variable $v$ is an occurrence of $v$ by which the value of $v$ is referenced.
Each use is classified as a computation use (c-use) or a predicate use (p-use) [27, 15]. A c-use of a variable $v$ at node $n$ (denoted $c_n^v$) is a use of $v$ that occurs in the right hand side of an assignment statement, or in an output statement, for instance, the use of variable “attempts” at T2 in Figure 2.1. On the other hand, a p-use of a variable $v$ on edge $(i, j)$ (denoted $p_{(i,j)}^v$) is a use that occurs in Boolean expressions in an enabling predicate of a transition, for instance, the use of variable “pin” in the enabling predicate of T4 in Figure 2.1.

In a flowgraph, with each node $n$, we have a possibly empty sets $\text{def}(n)$ and $\text{c-use}(n)$ of definitions and c-uses of variables occurring at node $n$, respectively. With each edge $(i, j)$ in a flowgraph, we have a possibly empty set $p\text{-use}(i, j)$ of p-uses of variables occurring on the edge.

Table 2.1 gives the def sets and c-use sets of the variables in Figure 2.2. Table 2.2 gives the p-use sets of the variables in Figure 2.2.

<table>
<thead>
<tr>
<th>nodes</th>
<th>def sets</th>
<th>c-use sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>card(pin, b)</td>
<td>{pin, b}</td>
<td>\emptyset</td>
</tr>
<tr>
<td>PIN(p)</td>
<td>{p}</td>
<td>\emptyset</td>
</tr>
<tr>
<td>withdrawal(w)</td>
<td>{w}</td>
<td>\emptyset</td>
</tr>
<tr>
<td>deposit(d)</td>
<td>{d}</td>
<td>\emptyset</td>
</tr>
<tr>
<td>T1</td>
<td>{attempts}</td>
<td>\emptyset</td>
</tr>
<tr>
<td>T2</td>
<td>{attempts}</td>
<td>{attempts}</td>
</tr>
<tr>
<td>T3</td>
<td>\emptyset</td>
<td>\emptyset</td>
</tr>
<tr>
<td>T4</td>
<td>\emptyset</td>
<td>\emptyset</td>
</tr>
<tr>
<td>T5</td>
<td>{b}</td>
<td>{b, w}</td>
</tr>
<tr>
<td>T6</td>
<td>{b}</td>
<td>{b, d}</td>
</tr>
<tr>
<td>T7</td>
<td>\emptyset</td>
<td>{b}</td>
</tr>
<tr>
<td>T8</td>
<td>\emptyset</td>
<td>\emptyset</td>
</tr>
</tbody>
</table>
Table 2.2 p-use sets in Figure 2.2

<table>
<thead>
<tr>
<th>edges</th>
<th>p-use sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>(PIN(p), T2)</td>
<td>{attempts, p, pin}</td>
</tr>
<tr>
<td>(PIN(p), T3)</td>
<td>{attempts, p, pin}</td>
</tr>
<tr>
<td>(PIN(p), T4)</td>
<td>{p, pin}</td>
</tr>
</tbody>
</table>

A path \((n_1, n_2, \ldots, n_{m-1}, n_m)\) is a def-clear path wrt a variable \(v\) from node \(n_i\) to node \(n_m\) or from node \(n_i\) to edge \((n_{m-1}, n_m)\) if there are no definitions of \(v\) at nodes \(n_1\) to \(n_{m-1}\).

Based on def, c-use and p-use sets, Frankl and Weyuker [15] define a set of nodes 
\[
dcu(v, i) = \{\text{node } j \mid v \in c\text{-use}(j) \text{ and there is a def-clear path wrt } v \text{ from } i \text{ to } j\}
\]
for the (last) def of \(v\) at node \(i\) and a set of edges 
\[
dpu(v, i) = \{\text{edge } (j, k) \mid v \in p\text{-use}(j, k) \text{ and there is a def-clear path wrt } v \text{ from } i \text{ to } (j, k)\}
\]
for the (last) def of \(v\) at node \(i\).

Def-use associations captured by dcu and dpu sets are categorized into two classes namely def-c-use association and def-p-use association [15]. A def-c-use association is a triple \((i, j, v)\) where \(v \in \text{def}(i)\) and \(j \in \text{dcu}(v, i)\). A def-p-use association is a triple \((i, (j, k), v)\) where \(v \in \text{def}(i)\) and \((j, k) \in \text{dpu}(v, i)\). Accordingly, a def-use association is either a def-c-use association or a def-p-use association. A def-use association is also called a du-pair. A def-use association is feasible (executable) if a def-clear path related to the association is a sub-path of some executable complete path; otherwise, it is infeasible (unexecutable).

A path \((n_i, \ldots, n_j, n_k)\) is a du-path [27] wrt variable \(v\) if \(n_i\) has a definition of \(v\) and either

- \(n_i\) has a c-use of \(v\) and \((n_i, \ldots, n_j, n_k)\) is a def-clear loop-free path wrt \(v\), or
- \((n_i, n_k)\) has p-use of \(v\) and \((n_i, \ldots, n_j, n_k)\) is a def-clear loop-free path wrt \(v\).

For example, in Figure 2.2, consider a def-c-use association \((\text{card(pin, b)}, T6, b)\). The definition of \(b\) at \(\text{card(pin, b)}\) can reach the c-use of \(b\) at \(T6\) through the def-clear path wrt
b “card(pin, b), T1, S1, PIN(p), T4, S2, deposit(d), T6”. As another example, consider a
def-p-use association (card(pin, b), (PIN(p), T3), pin) in Figure 2.2. The definition of pin
at card(pin, b) can reach the p-use of pin on the edge (PIN(p), T3) through the def-clear
path wrt pin “card(pin, b), T1, S1, PIN(p), T3”.

For the example EFSM in Figure 2.1, the def-c-use associations and the def-p-use
associations for each def of each variable as well as the corresponding def-clear paths are
given in Table 2.3.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Def(s)</th>
<th>Use(s)</th>
<th>def-clear path</th>
</tr>
</thead>
<tbody>
<tr>
<td>attempts</td>
<td>$d_{T1}$</td>
<td>$c_{T2}^{\text{attempts}}$, $P_{(\text{PIN}(p),T2)}^{\text{attempts}}$</td>
<td>T1 T2</td>
</tr>
<tr>
<td></td>
<td>$d_{T2}$</td>
<td>$p_{(\text{PIN}(p),T2)}^{\text{attempts}}$, $c_{T2}^{\text{attempts}}$, $p_{(\text{PIN}(p),T3)}$</td>
<td>T2 T2, T2 T3</td>
</tr>
<tr>
<td>b</td>
<td>$d_{T1}^b$</td>
<td>$c_{T5}^b$</td>
<td>T1 T4 T5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$c_{T6}^b$</td>
<td>T1 T4 T6</td>
</tr>
<tr>
<td></td>
<td>$d_{T5}^b$</td>
<td>$c_{T5}^b$</td>
<td>T5 T7 T5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$c_{T6}^b$</td>
<td>T5 T7 T6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$c_{T7}^b$</td>
<td>T5 T7</td>
</tr>
<tr>
<td></td>
<td>$d_{T6}^b$</td>
<td>$c_{T5}^b$</td>
<td>T6 T7 T5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$c_{T6}^b$</td>
<td>T6 T7 T6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$c_{T7}^b$</td>
<td>T6 T7</td>
</tr>
<tr>
<td>d</td>
<td>$d_{T6}^d$</td>
<td>$c_{T6}^d$</td>
<td>T6</td>
</tr>
<tr>
<td>p</td>
<td>$d_{T2}^p$</td>
<td>$P_{(\text{PIN}(p),T2)}^p$</td>
<td>T2</td>
</tr>
<tr>
<td></td>
<td>$d_{T3}^p$</td>
<td>$P_{(\text{PIN}(p),T3)}^p$</td>
<td>T3</td>
</tr>
<tr>
<td></td>
<td>$d_{T4}^p$</td>
<td>$P_{(\text{PIN}(p),T4)}^p$</td>
<td>T4</td>
</tr>
<tr>
<td>pin</td>
<td>$d_{T1}^{\text{pin}}$</td>
<td>$P_{(\text{PIN}(p),T2)}^{\text{pin}}$</td>
<td>T1 T2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$P_{(\text{PIN}(p),T3)}^{\text{pin}}$</td>
<td>T1 T3</td>
</tr>
<tr>
<td>Criterion</td>
<td>Association Required</td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------------------</td>
<td>--------------------------------------------------------------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>All-defs</td>
<td>Some ((i, j, x)) such that (j \in \text{dcu}(x, i)) or some ((i, (j, k), x)) such that ((j, k) \in \text{dpu}(x, i)).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>All-c-uses</td>
<td>All ((i, j, x)) such that (j \in \text{dcu}(x, i)).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>All-p-uses</td>
<td>All ((i, (j, k), x)) such that ((j, k) \in \text{dpu}(x, i)).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>All-p-uses/some-c-uses</td>
<td>All ((i, (j, k), x)) such that ((j, k) \in \text{dcu}(x, i)). In addition, if (\text{dpu}(x, i) = \phi) then some ((i, (j, k), x)) such that ((j, k) \in \text{dcu}(x, i)). Note that since (i) has a definition of (x), (\text{dpu}(x, i) = \phi \Rightarrow \text{dcu}(x, i) \neq \phi).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>All-c-uses/some-p-uses</td>
<td>All ((i, j, x)) such that (j \in \text{dcu}(x, i)) and all ((i, (j, k), x)) such that ((j, k) \in \text{dpu}(x, i)).</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2.2.2.2 Data Flow Oriented Test Selection Criteria

The motivation behind data flow oriented testing is to bridge the gap between all-paths and all-edges coverage criteria [27]. A family of applicable data flow oriented test selection criteria is defined in [15]. The criteria attempt to cover various combinations between defs and uses of the same variable. The family of data flow oriented test selection criteria is based on the requirement that the tests execute def-clear paths from each node containing a definition of a variable to the specified nodes containing c-uses and edges containing p-uses of that variable. For each variable definition, it is required that all/some def-clear paths wrt that variable from the node containing the definition to all/some of the uses/c-uses/p-uses reachable by some such paths be executed. These criteria are defined precisely in Table 2.4 [15].

Table 2.4 Some Data Flow Oriented Test Selection Criteria
| All-du-paths | All du-paths from $i$ to $j$ wrt $x$ for each $j \in \text{dcu}(x, i)$ and all du-paths from $i$ to $(j, k)$ wrt $x$ from each $(j, k) \in \text{dpu}(x, i)$. |

If variable $x$ has a definition in node $i$, the all-defs criterion requires the tests to exercise some def-clear path wrt $x$ from node $i$ to some node or edge at which the value assigned to $x$ in node $i$ is used. The all-uses criterion requires the tests to exercise at least one such path to each such node and to each such edge. The all-du-paths criterion requires that all of the du-paths from node $i$ to each such node and each such edge be exercised. The all-p-uses, all-c-uses, all-p-uses/some-c-uses, and all-c-uses/some-p-uses criteria lay the emphasis upon either c-uses or p-uses. Note that any program has only a finite set of def-use association, so none of the data flow oriented test selection criteria requires an infinite number of tests. Figure 2.3 illustrates the subsumption relationships between control and data flow oriented test selection criteria. A criterion $X$ subsumes another criterion $Y$, denoted $X \Rightarrow Y$, if a set of complete paths $P$ satisfying $X$ also satisfies $Y$. As we can observe from Figure 2.3, every data flow oriented test selection criterion is stronger than all-edges criterion. [15, 27] provide more details about the advantages of data flow oriented testing over control flow oriented testing.
The family of data flow oriented test selection criteria mentioned above is based on the association between a definition of a variable and each use of that variable or so-called "du-pair". However, there exists other types of relationship between variable occurrences. Such a relationship exists between the use of a variable and the def of another variable. IPO-def-Chains coverage criterion [37] takes both relations into account. Such relationships have been investigated by Ntafos’s required-k-tuples, Laski and Korel’s data-context, and Ural’s IPO-def-chains coverage criteria. Interested readers may refer to [26, 20, 37].

2.3 Regression Testing Techniques
During software maintenance the existing software is modified frequently. There are three types of software maintenance: Corrective maintenance is performed to correct an error that has been discovered in some part of the software. Adaptive maintenance is
usually performed when the software is modified to ensure its compatibility with the new environment in which it will operate. Perfective maintenance is performed to add new features to the software or to improve performance of the software [18].

A study performed by Hetzel indicates that the probability of introducing an error during program modification is between 50 and 80 percent. Therefore, in all types of maintenance regression testing is performed [18].

Regression testing is the process of validating that the changes introduced in a system are correct and do not adversely affect the unchanged portion of the system. A typical regression test proceeds as follows [29]:

Let $P$ be a procedure or program, let $P'$ be a modified version of $P$, and let $T$ be a test suite for $P$.

1. Select $T' \subseteq T$, a set of tests to execute on $P'$.
2. Test $P'$ with $T'$, establishing $P''$'s correctness wrt $T'$.
3. If necessary, create $T''$, a set of new functional or structural tests for $P'$.
4. Test $P'$ with $T''$, establishing $P''$'s correctness wrt $T''$.
5. Create $T'''$, a new test suite and test history for $P'$, from $T$, $T'$, and $T''$.

Based on the three types of software maintenance, regression testing can be classified into two categories [18]:

1. Corrective regression testing: is performed when the modification does not involve a change in the software specification.
2. Progressive regression testing: is performed when the modification does involve a change in the software specification.
In this thesis, we focus on the progressive regression testing that is performed when the modified version of the software involves a change in the specification.

One regression testing strategy, referred to as a retest all strategy, reruns all tests from the suite of existing tests created during software development. In some situations additional tests may be also created to satisfy selected testing criteria. This strategy is usually very expensive because of the large number of tests involved [18]. An alternative strategy, referred to as a selective retest strategy, selects a subset of tests that seem necessary to test the modified program from the existing test suite. Additional tests may be also created if selected coverage criteria are not satisfied. Evidently, selective retest strategy tries to reduce (minimize) the number of tests used to test the modified program [18]. If the cost of selecting a reduced subset of tests to run is less than the cost of running the tests that we omit, the selective retest technique is more economical than the retest-all technique [22].

The selection of suitable tests can be made in different ways, and a number of regression testing techniques and algorithms have been described in testing literature. We here briefly list some of them:

- Linear Equation Techniques

Fischer presents a selective retest technique that uses systems of linear equations to select test suites that yield segment coverage of the modified code. Linear equation techniques use systems of linear equations to express relationships between tests and program segments. The techniques obtain systems of equations from matrices that track program segments reached by tests, segments reachable from other segments, and (optionally) definition-use information about the segments [13].
• The Path Analysis Technique

Benedusi, Cimitile, and De Carlini present a selective retest technique based on path analysis. This technique takes as input the set of program paths in $P'$ expressed as an algebraic expression, and manipulates that expression to obtain a set of cycle-free exemplar paths: acyclic complete paths of the program. The technique then compares exemplar paths from $P$ to exemplar paths from $P'$, and classifies paths as new, modified, cancelled, or unmodified. Next, the technique analyzes tests to see which exemplar paths they traverse in $P$. The technique selects all tests that traverse modified exemplar paths [3].

• The Firewall Technique

Leung and White present a selective retest technique directed specifically at interprocedural regression testing that handles both code and specification changes. Their technique determines where to place a firewall around modified code modules. Where test selection from $T$ is concerned, the technique selects unit tests for modified modules that lie within the firewall, and integration tests for groups of interacting modules that lie again within the firewall [23].

• Slicing Techniques

Agrawal et al. define a family of selective retest techniques that use slicing. For each test $t \in T$, each technique constructs a slice. The authors discuss four different slice types: execution slice, dynamic slice, relevant slice, and approximate relevant slice. Given slice $s$ for test $t$, constructed by one of the four slicing techniques, if $s$ contains a modified statement, the technique selects $t$ [1].
• Graph Walk Techniques

Rothermel and Harrold present an intraprocedural regression test selection technique that builds control flow graphs (CFGs) for a procedure or program and its modified version and use these graphs to select tests that execute changed code from $T$. Their algorithm handles all language constructs of procedural languages and all types of program modifications. Rothermel and Harrold also present versions of their techniques that add data dependence information to CFGs to facilitate more precise test selection [31, 32, 33].

However, these regression testing techniques are code-based. They select tests using the source code of the original and modified programs. Code-based regression testing is often limited to unit testing and does not scale up to the system level because of the significant size of the source code involved. There exist some black-box regression testing techniques that are on the system level:

Briand, L.C., Labiche, Y., and Soccar, G., present a regression test selection technique that selects test cases based on UML designs. They propose a formal mapping between design changes and a classification of regression test cases, i.e., three categories: (1) reusable: a test case that is still valid but needs not to be rerun for the regression testing. (2) retestable: a test case that is still valid but needs to be rerun for the regression testing. (3) obsolete: a test case that cannot be executed on the new version of the system as it is ‘invalid’ in that context [6]. This technique reduces the number of regression test cases by classifying the existing test suite, then selecting retestable test cases for the regression testing.
Korel, B., Tahat, L. H., and Vaysburg, B., present a requirement-based regressing testing technique [19]. This technique is appropriate for state-based systems that can be modeled using formal description languages like Extended Finite State Machine (EFSM), in which testers want to partially test the system wrt a set of selected requirements, particularly on those modified requirements. The details of the technique will be reviewed in the next section.

2.3.1 Requirement-Based Regression Testing Strategies

Generally, in large and evolving software systems, specifications are frequently changed for a variety of reasons: to correct errors in specifications, to enhance or change functionality, to add new functionality, or to delete some existing functionality. Specification changes are typically made on a requirement level, i.e. system engineers make changes to a specification by changing individual requirements. For example, a new requirement is added, an existing requirement is modified, or an existing requirement is deleted. Since changes to the specification are made at the requirement level, developers would like to use regression testing techniques that are related to the changes at the requirement level. Unfortunately, the existing regression testing techniques generally do not support this type of testing except the one introduced in [19]. The technique in [19] identifies the differences between the original requirements and the new requirements as a set of elementary modifications. It identifies two types of elementary modifications at the requirement level: (1) addition of a new requirement, and (2) deletion of an existing requirement. It claims that any other requirement modification can be expressed as a set of these two types of elementary modifications. It states that the
sequence with which these elementary modifications are applied to the original model is not relevant.

When a given set of requirements is represented as an EFSM, an addition of a transition may occur between existing states or may involve an introduction of a new state when a transition is added to the EFSM. Similarly, a deletion of a transition may, in some cases, result in the deletion of a state. Notice, however, that an addition of a state and a deletion of a state are always associated with an addition of a transition and a deletion of a transition, respectively. Therefore, addition of a new state or a deletion of an existing state is not considered as an elementary modification.

In the remainder of this subsection we present a review of regression testing strategies related to individual requirement changes.

- Addition of a New Requirement

The main goal here is to test whether the new requirement performs expected functionality. The problem of testing a new requirement is equivalent to selective testing wrt the new requirement. An addition of a requirement is reflected in a new EFSM model as a new transition or a new EFSM sub-model connected to the existing states of the original EFSM model. Selective testing of the new transitions in the new EFSM model is used to test the “added” transitions.

For example, a new requirement of balance inquiry transaction is added, referred to as r5, into the simplified ATM system. The added requirement r5 can be adequately represented by adding a single transition T9 into the EFSM model of this simplified ATM system (The modified EFSM model is shown in Figure 2.4):
r5: After entering a correct pin in less than four attempts, user selects balance function. The system displays the balance, and then a menu with withdrawal, deposit, balance inquiry, and exit functions.

$$r5 = x T4 T9$$

Testing whether the new requirement r5 performs expected functionality is equivalent to testing the added transition T9.

![Figure 2.4 EFSM Model of a Simplified ATM System with Added Transition T9](image)

- Deletion of the Existing Requirement

The main goal here is to test whether the removal of a requirement does not cause unintended affects. For this reason all tests containing a deleted requirement should be used to test a new version of the system. A deletion of a requirement corresponds to a deletion of a transition(s) and possibly states from the original EFSM model. For this type of modification, "dummy" transitions and states corresponding to the deleted transitions and states are introduced into the new EFSM model. Selective testing of "dummy" transitions in the new EFSM model is used to test the "deleted" transitions. The
goal of this type of regression testing is to test whether the system performs correctly in situations when the removed transitions (requirement) were involved.

For example, the requirement \textit{r3} is deleted from the example simplified ATM system: \textit{r3}: After entering a correct pin in less than four attempts, user selects deposit function.

The system adjusts the balance, and displays a menu with withdrawal, deposit, balance inquiry, and exit functions.

\[ r3 = x \text{ T4 T6} \]

The deletion of this requirement corresponds to a deletion of transition T6 from the original EFSM model (Figure 2.1). When T6 is deleted, a "dummy" transition corresponding to this deleted transition is introduced into the new EFSM model shown in Figure 2.5. Testing the "dummy" transition in the new EFSM model is used to test the "deleted" transition T6. The goal of this type of regression testing is to test whether the system performs correctly in situations when the removed requirement \textit{r3} is involved. Testing whether the system performs correctly in situation when the removed requirement \textit{r3} is involved is equivalent to testing the "dummy" transition.

![Figure 2.5 EFSM Model of a Simplified ATM System with Deleted Transition T6](image_url)

26
2.3.2 Requirement-Based Regression Test Suite Reduction Techniques

The selective regression testing strategy is frequently used to generate regression test suites [19]. However, the size of these regression test suites still may be very large even for relatively small systems. In this thesis, we focus on requirement-based regression test suite reduction, which are dealing only with retestable test cases.

In this research, we present an extension to the existing requirement-based regression test suite reduction approach of [19] that uses the EFSM model dependence analysis to reduce a given regression test suite. [19] concentrates on the use of EFSM models for the representation of system requirements. This approach is based on the difference between the original model and the modified model expressed as a set of elementary model modifications: elementary addition of a transition and elementary deletion of a transition. As in [19], in this thesis we assume that any other modifications to the requirements can be expressed as a set of these two types of elementary modifications. When these elementary modifications are applied to the original model, the resulting model is expected to be the same as the modified model [19].

When a model is modified, three types of model-based regression testing need to be performed:

1. Test the effect of the model on the modification (modified part of the model),

2. Test the effect of the modification on the remaining part of the system model, and

3. Test side effects caused by the modification [19].

During test execution, different elements of a system interact with each other, and different tests may exercise different interactions. Therefore, the goal is to test different
interactions or patterns of interactions between functional elements of the system wrt a transition that represents an elementary modification.

The three types of regression testing identified above introduce three types of interaction patterns related to each modification:

(1) Affecting interaction patterns,
(2) Affected interaction patterns, and
(3) Side-effect interaction patterns.

This approach uses EFSM dependence analysis to capture three potential interactions between EFSM transitions. The potential interactions are then used to reduce an existing regression test suite by eliminating repetitive test cases.

The approach presented in [19] is one of the techniques that can be used at the system level. It can significantly reduce the size of a given regression test suite. However, the approach in [19] is not complete and accurate. The approach is not formally defined. In this thesis, we formalized the approach, defined new dependences, and developed algorithms. We also implemented the approach.

We will discuss, in detail, the approach of regression test suite reduction using EFSM dependence analysis in Chapters 4 and Chapter 5. Before we present this approach, we introduce dependencies that may exist in the EFSM model in Chapter 3.
Chapter 3

Dependence Analysis in the EFSM Model

In the EFSM model, two types of dependencies, namely data dependence and control dependence may exist between transitions. Dependence analysis focuses on interactions between transitions in the EFSM model.

3.1 Data Dependence

Data dependence captures the notion that one transition defines a value to a variable and the same or another transition may potentially use this value [19]. Formally, data dependence is defined as follows:

Let $T_i$ and $T_j$ be transitions and $v$ be a variable in the EFSM model. There is a \textit{data dependence} from $T_i$ to $T_j$ wrt a variable $v$ iff:

1. $v$ is defined in $T_i$ as the last definition of $v$,
2. $v$ is used (a c-use or p-use of $v$) in $T_j$ (before $v$ is (possibly) defined in $T_j$),
3. There is a path (transition sequence) from $T_i$ to $T_j$ along which $v$ is not redefined.

Such a path is referred to as \textit{definition-clear} (def-clear) \textit{path} wrt $v$, as defined earlier.

Note that $T_i$ and $T_j$ can be the same transition, i.e. $T_i = T_j$.

Figure 3.1 shows the graphical representation of a data dependence.
For example, in the example EFSM model (Figure 2.1), $T_i$ defines variable $b$, $T_5$ uses $b$, and along $T_1$ ($T_2$) $T_4$ $T_5$, $b$ is not redefined (there exists a def-clear path wrt variable $b$ from $T_i$ to $T_5$). Thus, there exists a data dependence from $T_i$ to $T_5$ wrt $b$. Also, $T_7$ defines $b$, $T_2$ uses $b$, and along $T_7$ $T_4$ $T_5$, $b$ is not redefined. Therefore, a data dependence exists from $T_i$ to $T_7$ wrt $b$, as shown in Figure 3.2.

3.2 Control Dependence

Control dependence captures the notion that one transition may "influence" the traversal of another transition. Control dependence between transitions is defined in terms of the concept of post-dominance [19].

Let $Y$ and $Z$ be two states, and $T$ be an outgoing transition from $Y$ in the EFSM model:

1. $Z$ post-dominates $Y$ iff $Z$ is on every path from $Y$ to the exit state.
2. $Z$ post-dominates $T$ iff $Z$ is on every path from $Y$ to the exit state through transition $T$.

Figure 3.3 shows the graphical representation of a post-dominance.

![Figure 3.3 Post-Dominance](image)

Let $T_i$ and $T_j$ be two outgoing transitions from $Y$ and $Z$, respectively (see Figure 3.4). There is a control dependence from $T_i$ to $T_j$ iff $Z$ does not post-dominate $Y$ and $Z$ post-dominates $T_i$.[19]

![Figure 3.4 Control Dependence](image)

In our example EFSM model (Figure 2.1), $S_2$ does not post-dominate $S_1$ but $S_2$ post-dominates $T_4$. Therefore, there is a control dependence from $T_4$ to $T_5$ (Figure 3.5).

![Figure 3.5 Control Dependence in the Example EFSM in Figure 2.1](image)
3.3 Static Dependence Graph

Data dependencies and control dependencies in the EFSM model can be graphically represented in a Static Dependence Graph (SDG). In SDG, nodes represent EFSM transitions and directed edges represent data or control dependencies. Figure 3.6 shows Static Dependence Graph of the EFSM model of Figure 2.1.

![Figure 3.6 Static Dependence Graph (SDG) for the Example EFSM in Figure 2.1](image)

Figure 3.6 Static Dependence Graph (SDG) for the Example EFSM in Figure 2.1

Given an original EFSM model $R_o$, its SDG $S_o$ is obtained by applying the definitions given in sections 3.1 and 3.2. An algorithm for constructing $S_o$ from a given $R_o$ is presented in [9]. This algorithm (and the ones that will be proposed in our thesis) utilizes the following data structures for $R_o$ and $S_o$:

Let

$$R_o \text{ be } (\text{name, startState, exitState, \{transition\}}) \text{ where}$$

name denotes the name of $R_o$,
$startState$ denotes the start state of $R_O$,

$exitState$ denotes the exit state of $R_O$,

transition denotes $(Label, Source, Dest, ActionList, VOList)$ where

$Label$ denotes the label of a transition,

$Source$ denotes the source state of a transition by the index of the state,

$Dest$ denotes the destination state of a transition by the index of the state,

$ActionList$ denotes the list of the actions in a transition,

$VOList$ denotes $\{(OfType, v, Label, OOrder) \mid \text{variable } v \text{ occurs in the transition as } OType \text{ in order } OOrder\}$ where

$OfType$ denotes the type of occurrence which is either a definition denoted $def$, or a c-use denoted $c$-use, or a p-use denoted $p$-use, and

$OOrder$ denotes the order of occurrence of variable $v$, which is numbered from top to bottom and from right to left in the textual representation of the transition. Elements in $VOList$ set are referenced by an index between 0 and size of $VOList$ – 1, corresponding to their order in the set. For example, in transition $T_1$ of the EFSM shown in Figure 2.1, the variable $b$ occurs as a definition in the order of occurrence of 1, the variable $pin$ occurs as a definition in the order of occurrence of 2, and the variable $attempts$ occurs as a definition in the order of occurrence of 3.

$S_O$ be a matrix where each entity $S_O[T_i, T_j]$ is either an empty set or a set containing one or more $(v, dependenceType, v_i)$ where

$T_i, T_j$ are transition numbers in $R_O$

$dependenceType$ denotes the type of dependence which is one of the following:
1) a data dependence: $dd$

2) a control dependence: $cd$

3) a du-pair based on a c-use: $dcu$-pair

4) a du-pair based on a p-use: $dpu$-pair

$v_y$ denotes ($oo\_def$, $oo\_use$) where

variable $v$ occurs in $T_i$ as a definition in the order of occurrence $oo\_def$, and in $T_j$ as a use, in the order of occurrence $oo\_use$

### 3.4 Algorithm for Generating the SDG of the Modified EFSM

When a system is modified, its EFSM model is modified. There are some new dependencies that do not exist in the original EFSM model which may exist in the modified EFSM model. Therefore, we need to construct the SDG of the modified EFSM after we obtain the modified EFSM.

Let

$R_O$ be the original EFSM model

$S_O$ be the SDG of $R_O$

$M$ be \{ $m$ | $m$ is an elementary modification $\}$

$m$ be ($MType$, $Trans$) where

$MType$ denotes the modification type of a transition, which is either an addition or a deletion,

$Trans$ denotes the added transition ($Label$, $Source$, $Dest$, $ActionList$, $VOList$) or denotes the transition number of the deleted transition where

$Label$ denotes the label of the transition,

$Source$ denotes source state by the index of the state,
\textit{Dest} denotes destination state by the index of the state,

\textit{ActionList} denotes the list of the actions in this transition,

\textit{VOList} denotes the variable and occurrence list in this transition.

\(R_M\) be the modified EFSM model

Then,

Let \(R_M\) be \(R_O\)

for each \(m \in M\)

if \(m\) is an addition of a transition \(T_i\)

add \(T_i\) to \(R_M\)

end if

if \(m\) is a deletion of a transition \(T_i\)

delete \(T_i\) from \(R_M\)

and

add a dummy self-loop transition \(T_{i, dummy}\) to \(R_M\) at the starting state of \(T_i\)

end if

end for

Now we have the modified EFSM \(R_M\), the algorithm for generating the SDG \(S_M\) of \(R_M\) is given as follows:

\textbf{Input:} \(R_M, S_O, M\)

\textbf{Output:} \(S_M\)

\textbf{Data Structure:}

Let

\(S_O\) be a matrix as defined in Section 3.3
\( V \) be \( \{ v \mid v \text{ is a variable in } R_M \} \)

\( S_M \) be a matrix where each entity

\( S_M[T_i, T_j] \) is in the form of \((f_1, f_2, f_3)\) where

- \( T_i, T_j \) are transition numbers in \( R_M \)
- \( f_1, f_2 \) denote status information given below, relevant to diagonal elements of \( S_M[T_i, T_i] \)
- if \( T_i \) is an added transition, put \textit{added} in \( f_1 \), \textit{empty} in \( f_2 \) in \( S_M[T_i, T_i] \)
- if \( T_i \) is a deleted transition, put \textit{deleted} in \( f_1 \), \( T_{i,\text{dummy}} \) in \( f_2 \) in \( S_M[T_i, T_i] \)
- if \( T_i \) is \( T_{i,\text{dummy}} \), put \textit{replacing} in \( f_1 \), \( T_i \) in \( f_2 \) in \( S_M[T_{i,\text{dummy}}, T_{i,\text{dummy}}] \)
- all other elements of \( S_M[T_i, T_j] \) will have both of these two fields empty i.e., \( \varepsilon \)
- \( f_3 \) is either an empty set or a set containing one or more \((t, v, \textit{dependenceType}, v_j)\)

where

- \( t \) is the transition number which is related to \( m \)
- \( \textit{dependenceType} \) denotes the type of dependence which is either one of those defined for \( S_O \) or the new dependencies that are introduced by elementary modifications.
- \( v_j \) denotes \((oo\_def, oo\_use)\) where variable \( v \) in \( V \) occurs in transition \( T_i \) in \( R_M \) as a \textit{def}, in the order of occurrence \( oo\_def \), and in \( T_j \) in \( R_M \) as a \textit{use}, in the order of occurrence \( oo\_use \)

Algorithm SDG\_\( S_M \)

Let

\[
\text{Def}(v) = \{ t \mid t \text{ is a transition in } R_M \text{ which defines } v \},
\]

\[
\text{Use}(v) = \{ t \mid t \text{ is a transition in } R_M \text{ which uses } v \}.
\]
\textit{VOList}(T) \textit{denotes the variable and occurrence list in transition} T

\textbf{Steps:}

1. Copy each entry of \( S_O \) into \( S_M \) such that each element of \( S_O \) is reformatted as expressed above

2. \textit{for each} \( m \in M \)
   
   \text{\begin{itemize}
   \item \text{if} \( m \) \text{ is an addition of a transition} \( T_i \)
   \begin{itemize}
   \item add a new row and new column of \( T_i \) into \( S_M \)
   \item mark \( T_i \) as \textit{added} at the first field of \( S_M[T_i, T_i] \)
   \item and put \textit{empty} at the second field of \( S_M[T_i, T_i] \)
   \end{itemize}
   \text{\end{itemize}}

   \text{\begin{itemize}
   \item \text{if} \( m \) \text{ is a deletion of a transition} \( T_i \)
   \begin{itemize}
   \item mark \( T_i \) as \textit{deleted} at the first field of \( S_M[T_i, T_i] \)
   \item and put \( T_i_{\text{dummy}} \) at the second field of \( S_M[T_i, T_i] \)
   \item and
   \item add a new row and new column of \( T_i_{\text{dummy}} \) into \( S_M \)
   \item mark \( T_i_{\text{dummy}} \) as \textit{replacing} at the first field of \( S_M[T_{i_{\text{dummy}}}, T_{i_{\text{dummy}}} ] \)
   \item and put \( T_i \) at the second field of \( S_M[T_{i_{\text{dummy}}}, T_{i_{\text{dummy}}} ] \)
   \end{itemize}
   \text{\end{itemize}}

3. Generate \text{Def}(v) and \text{Use}(v) for each \( v \in V \) in \( R_M \)
   
   \text{* Def}(v) and \text{Use}(v) are generated by using Olfa Chemli’s Def-UseAlg [9].*

4. \text{\textit{for each}} \( m \in M \)
   
   \text{\begin{itemize}
   \item \text{* Determine new dependencies introduced by each elementary modification*}
   \item \text{if} \( m \) \text{ is an addition of a transition} \( T_i \)
   \item then determine affecting data dependence(s) from \( T_j \) to \( T_i \)
   \end{itemize}
determine affected data dependence(s) from $T_i$ to $T_j$

determine affected control dependence(s)

determine activation dependence(s)

else /* $m$ is a deletion of a transition $T_i$ */

search column $T_i$ to identify and mark affecting ghost data dependence(s)

search row $T_i$ to identify and mark affected ghost data dependence(s)

determine ghost activation dependence(s)

The definitions and detailed algorithms for new dependencies introduced by each elementary modification are presented in the remainder of this chapter. Addition of a transition may eliminate an existing control dependence, deletion of a transition may add a new control dependence. In our work, we assume that this is not the case.

3.4.1 Algorithm for Affecting Data Dependence

In the modified model, addition of a transition may introduce new data dependencies in the modified model, which represent the affect of the original EFSM model on the added transition: each of which we call affecting data dependence, and define as follows:

Let $T_i$ be an added transition, and $v$ be a variable in the EFSM model. There is an affecting data dependence from $T_j$ to $T_i$ wrt a variable $v$ iff there is a data dependence from $T_j$ to $T_i$ wrt the variable $v$.

Figure 3.7 shows the graphical representation of an affecting data dependence from $T_j$ to $T_i$ wrt variable $v$. 

38
Algorithm Affecting_DD

/* determine affecting data dependence(s) from $T_j$ to $T_i$ because of addition of $T_i$ */

for each $(use, v, T_i, oo_use)$ in VOList($T_i$)

    for each transition $T_j$ in Def($v$)

        for each $(def, v, T_j, oo_def)$ in VOList($T_j$)

            if $(def, v, T_j, oo_def)$ in VOList($T_j$) is the last def of $v$ in $T_j$

                and

                there is no def of $v$ in $T_i$ before $(use, v, T_i, oo_use)$ in VOList($T_i$)

                and

                there is a def-clear path from $T_j$ to $T_i$ wrt $v$ in $R_M$

                then /*there is an affecting data dependence from $T_j$ to $T_i$ wrt $v$ in $S_M$*/

                    dependenceType $\leftarrow$ affecting_dd

                    insert ($e$, $v$, dependenceType, $(oo_def, oo_use)$) in the 3rd field of $S_M$[T_jT_i]

                end if

        end for

    end for

end for
3.4.2 Algorithm for Affecting Control Dependence

In the modified model, addition of a transition may introduce new control dependencies in the modified model which represent the affect of the original EFSM model on the added transition: each of which we call affecting control dependence, and define as follows:

Let $T_i$ be an added transition in the EFSM model. There is an affecting control dependence from $T_j$ to $T_i$ iff there is a control dependence from $T_j$ to $T_i$.

Figure 3.8 shows the graphical representation of an affecting control dependence from $T_j$ to $T_i$.

$$
\text{Control dependence (} T_j, T_i \text{)}
$$

Figure 3.8 Affecting Control Dependence

Algorithm Affecting_CD

/* determine affecting control dependence(s) from $T_j$ to $T_i$ because of addition of $T_i$ */

for each state $y$ in $R_M$

if the starting state of $T_i$ does not post-dominate $y$

and

for each outgoing transition $T_j$ of state $y$

if the starting state of $T_i$ post-dominates $T_j$

then /* there is an affecting control dependence from $T_j$ to $T_i$ */

dependenceType $\leftarrow$ affecting_cd

insert ($e$, nil, dependenceType, nil) in the 3rd field of $S_M[T_j, T_i]$

end if
end for

end if

end for

3.4.3 Algorithm for Affected Data Dependence

In the modified model, addition of a transition may introduce new data dependencies in the modified model, which represent the affect of the added transition on the original EFSM model: each of which we call affected data dependence, and define as follows:

Let $T_i$ be an added transition, and $v$ be a variable in the EFSM model. There is an affected data dependence from $T_i$ to $T_j$ wrt a variable $v$ iff there is a data dependence from $T_i$ to $T_j$ wrt the variable $v$.

Figure 3.9 shows the graphical representation of an affected data dependence from $T_i$ to $T_j$ wrt the variable $v$.

![Figure 3.9 Affected Data Dependence](image)

Algorithm Affected_DD

/* determine affected data dependence(s) from $T_i$ to $T_j$ because of addition of $T_i$ */

for each $(def, v, T_i, oo_def)$ in VOList($T_i$)

    for each $(use, v, T_j, oo_use)$ in VOList($T_j$)

        if $(def, v, T_i, oo_def)$ in VOList($T_i$) is the last def of $v$ in $T_i$

            and

        end if

    end for

end for
there is no def of v in Tj before (use, v, Tj, oo_use) in VOList(Tj)

and

there is a def-clear path from Ti to Tj wrt v in R_M

then /* there is an affected data dependence from Ti to Tj wrt v in S_M */

dependenceType ⇐ affected_dd

insert (ε, v, dependenceType, (oo_def, oo_use)) in the 3rd field of

S_M[Ti, Tj]

end if

end for

end for

3.4.4 Algorithm for Affected Control Dependence

In the modified model, addition of a transition may introduce new control dependencies in the modified model, which represent the affect of the added transition on the original EFSM model: each of which we call affected control dependence, and define as follows:

Let Ti be an added transition in the EFSM model. There is an affected control dependence from Ti to Tj iff there is a control dependence from Ti to Tj.

Figure 3.10 shows the graphical representation of an affected control dependence from Ti to Tj.

![Figure 3.10 Affected Control Dependence](image)

42
Algorithm Affected_CD
/* determine affected control dependence(s) from $T_i$ to $T_j$ because of addition of $T_i$ */

for each state $z$ in $R_M$

if state $z$ does not post-dominate the starting state of $T_i$

and

state $z$ post-dominates $T_i$

then /* there is an affected control dependence from $T_i$ to $T_j$ */

dependenceType $\leftarrow$ affected_cd

insert ($z$, nil, dependenceType, nil) in the 3^{rd} field of $S_M[T_i, T_j]$

end if

end for

3.4.5 Algorithm for Activation Dependence
Side effects introduced by the addition of a transition are captured by the dependence referred to as an activation dependence [19] and is defined as follows: Suppose a new transition $T_i$ has been added to the original EFSM model. Due to introduction of transition $T_i$, a new data dependence may arise from $T_j$ to $T_k$ wrt variable $v$ in the modified EFSM model that does not exist in the original EFSM model. More formally: there exists an activation dependence from $T_i$ to the data dependence from $T_j$ to $T_k$ wrt variable $v$ iff

(1) There exists a definition-clear path from $T_j$ to $T_i$, wrt $v$ in the modified model,

(2) $T_i$ is triggered, and

(3) There exists a definition-clear path from $T_i$ to $T_k$ wrt $v$ [19].

Figure 3.11 shows the graphical representation of an activation dependence from $T_i$ to the data dependence from $T_j$ to $T_i$ wrt variable $v$. 

43
Algorithm Activation_Dependence

/* Determine activation dependence(s) because of addition of $T_i$ */

for each $v \in V$

for each transition $T_j$ in Def($v$)

for each $(\text{def}, v, T_j, \text{oo_def})$ in VOList($T_j$)

for each transition $T_k$ in Use($v$)

for each $(\text{use}, v, T_k, \text{oo_use})$ in VOList($T_k$)

if $(\text{def}, v, T_j, \text{oo_def})$ in VOList($T_j$) is the last def of $v$ in $T_j$

and

there is no def of $v$ in $T_k$ before $(\text{use}, v, T_k, \text{oo_use})$ in VOList($T_k$)

and

there is a def-clear path from $T_j$ to $T_k$ wrt $v$ in $R_M$

and

$S_0[T_j, T_k]$ does not have $(v, dd, v_k)$

then there is a new data dependence from $T_j$ to $T_k$ wrt $v$

for each $m \in M$
if the def-clear path from $T_j$ to $T_k$ wrt $v$ contains one (or more) $T_i$ in $R_M$

then /* there is an activation dependence from $T_i$ to $(T_j, T_k, v)$ */

$dependenceType \leftarrow ad$

insert $(T_i, v, dependenceType, (oo\_def, oo\_use))$

in the 3\text{rd} field of $S_M[T_j, T_i]$

else /* the new data dependence must have been an affecting or affected data dependence */

end for

def for

end for

def for

end for

end for

3.4.6 Algorithm for Affecting Ghost Data Dependence

Deletion of a transition can cause elimination of dependencies associated with the deleted transition where the deleted transition was dependent on another transition. The eliminated dependence is referred to as the \textit{affecting ghost dependence} [19]. The affecting ghost dependence is defined as follows: Suppose $T_i$ is a deleted transition. There exists an affecting ghost data dependence from $T_j$ to $T_i$ wrt a variable $v$ in the modified EFSM model, iff:
(1) There is a data dependence from $T_j$ to $T_i$ wrt to a variable $v$, in the original EFSM model,

(2) There exists a definition-clear path from $T_j$ to the starting state of $T_i$ wrt $v$, in the modified EFSM model,

(3) Event $E(T_j)$ occurs and condition $C(T_i)$ evaluates to TRUE at the starting state of $T_i$ [19].

Note that to satisfy (3), a dummy self-loop transition is added to the modified EFSM model at the starting state of $T_i$ such that it only has $E(T_j)$ and $C(T_i)$ of the deleted $T_i$ [19].

Figure 3.12 shows the graphical representation of an affecting ghost data dependence from $T_j$ to $T_i$ wrt variable $v$.

![Figure 3.12 Affecting Ghost Data Dependence](image)

Algorithm Affecting GDD

/* Search column $T_i$ to identify and mark affecting ghost data dependence(s) */

search column $T_i$

for each row $T_j$ of $S_M$

for each $dd$ from $T_j$ to $T_i$ wrt $v$, i.e., ($\varepsilon$, $v$, $dd$, (oo_def, oo_use)) in the 3rd field of $S_M[T_j, T_i]$

if there is a def-clear path from $T_j$ to the starting state of $T_i$ wrt $v$ in $R_M$

then /* there is an affecting ghost data dependence from $T_j$ to $T_i$ wrt $v$ in $S_M$ */
dependenceType ← affecting_gdd

change (ε, ν, dd, (oo_def, oo_use)) in the 3rd field of \( S_M[T_j, T_i] \) to (ε, ν, dependenceType, (oo_def, oo_use))

end if

end for

end for

3.4.7 Algorithm for Affected Ghost Data Dependence

Deletion of a transition can cause elimination of the dependence associated with the deleted transition where some transition was dependent on the deleted transition. The eliminated dependence is referred to as an affected ghost data dependence [19]. This dependence is defined as follows: Suppose \( T_i \) is a deleted transition. There exists an affected ghost data dependence from \( T_i \) to \( T_j \) wrt to a variable \( ν \) iff:

(1) Event \( E(T_i) \) occurs and condition \( C(T_i) \) evaluates to TRUE at the starting state of \( T_i \),

(2) Suppose transition \( T_k \) is triggered (\( T_k \) must be a newly added transition due to the deletion of \( T_i \)),

(3) There exists a definition-clear path from \( T_k \) to \( T_j \) wrt \( ν \) in the modified model, and

(4) \( T_i \) does not have a definition of \( ν \) (a case when \( T_k \) has a definition of \( ν \) is covered by testing the addition of \( T_i \) as this is a ‘new’ dependence) [19].

Figure 3.13 shows the graphical representation of an affected ghost data dependence from \( T_i \) to \( T_j \) wrt variable \( ν \).
Algorithm Affected_GDD

/* search row $T_i$ to identify and mark affected ghost data dependence(s) */

search row $T_i$

for each column $T_j$ of $S_M$

for each $dd$ from $T_i$ to $T_j$ wrt $v$, i.e., ($e$, $v$, $dd$, (oo_def, oo_use)) in the 3\textsuperscript{rd} field of $S_M[T_i, T_j]$

if there is a def-clear path from $T_{i, dummy}$ to $T_j$ wrt $v$ in $R_M$

and

transition $T_{i, dummy}$ not in Def($v$)

then /*there is an affected ghost data dependence from $T_i$ to $T_j$ wrt $v$ in $S_M$

*/

dependenceType $\leftarrow$ affected_gdd

change ($e$, $v$, $dd$, (oo_def, oo_use)) in the 3\textsuperscript{rd} field of $S_M[T_i, T_j]$ to ($e$, $v$,

dependenceType, (oo_def, oo_use))

end if

end for

end for
3.4.8 Algorithm for Ghost Activation Dependence

Deletion of a transition may cease dependencies that previously exist in the original EFMS model. Suppose that transition $T_i$ has been deleted from the original EFMS model and there exists a data dependence from $T_j$ to $T_k$ wrt variable $v$ in the original EFMS model. Due to deletion of transition $T_i$, the data dependence from $T_j$ to $T_k$ wrt to $v$ ceases to exist in the modified EFMS model. This dependence between $T_i$ and the data dependence from $T_j$ to $T_k$ is referred to as a *ghost activation dependence* [19]. More formally: there exists a ghost activation dependence from $T_i$ to the data dependence from $T_j$ to $T_k$ wrt $v$ iff:

1. There exists a definition-clear path from $T_j$ to the starting state of $T_i$ wrt $v$,
2. Event $E(T_i)$ occurs and condition $C(T_i)$ evaluates to TRUE at the starting state of $T_i$,
3. Suppose transition $T_n$ is triggered ($T_n$ is a newly added transition due to the deletion of $T_i$), and
4. There exists a definition-clear path from $T_n$ to $T_k$ wrt $v$ [19].

Figure 3.14 shows the graphical representation of a ghost activation dependence from $T_i$ to the data dependence from $T_j$ to $T_k$ wrt variable $v$.

![Figure 3.14 Ghost Activation Dependence](image)
Algorithm Ghost_Activation_Dependence

/* determine ghost activation dependencies from $T_i$ to $(T_j, T_k, v)$ */

for each row $T_j$ in $S_M$

    for each column $T_k$ in $S_M$

        for each $dd$ $(T_j, T_k, v)$, i.e., $(e, v, dd, (oo\_def, oo\_use))$ in the 3rd field of $S_M[T_j, T_k]$

            for each $m \in M$ which is for a deletion of a transition $T_i$

                if $(T_j \neq T_i$ and $T_k \neq T_i)$$\quad$ and

                there is a def-clear path from $T_j$ to the starting state of $T_i$ wrt $v$ in $R_M$$\quad$ and

                there is def-clear path from $T_{i, dummy}$ to $T_k$ wrt $v$ in $R_M$

                then /* there is a ghost activation dependence from $T_i$ to $(T_j, T_k, v)$ */

                    $dependenceType \leftarrow gad$

                    change $(e, v, dd, (oo\_def, oo\_use))$ in the 3rd field of $S_M[T_j, T_k]$$\quad$ to $(T_i, v, dependenceType, (oo\_def, oo\_use))$

                end if

            end for

        end for

    end for

end for
3.5 SDG of the Simplified ATM System with Modified Transactions

Consider our example EFSM model of the simplified ATM system (Figure 2.1). The Balance Inquiry transaction is added to the simplified ATM system, and the Deposit transaction is deleted from the simplified ATM system. The added transaction is represented by transition T9 in the modified EFSM model of Figure 3.15. The deleted transaction is represented by transition T6 in the original EFSM model, which does not appear in the modified EFSM model of Figure 3.15. Notice that transition T6dummy is added to the EFSM model to indicate that the deposit event is handled in state S2 and represents an empty operation. It is possible to imitate traversal of the deleted transition when the event associated with the deleted transition is generated and the enabling condition of the deleted transition evaluates to TRUE. This is considered as testing of the deleted transition [19].

Figure 3.15 EFSM Model of the Simplified ATM System with Added Transition T9 and Deleted Transition T6
All dependencies in the modified EF$^3$M model are graphically represented in a new SDG. Figure 3.16 shows SDG of the modified EF$^3$M model of Figure 3.15.

![Figure 3.16 SDG of the Simplified ATM System with Modified Transactions](image)
Chapter 4

Dependence Analysis for Regression Test Suite Reduction

When a model is modified, three types of regression testing need to be performed:

(1) Test the effect of the model on the modification (modified part of the model),

(2) Test the effect of the modification on the remaining part of the model, and

(3) Test side effects caused by the modification.

Since there are three types of regression testing, three types of interaction patterns related to each elementary modification are introduced:

(1) an affecting interaction pattern,

(2) an affected interaction pattern, and

(3) a side-effect interaction pattern.

The affecting interaction pattern captures interactions between model elements that affect the elementary modification. The affected interaction pattern captures interactions between model elements that are affected by the elementary modification. The side-effect interaction pattern captures interactions between model elements that occur because of side effects introduced by the elementary modification.

In this thesis, EFSM dependences are used to identify equivalent test cases in a given regression test suite. During traversal of a test case, for each elementary modification, three interaction patterns (sub-SDG) are computed: (1) affecting, (2) affected, and (3) side-effect interaction pattern. If the same interaction pattern of a certain type is computed for two different test cases for an elementary modification, these test cases are considered equivalent, wrt the elementary modification and the interaction pattern. In
Section 4.1 and Section 4.2, testing the addition of a transition and testing the deletion of a transition are discussed.

4.1 Testing the Addition of a Transition

During the traversal of a test case, first a sub-SDG is constructed from the SDG $S_M$ where all dependencies that are encountered are identified. Then, the dependencies related to the three types of interaction patterns for the transition under test (elementary modification) are identified as follows:

(1) In the sub-SDG, dependencies that “affect” the added transition are identified by traversing backwards from the added transition and marking all traversed data and control dependencies. All unmarked dependencies are removed from the sub-SDG. The resulting sub-SDG is referred to as an Affecting Interaction Pattern;

(2) In the sub-SDG, dependencies that are “affected” by the added transition are identified by traversing forward from the added transition and marking all traversed data and control dependencies. All unmarked dependencies are removed from the sub-SDG. The resulting sub-SDG is referred to as a Side-Effect Interaction Pattern.

For example, consider the case where the Balance Inquiry transaction is added to the simplified ATM system, i.e. add transition $T9$ to the EFSM of the simplified ATM
system (Figure 2.1). Suppose the regression test suite contains the following two test cases:

Test_1 = T1, T4, T9, T7, T5, T7, T9, T7, T8, and

Test_2 = T1, T2, T4, T9, T7, T5, T7, T9, T7, T8,

During traversal of each of these two test cases, first a sub-SDG is constructed from the SDG of the modified EFSM. The sub-SDGs of Test_1 and Test_2 are presented in Figure 4.1 (a) and 4.1 (b) respectively. Then, for each test case, three interaction patterns are computed. Both of these two test cases result in the same affecting interaction pattern shown in Figure 4.2. Also, both of these test cases result in the same affected interaction pattern shown in Figure 4.3, and the same side-effect interaction pattern shown in Figure 4.4. Since, these two test cases are equivalent wrt all three interaction patterns and the modification of added transition T9, either Test_1 or Test_2 can be eliminated.
4.2 Testing the Deletion of a Transition

During the traversal of a test case, first a sub-SDG is constructed from the SDG $S_M$ where all dependencies that are encountered are identified. Then, the dependencies related to the three types of interaction patterns for the transition under test (elementary modification) are identified as follows:

(1) In the sub-SDG, dependencies that “affect” the deleted transition are identified by traversing backwards from the deleted transition through the affecting ghost data dependencies and marking all traversed data and control dependencies. All unmarked dependencies are removed from the sub-SDG. The resulting sub-SDG is referred to as an Affecting Interaction Pattern;
(2) In the sub-SDG, dependencies that are “affected” by the deleted transition are identified by traversing forward from the deleted transition through the affected ghost data dependencies and marking all traversed data and control dependencies. All unmarked dependencies are removed from the sub-SDG. The resulting sub-SDG is referred to as an Affected Interaction Pattern;

(3) In the sub-SDG, dependencies that are “affected” by the deleted transition are identified by traversing forward from the deleted transition through the marked ghost activation dependencies and marking all traversed data and control dependencies. All unmarked dependencies are removed from the sub-SDG. The resulting sub-SDG is referred to as a Side-Effect Interaction Pattern.

Let us consider another example, where the Deposit transaction is deleted from the simplified ATM system, i.e. delete transition T6 from the EFMS of simplified ATM system (Figure 2.1). Suppose the regression test suite contains the following two test cases:

\[
\text{Test}_1 = T1, \ T4, \ T6\text{dummy}, \ T5, \ T7, \ T8, \text{ and}
\]

\[
\text{Test}_2 = T1, \ T2, \ T4, \ T6\text{dummy}, \ T5, \ T7, \ T8
\]

During traversal of each of these two test cases, first a sub-SDG is constructed from the SDG of the modified EFMS. The sub-SDGs of Test_1 and Test_2 are presented in Figure 4.5 (a) and 4.5 (b) respectively. Then, for each test case, three interaction patterns are computed. Both of these two test cases result in the same affecting interaction pattern shown in Figure 4.6. Also, both of these test cases result in the same affected interaction pattern shown in Figure 4.7, and there are no side-effect interaction patterns for Test_1
and Test_2. Since these two test cases are equivalent wrt all three interaction patterns and the modification of deleted transition T6, either Test_1 or Test_2 can be eliminated.

To implement this approach, we break down the solution strategy we will use as follows:

First of all, for each elementary modification, we obtain a subset of regression test suite associated with a \textit{test} that related to an elementary modification.

Second, for each regression test case, we obtain three interaction patterns.

Third, the interaction patterns are then used to reduce the regression test suite.
Besides the reduced regression test suite, we also obtain a collection of a set of distinct interaction patterns wrt each \textit{tut}. The following is the detail description:

Given $S_M$, \textit{RTS}, \textit{TUT} where

- $S_M$ denotes the SDG of the EFSM modified by a set $M$ of elementary modifications
- $TUT$ denotes a set of transitions under test related to $M$
- \textit{tut} denotes a transition under test in \textit{TUT} related to an elementary modification in $M$
- \textit{RTS} denotes a set of regression test cases for \textit{TUT}
- \textit{rts} denotes a regression test case in \textit{RTS},

Let

- $RTS_{\textit{tut}}$ denote the subset of \textit{RTS} associated with a \textit{tut}
- $P\text{atternSet}_{\textit{tut}}$ denote a set of distinct interaction patterns for a \textit{tut}
- $P\text{atternSet}$ denote the collection of the $P\text{atternSet}_{\textit{tut}}$
- $RRTS$ denote the reduced regression test suite for \textit{TUT}$
- $ERSTS_{\textit{tut}}$ denote the eliminated regression test suite associated with a \textit{tut}

Then,

$$\forall \textit{tut} \in \textit{TUT},$$
$$\forall \textit{rts} \in \textit{RTS}_{\textit{tut}},$$

obtain $RTS_{\textit{tut}}$ from $RTS$

 obtain three interaction patterns, namely affecting interaction pattern, affected interaction pattern, and side-effect interaction pattern

/* use interaction patterns to reduce regression test suite */
if there is one or more patterns that are not included in $\text{PatternSet}_{rus}$

insert the pattern(s) into $\text{PatternSet}_{rus}$ and

insert the $rts$ into $\text{RTS}$

else /* all three interaction patterns are included in $\text{PatternSet}_{rus}$ */

insert the $rts$ into $\text{ERTS}_{rus}$

append $\text{PatternSet}_{rus}$ into $\text{PatternSet}$

To implement this solution strategy, two algorithms have been developed in this thesis. They are presented in the remainder of this chapter: Section 4.3 describes the algorithm for reducing regression test suite. Section 4.4 describes the algorithm for generating interaction patterns for a given test sequence.

**4.3 Algorithm for Reducing Regression Test Suite**

In this algorithm, we use the interaction patterns to reduce the regression test suite. A test case is included in the reduced regression test suite if at least one of its interaction patterns does not exist for any of the test cases in the reduced regression test suite. In order to reduce the complexity, we separate the $\text{PatternSet}_{rus}$ into three subsets, namely $\text{Pattern1Set}$, $\text{Pattern2Set}$, and $\text{Pattern3Set}$, respectively because these three types of interaction patterns cannot be the same. The details of the algorithm are given as follows:

**Input:** $S_M$, $RTS$, $TUT$

**Output:** $\text{RRTS}$, $\text{PatternSet}$

Let

$\text{Pattern1Set}$ be \{Pattern1 | Pattern1 is Affecting Interaction Pattern\},

$\text{Pattern2Set}$ be \{Pattern2 | Pattern2 is Affected Interaction Pattern\},

$\text{Pattern3Set}$ be \{Pattern3 | Pattern3 is Side-effect Interaction Pattern\},
Steps:

\[ RRTS \leftarrow \emptyset \]

for each \( tut \in TUT \)

obtain \( RTS_{tut} \) from \( RTS \)

\[ Pattern1Set \leftarrow \emptyset, \ Pattern2Set \leftarrow \emptyset, \ Pattern3Set \leftarrow \emptyset, \ ERTS \leftarrow \emptyset \]

for each \( rts \in RTS_{tut} \)

using algorithm that will be described in Section 4.4, obtain three patterns, namely \( Pattern1 \), \( Pattern2 \), and \( Pattern3 \)

counter \( \leftarrow 0 \)

if \( Pattern1 \not\in Pattern1Set \)

\[ \text{then } Pattern1Set \leftarrow Pattern1Set \cup \{\text{pattern1}\} \]

else counter \( \leftarrow \) counter + 1

if \( Pattern2 \not\in Pattern2Set \)

\[ \text{then } Pattern2Set \leftarrow Pattern2Set \cup \{\text{Pattern2}\} \]

else counter \( \leftarrow \) counter + 1

if \( Pattern3 \not\in Pattern3Set \)

\[ \text{then } Pattern3Set \leftarrow Pattern3Set \cup \{\text{Pattern3}\} \]

else counter \( \leftarrow \) counter + 1

if counter = 3 // all three patterns are in respective PatternSets

\[ \text{then } ERTS_{tut} \leftarrow ERTS \cup \{rts\} \]

else \( RRTS \leftarrow RRTS \cup \{rts\} \)

end for

append \( Pattern1Set \), \( Pattern2Set \), and \( Pattern3Set \) into \( PatternSet \)
end for

4.4 Algorithm for Generating Interaction Patterns for a Given Test Sequence

In this algorithm, we have been given a test case (test sequence). During traversal of this test case, three interaction patterns are computed for each elementary modification: (1) an affecting interaction pattern, (2) an affected interaction pattern, and (3) a side-effect interaction pattern. The details of the interaction patterns generation algorithm are given as follows:

**Input:** $S_M, \text{tut, rts}$ (regression test sequence)

**Output:** the affecting interaction pattern, affected interaction pattern, and the side-effect interaction pattern for $\text{rts}$

Let $dd$ stands for $dd, \text{affecting}\_dd$, or $\text{affected}\_dd$

$cd$ stands for $cd, \text{affecting}\_cd$, or $\text{affected}\_cd$

$T$ be a transition in the $\text{rts}$

$T_{\text{tut}}$ be the transition $\text{tut}$ (i.e., transition under test) which is the transition related to $m$

**Steps:**

1. During traversal of a test case $\text{rts} \in RTS_{\text{tut}}$, construct sub-SDG $G$ as a graph induced by a set $E$ of dependence edges, i.e., $G[E]$. $E$ is a subset of the set of dependence edges in $S_M$ as follows:

/* if $T \in \text{rts}$ is a dummy transition due to a deleted transition $T_i$ then use $T_i$ instead of $T$ */

Loop from the last transition (say $T_k$) of $\text{rts}$ to the second transition of $\text{rts}$

Loop from the second last transition of $\text{rts}$ (say $T_j$) to the first transition of $\text{rts}$
if there is one or more $dd$ wrt $v$ in $S_M[T_j, T_k]$

and there is no $dd/ad$ edge wrt $v$ incoming to $T_k$

add a $\longrightarrow$ edge $(T_j, T_k)$ to $E$

if there is one or more $cd$ in $S_M[T_j, T_k]$

and there is no $cd$ edge incoming to $T_k$

add a $\longrightarrow$ edge $(T_j, T_k)$ to $E$

if there is one or more $affecting\_gdd$ wrt $v$ in $S_M[T_j, T_k]$

and $T_k$ is the deleted transition related to $T_{int}$

and there is no $affecting\_gdd$ edge wrt $v$ incoming to $T_k$

add a $\longrightarrow$ edge $(T_j, T_k)$ to $E$

if there is one or more $affected\_gdd$ wrt $v$ in $S_M[T_j, T_k]$

and $T_j$ is the added transition related to $T_{int}$

and there is no $affected\_gdd$ edge wrt $v$ incoming to $T_k$

add a $\longrightarrow$ edge $(T_j, T_k)$ to $E$

if there is one or more $ad$ wrt $v$ from $T$ to $(T_j, T_k)$ in $S_M[T_j, T_k]$

and $T$ is the added transition related to $T_{int}$

and there is no $ad/dd$ edge wrt $v$ incoming to $T_k$

add a $\longrightarrow$ edge $(T_j, T_k)$ to $E$

and add a $\longrightarrow$ edge from $T$ to edge $(T_j, T_k)$ to $E$

if there is one or more $gad$ wrt $v$ from $T_i$ to $(T_j, T_k)$ in $S_M[T_j, T_k]$

and $T_i$ is the deleted transition related to $T_{int}$

and there is no $gad$ edge wrt $v$ incoming to $T_k$

add a $\longrightarrow$ edge from $T_i$ to edge $(T_j, T_k)$ to $E$
the resulting graph is $G[E]$

2. In $G[E]$, identify dependencies that “affect” $T_{\text{ut}}$ to obtain an **Affecting Interaction** Pattern:

   if $T_{\text{ut}}$ is an added transition
   
   traverse backwards from $T_{\text{ntr}}$ and mark all traversed $dd(s)$, $cd(s)$ in $G[E]$
   
   remove all unmarked dependencies from $G[E]$ to obtain an Affecting Interaction Pattern

   else /* $T_{\text{ut}}$ is a dummy transition $T_{i,\text{dummy}}$, corresponding to deleted transition $T_i$ */
   
   traverse backwards from $T_i$ through the $\text{affecting\_gdd}(s)$, and mark all traversed $dd(s)$ and $cd(s)$ in $G[E]$

   remove all unmarked dependencies from $G[E]$ to obtain an Affecting Interaction Pattern

3. In $G[E]$, identify dependencies that are “affected” by $T_{\text{ut}}$ to obtain an **Affected Interaction** Pattern:

   if $T_{\text{ut}}$ is an added transition
   
   traverse forward from $T_{\text{ntr}}$ and mark all traversed $dd(s)$, and $cd(s)$ in $G[E]$

   remove all unmarked dependencies from $G[E]$ to obtain an Affected Interaction Pattern

   else /* $T_{\text{ut}}$ is a dummy transition $T_{i,\text{dummy}}$, corresponding to deleted transition $T_i$ */
   
   traverse forward from $T_i$ through the $\text{affected\_gdd}(s)$, mark all traversed $dd(s)$ and $cd(s)$ in $G[E]$

   remove all unmarked dependencies from $G[E]$ to obtain an Affected Interaction Pattern
4. In $G[E]$, identify dependencies that are “affected” by $T_{nat}$ to obtain a **Side-Effect Interaction Pattern**:

   if $T_{nat}$ is an added transition
   
   traverse forward from $T_{nat}$ through the activation dependencies, and mark all traversed $dd(s)$ and $cd(s)$ in $G[E]$
   
   remove all unmarked dependencies from $G[E]$ to obtain a Side-Effect Interaction Pattern

   else /* $T_{nat}$ is a dummy transition $T_{i, dummy}$, corresponding to deleted transition $T_i */
   
   traverse forward from $T_i$ through the ghost activation dependencies, and mark all traversed $dd(s)$ and $cd(s)$ in $G[E]$
   
   remove all unmarked dependencies from $G[E]$ to obtain a Side-Effect Interaction Pattern

**Data Structure:**

$G = (N, E)$ where

$$N = \{ n_t \mid n_t \text{ is a node representing transition } T \text{ in rts, } n_0 \text{ denotes the entry node of } G,$$

$\quad n_{end} \text{ denotes the exit node of } G, \text{ and } n_{nat} \text{ denotes the transition } T_{nat} \text{ node in } G \}$$

and

$$E = \{ e \mid e = (n_s, n_d, \text{ type}), n_s, n_d \in N \} \text{ where}$$

$e$ is an edge representing a dependence between two transitions

$n_s$ and $n_d$ are the source node and destination node of $e$ respectively, and

*type* denotes the type of dependence which is one of the following:

- 1) data dependence, affecting data dependence, or affected data dependence:

  $dd$
2) control dependence, affecting control dependence, or affected control
dependence: cd
3) affecting ghost data dependence: affecting_gdd
4) affected ghost data dependence: affected_gdd
5) activation dependence: ad
6) ghost activation dependence: gad

Algorithm Interaction_Patterns

Let

\[ uniqueT = \{T \mid T \text{ is a transition in } rts \} \]

\[ \text{node}(i) \text{ denote a node whose index is equal to } i \text{ in } G, \]

\[ \text{finish}[] \text{ denote an array of integers.} \]

Steps:

\[ uniqueT \leftarrow \emptyset, \text{ sub-SDG } G \leftarrow (\emptyset, \emptyset) \]

1. for each transition \( T \) in \( rts \) from the first transition to the last transition, do

   /* if \( T \in rts \) is a dummy transition due to a deleted transition \( T_i \)

      then use \( T_i \) instead of \( T \) */

   if \( T \not\in uniqueT \) then

   \[ uniqueT \leftarrow uniqueT \cup \{T\} \]

end for loop

sort the elements of \( uniqueT \) in ascending order of transitions’ labels.

/* Each element of sorted \( uniqueT \) can be referred by its index e.g.,

\( uniqueT[0] \) denotes the first element of \( uniqueT \). */
/* construct sub-SDG G */

/* First, create nodes in sub-SDG G */

lastIndex ← the index of the last element in uniqueT

i = 0

for each T \in uniqueT from uniqueT[0] to uniqueT[lastIndex] do

/* if there is no node in N wrt T, add such a node in N */

if \( n_t \notin N \) then

\( N \leftarrow N \cup \{n_t\} \)

set the index of node \( n_t \) to \( i \)

\( i = i + 1 \)

end if

end for

/* Second, create edges in sub-SDG G */

Loop from the last transition to the second transition of \( rts \)

\( rts[k] \leftarrow \) the current transition in \( rts \)

\( n_k \leftarrow \) the node representing the transition \( rts[k] \) in \( N \)

Loop from the second last transition of \( rts \) to the first transition of \( rts \)

\( rts[j] \leftarrow \) the current transition in \( rts \)

\( n_j \leftarrow \) the node representing the transition \( rts[j] \) in \( N \)

if (there is a \( dd \) wrt \( v \) from \( rts[j] \) to \( rts[k] \) in \( S_M \))

and (there is no \( dd/ad \) edge wrt \( v \) incoming to \( n_k \) in \( G \))

\( E \leftarrow E \cup \{ (n_j, n_k, dd) \} \)

end if
if ((there is a cd from $rts[j]$ to $rts[k]$ in $S_M$)
and (there is no cd edge incoming to $n_k$ in $G$))

$E \leftarrow E \cup \{(n_j, n_k, cd)\}$

end if

if ((there is an affecting_gdd wrt $v$ from $rts[j]$ to $rts[k]$ in $S_M$)
and ($rts[k]$ is the deleted transition related to $T_{lut}$)
and (there is no affecting_gdd edge wrt $v$ incoming to $n_k$ in $G$))

$E \leftarrow E \cup \{(n_j, n_k, affecting_gdd)\}$

end if

if ((there is an affected_gdd wrt $v$ from $rts[j]$ to $rts[k]$ in $S_M$)
and ($rts[j]$ is the deleted transition related to $T_{lut}$)
and (there is no affected_gdd edge wrt $v$ incoming to $n_k$ in $G$))

$E \leftarrow E \cup \{(n_j, n_k, affected_gdd)\}$

end if

if ((there is an ad wrt $v$ from $T$ to ($rts[j], rts[k]$) in $S_M$)
and ($T$ is the added transition related to $T_{lut}$)
and (there is no ad/dd edge wrt $v$ incoming to $n_k$ in $G$))

$E \leftarrow E \cup \{(n_j, n_k, ad)\}$

/* Since this ad is related to $T_{lut}$, we don’t need to add an addition edge from $n_i$ to ($n_j, n_k$) here. We know this ad is activated by $T_{lut}$ */

end if

if ((there is a gad wrt $v$ from $T_i$ to ($rts[j]$ to $rts[k]$) in $S_M$)
and \((T_i\) is the deleted transition related to \(T_{lut}\))

and (there is no \(gad\) edge wrt \(\nu\) incoming to \(n_k\) in \(G\))

\[E \leftarrow E \cup \{(n_j, n_k, gad)\}\]

/* Since this \(gad\) is related to \(T_{lut}\), we don’t need to add an addition edge from \(n_i\) to \((n_j, n_k)\) here. We know this \(gad\) is activated by \(T_{lut}\) */

end if

end for

end for

2. In the sub-SDG \(G\), identify dependencies that “affect” \(T_{lut}\) to obtain an **Affecting Interaction Pattern**:

/* if \(T_{lut}\) is an added transition, \(n_{lut}\) represents this \(T_{lut}\) in \(G\), else if \(T_{lut}\) is a dummy transition \(T_{i_{dummys}}\) corresponding to deleted transition \(T_i\), \(n_{lut}\) represents this deleted transition \(T_i\) in \(G\). Therefore, we can obtain an Affecting Interaction Pattern by traversing backward from \(n_{lut}\) to \(n_0\) in \(G\), mark traversed nodes and edges, and removing all unmarked nodes and edges from \(G\) */

mark \(n_{lut}\) in \(N\)

\(i\) ← the index of \(n_{lut}\) in \(N\)

/* initialize the value of each element in finish[] equal to 0 */

\(l = \) the index of the last node in \(N\)

for \(i = 0\) to \(l\) do

\(finish[i] \leftarrow 0\)

Call traverse_backward( \(i, finish[]\) )

Remove all unmarked nodes and unmarked edges in \(G\)
The resulting graph $G$ is an Affecting Interaction Pattern.

3. In the sub-SDG $G$, identify dependencies that are “affected” by $T_{lut}$ to obtain an

Affected Interaction Pattern:

/* if $T_{lut}$ is an added transition, $n_{lut}$ represents this $T_{lut}$ in $G$, else if $T_{lut}$ is a dummy transition $T_{i_dumm}$, corresponding to deleted transition $T_i$, $n_{lut}$ represents this deleted transition $T_i$ in $G$, therefore, we can obtain an Affecting Interaction Pattern by traversing forward from $n_{lut}$ to $n_{end}$ in $G$, mark traversed nodes and edges, and removing all unmarked nodes and edges from $G$ */

mark $n_{lut}$ in $N$

$i$ ← the index of $n_{lut}$

$j$ ← the index of $n_{end}$ in $G$

/* initialize the value of each element in finish[] equal to 0 */

$l$ = the index of the last node in $N$

for $i = 0$ to $l$ do

    $finish[i] \leftarrow 0$

Call traverse_forward($i, finish[], j$)

Remove all unmarked nodes and unmarked edges in $G$

The resulting graph $G$ is an Affecting Interaction Pattern.

4. In the sub-SDG $G$, identify dependencies that are “affected” by $T_{lut}$ to obtain a Side-

Effect Interaction Pattern:

if $T_{lut}$ is an added transition
/* traverse forward from $n_{int}$ through the marked activation dependencies to $n_{end}$ in $G$, mark traversed nodes and edges. After the traversal, remove all unmarked nodes and edges in $G$ */
mark $n_{int}$ in $N$

for each $ad$ edge $e$ in $E$
    mark the $ad$ edge $e$ in $E$
    and mark $n_s$ and $n_d$ in $N$

$i \leftarrow$ the index of $n_d$

$j \leftarrow$ the index of $n_{end}$ in $G$

/* initialize the value of each element in finish[] equal to 0 */

$l =$ the index of the last node in $N$

for $i = 0$ to $l$ do

    $finish[i] \leftarrow 0$

call traverse_forward($i$, $finish[]$, $j$)

remove all unmarked nodes and unmarked edges in $G$

The resulting graph $G$ is a Side-Effect Interaction Pattern.

else /* $T_{int}$ is a dummy transition $T_{i\_dummy}$, corresponding to deleted transition $T_i$, 
traverse forward from $n_{int}$ through the marked ghost activation dependencies to $n_{end}$ in $G$, mark traversed nodes and edges. After the traversal, remove all unmarked nodes and edges in $G$ */
mark $n_{int}$ in $N$

for each $gad$ edge $e$ in $E$
    mark the $gad$ edge $e$ in $E$
and mark $n_s$ and $n_d$ in $N$

$i \leftarrow$ the index of $n_d$

$j \leftarrow$ the index of $n_{end}$ in $G$

/* initialize the value of each element in finish[] equal to 0 */

$l =$ the index of the last node in $N$

for $i = 0$ to $l$ do

$finish[i] \leftarrow 0$

call traverse_backward($i$, $finish[]$, $j$ )

remove all unmarked nodes and unmarked edges in $G$

The resulting graph $G$ is a Side-Effect Interaction Pattern.

**Procedure** traverse_backward($i$, $finish[]$) {

/**************************************************************************/ 

$finish[i] = 0$ designates that $node(i)$ has never been visited,

$finish[i] = -1$ designates that $node(i)$ has already been visited,

$finish[i] = 1$ designates all paths leading from $n_0$ to $node(i)$ have been already traversed

**************************************************************************/

if (finish[$i$] != 1) then

$finish[i] = -1$; /* to indicate that $node(i)$ has already been visited */

for each edge $e$ incoming to $node(i)$ do

if $e$ is not a ad/gad edge

then $s \leftarrow$ the source node of $e$

if $(e$ is unmarked) mark $e$
if (s is unmarked) mark s

k ← the index of node s

if ( (s ≠ n₀) /* s is not the source node in G */
    AND (finish[k] = 0) /* s has never been visited */
    AND (s ≠ node(i)) /* to avoid infinite recursion */

traverse_backward(k, finish[])

) // end traverse_backward

Procedure traverse_forward(i, finish[], j) {

/*****************************/

finish[i] = 0 designates that node(i) has never been visited,

finish[i] = -1 designates that node(i) has already been visited,

finish[i] = 1 designates all paths leading from node(i) to n_end have been already traversed

/*****************************/

if (finish[i] != 1) then

    finish[i] = -1; /* to indicate that node(i) has already been visited */

for each edge e outgoing from node(i) do

    if e is not a ad/gad edge

        then d ← the destination node of e

        if (e is unmarked) mark e

        if (d is unmarked) mark d

73
\[ k \leftarrow \text{the index of node } d \]

if \((d \neq n_{\text{end}}) /\!* d \text{ is not the end node in } G */\)

\[ \text{AND } (\text{finish}[k] = 0) /\!* d \text{ has never been visited } */\]

\[ \text{AND } (d \neq \text{node}(i)) /\!* \text{to avoid infinite recursion } */\]

traverse_forward\((k, \text{finish}[], j)\)

end for

\[ \text{finish}[i] = 1 \]

} // end traverse_forward

The above algorithm is correct because it follows the formal definitions given earlier and actually does what it is suppose to do, i.e., to obtain a reduced regression test suite \(RRTS\) from a given regression test suite \(RTS\) by eliminating repetitive test cases. We can prove it by this way:

Let's assume that there is a regression test case \(rts\), which is in \(RTS\) and should not be eliminated after applying this algorithm. Suppose that it is not in \(RRTS\). If this \(rts\) should not be eliminated, that means there is at least one of its interaction patterns that is not covered by any test case in the \(RRTS\). However, according to our algorithm, if there is one or more interaction patterns that are not covered by any test case in \(RRTS\), we insert the \(rts\) into \(RRTS\). This is contradiction with our assumption that \(rts\) is not in \(RRTS\). This proves that our algorithm is correct.

In the next chapter, we present the Regression Test Suite Reduction tool, which has been developed based on algorithms introduced in this chapter.
Chapter 5
RTSR Software Tool

5.1 RTSR Overview

Based on the algorithms for the requirement-based regression test reduction using
dependence analysis described in Chapter 3 and Chapter 4, a tool called Regression Test
Suite Reduction (RTSR) has been developed as a part of the Test Suite
Reduction/Generation software tool (TSRG) [17], which is implemented in Sun Solaris
Sparc 5.8 using C++ and Java 2 Platform.

RTSR is built to reduce the number of regression test cases in a given test suite by
employing interaction patterns computed for each test case. For each elementary
modification, during traversal of a test case, three interaction patterns are computed: (1)
affection interaction pattern, (2) affected interaction pattern, and (3) side-effect
interaction pattern. If the same interaction pattern of a certain type is computed for two
different test cases for an elementary modification, these test cases are considered
equivalent, wrt the elementary modification and the interaction pattern. A test case is
included in the reduced regression test suite if at least one of its interaction patterns does
not exist for any of the test cases in the reduced regression test suite. Figure 5.1 shows the
structure of the RTSR tool.
The RTSR performs the following tasks:

Phase 1: Construction of SDG $S_O$ and modified EFSM $R_M$

Phase 2: Construction of modified SDG $S_M$

Phase 3: Generation of Interaction Patterns

Phase 4: Construction of Reduced Regression Test Suite $RTTS$
The details of the implementations of these phases are presented in Section 5.4. Section 5.2 gives the input file formats of RTSR, and Section 5.3 gives the output file formats of RTSR.

It must be noted that the EFSM parser was developed by Tuong Nguyen to yield internal data structures EFSM $R_0$ and MOD $M$ [28]. The construction of SDG $S_0$ was implemented by Gao Yan based on the algorithm for generating static dependence graph from an EFSM in [9].

5.2 Input File Formats

As shown in Figure 5.1, RTSR requires three input files: EFSM input file, MOD input file, and RTS input file. The EFSM input file is a file that represents an EFSM model, which is given the "efsml" extension; the MOD input file is a file that represents a set of elementary modifications, which is given the "mod" extension; and the RTS input file is a file that represents a test suite, which is given the "ts" extension.

In this thesis, the EFSM input file is defined formally below using the Backus-Naur Form (BNF) [28]. Note that although some constructs of SDL such as set, reset, procedure are included in this definition, they are not considered in our work.

Table 5.1 BNF Definition of an EFSM Input File

```
<efsm> ::= 
  efsml
  numStates startStateIndex exitStateIndex
<transitions>
```

77
<transition> ::= 
  transition transitionId 
  sourceStateIndex destinationStateIndex 

<requirement> ::= 
  [ <input> ] 
  [ <enablingPredicate> ] 
  / 
  [ <actions> ] 

<actions> ::= 
  <action> | <actions> <action> 

$action> ::= 
  <output> | <assignment> | <set> | <reset> | <procedureCall> 

<input> ::= 
  inputId ( [ <parameters> ] ) 

<output> ::= 
  outputId ( [ <parameters> ] ) 

<enablingPredicate> ::= 
  <variableIds> [ /* BooleanExpression */ ] 

<assignment> ::= 
  <variableId> := <expression> 

;set> ::=
set ( constant, timerId )

/reset> ::= 

reset ( timerId )

procedureCall ::= 

procedure ( procedureId ( <variableIds> [ ; <variableIds> ] ) ) { <pbrDefs> }

<parameters> ::= 

<parameter> {<parameter>}*

<parameter> ::= 

<variableId> | constant

<variableIds> ::= 

<variableId> {<variableId>}*

<pbrDefs> ::= 

<pbrDef> | <pbrDefs> <pbrDef>

<pbrDef> ::= 

<variableId> := <expression> ;

<expression> ::= 

function ( <variableIds> ) | constant

<variableId> ::= id

The "efsm" file for the EFSM of the simplified ATM system in Figure 2.1 is given in Appendix A.6.

The MOD input file is a set of elementary modifications where each elementary modification is represented by a modification type and the modified transition. A
modification type is either an addition or a deletion. The MOD input file is defined formally below in BNF:

Table 5.2 BNF Definition of a MOD Input File

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;mod&gt;</td>
<td>:=</td>
</tr>
<tr>
<td></td>
<td>&lt;modTransitions&gt;</td>
</tr>
<tr>
<td>&lt;modTransitions&gt;</td>
<td>:=</td>
</tr>
<tr>
<td></td>
<td>&lt;modTransition&gt;</td>
</tr>
<tr>
<td>&lt;modTransition&gt;</td>
<td>:=</td>
</tr>
<tr>
<td></td>
<td>mtransition mType transitionId</td>
</tr>
<tr>
<td></td>
<td>/* mType (addition = 0, deletion = 1, unknown = 2) */</td>
</tr>
<tr>
<td></td>
<td>sourceStateIndex destinationStateIndex</td>
</tr>
<tr>
<td>&lt;requirement&gt;</td>
<td>:=</td>
</tr>
<tr>
<td></td>
<td>[&lt;input&gt;]</td>
</tr>
<tr>
<td></td>
<td>[&lt;enablingPredicate&gt;]</td>
</tr>
<tr>
<td></td>
<td>/</td>
</tr>
<tr>
<td></td>
<td>[&lt;actions&gt;]</td>
</tr>
<tr>
<td>&lt;actions&gt;</td>
<td>:=</td>
</tr>
<tr>
<td></td>
<td>&lt;action&gt;</td>
</tr>
<tr>
<td>&lt;action&gt;</td>
<td>:=</td>
</tr>
<tr>
<td></td>
<td>&lt;output&gt;</td>
</tr>
<tr>
<td>&lt;input&gt;</td>
<td>:=</td>
</tr>
<tr>
<td></td>
<td>inputId ( [&lt;parameters&gt;] )</td>
</tr>
<tr>
<td>&lt;output&gt;</td>
<td>:=</td>
</tr>
<tr>
<td></td>
<td>outputId ( [&lt;parameters&gt;] )</td>
</tr>
</tbody>
</table>
<enablingPredicate> ::=  
   <variableIds> [/* booleanExpression */]

<assignment> ::=  
   <variableId> ::= <expression>

<set> ::=  
   set ( constant , timerId )

<reset> ::=  
   reset ( timerId )

procedureCall ::=  
   procedure ( procedureId ( <variableIds> [; <variableIds>] ) ) { <pbrDefs> }

<parameters> ::=  
   <parameter> {, <parameter>}*

<parameter> ::=  
   <variableId> | constant

<variableIds> ::=  
   <variableId> {, <variableId>}*

<pbrDefs> ::=  
   <pbrDef> | <pbrDefs> <pbrDef>

<pbrDef> ::=  
   <variableId> ::= <expression> ;

<expression> ::=  
   function ( <variableIds> ) | constant

<variableId> ::=  
   id
An example “.mod” file for the EFSM of the simplified ATM system in Figure 2.1 is given in Appendix A.7.

The RTS input file consists of a set of transitions under test, each of which is the transition related to an elementary modification and a collection of regression test cases. A test case is a complete sequence of transitions that starts at the start state and ends at the exit state of the EFSM. The RTS input file is defined formally below in BNF [28]:

Table 5.3 BNF Definition of a TS Input File

```
<ts> ::= 
    efsmId <tuts> <tests>
<tuts> ::= 
    <tut> | <tuts> <tut>
<tut> ::= 
    transitionId
<tests> ::= 
    <test> | <tests> <test>
<test> ::= 
    test testId <transitionSeq>
<transitionSeq> ::= 
    transitionId | <transitionSeq> transitioned
```

An example “.ts” file for the EFSM of the simplified ATM system in Figure 2.1 is given in Appendix A.8.
5.3 Output File Formats

RTSR generates two output files: a Reduced RTS output file and an IP output file. The Reduced RTS file is a file that represents the reduced regression test suite where redundant test cases have been eliminated. A Reduced RTS file is given the “.rr.ts” extension and has the same format as the original RTS file.

The IP file is a file that represents a set of interaction patterns wrt each \textit{tut}. Each interaction pattern indicates a group of equivalent test cases that result in it. The test cases are referred to by their test id. The extension of the IP output file is “.rip”. The IP output file is defined formally below in BNF:

<table>
<thead>
<tr>
<th>Table 5.4 BNF Definition of an IP Output File</th>
</tr>
</thead>
</table>
| \[
\begin{align*}
\text{<ip>} &::= \\
& \quad \text{efsmId} \text{ <tut>} \text{ <ips>} \\
\text{<tut>} &::= \\
& \quad \text{transitionId} \\
\text{<ips>} &::= \\
& \quad \text{<ip>} | \text{ <ips>} \text{ <ip>} \\
\text{<ip>} &::= \\
& \quad \text{<ipType>} \text{ ipId} [\text{ <testIds>}] \text{ <nodes>} \\
\text{<ipType>} &::= \\
& \quad \text{ip\_affecting} | \text{ip\_affected} | \text{ip\_sideEffect} \\
\text{<testIds>} &::= \\
& \quad \text{TestId} | \text{<testIds>} \text{ TestId} \\
\text{<nodes>} &::= \\
\end{align*}
|
It is noted that an IP output file distinguishes three types of interaction patterns according to the ipType, i.e. “ip_affecting”, “ip_affected”, and “ip_sideEffect” that denote affecting interaction pattern, affected interaction pattern, and side-effect interaction pattern, respectively. The prefix of ipId “R” denotes that the three types of interaction patterns are generated for regression testing. An example “.rip” file for the EFSM of the simplified ATM system in Figure 2.1 is given in Appendix A.9.
5.4 RTSR Tool

RTSR uses EFSM model dependence analysis to reduce regression test suites. We assume that interactions between EFSM transitions are represented as EFSM dependencies between transitions. If the same interaction pattern of a certain type is computed for two different test cases for an elementary modification, these test cases are considered equivalent, wrt the elementary modification and the interaction pattern. A test case is included in the reduced test suite if at least one of its interaction patterns does not exist for any of the test cases in the reduced test suite [19]. RTSR can be broken down into four phases as mentioned in Section 5.1.

Phase 1: Construction of SDG $S_O$ and Modified EFSM $R_M$

Given two input files, EFSM file and MOD file, RTSR concatenates the EFSM file and MOD file, then analyzes the concatenated file by lexical parser, and forms the EFSM and MOD internal data structures $R_O$ and $M$, respectively. The SDG $S_O$ of the original EFSM $R_O$ can be built using EFSM internal data structure. The modified EFSM $R_M$ can be built using $R_O$ and $M$, which are described in Chapter 3. For example, from the EFSM input file given in Appendix A.6 and the MOD input file given in Appendix A.7, we can construct $S_O$ and $R_M$. Parts of $S_O$ and $R_M$ are shown in Table 5.5 and Table 5.6, respectively. Table 5.5 represents the dependencies existing from transition T1 to transition T2. Table 5.6 represents the modified EFSM with added transition T9 between state S2 and S3.

<table>
<thead>
<tr>
<th>EFSM id: ATM_System</th>
</tr>
</thead>
</table>

Original Static Dependency Graph (SDG):
Source node index: 0, Label: T1

Destination node index: 1, Label: T2

Number of edges: 3

List of edges: DType(Data=0,Control=1,CUse=2,PUse=3)

(Variable, DType, OOrder in def, OOrder in use)

(pin,0,2,3)

(attempts,0,3,2)

(attempts,0,3,5)

Table 5.6 An Example Internal Data Structure of Modified EFSM with Added Transition T9

EFSM id: ATM_System

Label: T9, Internal index: 8

Source state: 2

Destination state: 3

List of variables & occurrences: OType(Def=0,CUse=1,PUse=2)

(OType,Var,TLabel,OOrder)

(1,b,T9,1)

Number of components: 2

List of components:

AType(INPUT=0,OUTPUT=1,ASSIGN=2,SET=3,RESET=4,PRED=5,PROC=6)

Index,Id,AType,List(OType,Var,TLabel,OOrder)

0,Balance,0

1,Print,1,(1,b,T9,1)
Phase 2: Construction of Modified SDG $S_M$

In this phase, we have $S_o$, $R_M$, and $M$, and can construct the SDG $S_M$ of modified EFSM. The detailed algorithm for constructing $S_M$ is described in Chapter 3. A part of $S_M$ which represents the dependencies existing from transition T1 to transition T9 of the EFSM of the simplified ATM system in Figure 2.1 is shown in Table 5.7.

<table>
<thead>
<tr>
<th>Source node index: 0, Label: T1</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Destination node index: 8, Label: T9</td>
<td>3</td>
</tr>
<tr>
<td>MType (ADD=0, DEL=1, REP=2, MUK=3)</td>
<td></td>
</tr>
<tr>
<td>Related edge: N/A</td>
<td></td>
</tr>
<tr>
<td>Number of edges: 1</td>
<td></td>
</tr>
<tr>
<td>List of edges: DType (Data=0, Control=1, CUse=2, PUse=3, AFFNGDA=4, AFFEDDA=5, AD=6, AFFNGGAD=7, AFFEDGAD=8, GAD=9, AFFNGCO=10, AFFEDCO=11, DUK=12)</td>
<td></td>
</tr>
<tr>
<td>(Variable, DType, OOrder in def, OOrder in use)</td>
<td></td>
</tr>
<tr>
<td>(b, 4, 1, 1)</td>
<td></td>
</tr>
</tbody>
</table>

Phase 3: Generation of Interaction Patterns

In this phase, we have $S_M$. In order to generate interaction patterns, an RTS input file is required. As stated previously, an RTS input file, "ts" consists of a set of elementary modifications, which is represented by a set of transitions under test ($TUT$) and a collection of regression test cases ($RTS$) where each regression test case ($rts$) is a transition sequence starting at the start state and ending at the exit state of the corresponding EFSM. RTSR extracts $TUT$ and $RTS$. For each $tut$, RTSR obtains $RTS_{tut}$
which is the subset of $RTS$ associated with a $tut$. For each $rts$ in $RTS_{tn}$, three interaction
patterns are computed: affecting interaction pattern, affected interaction pattern, and side-
effect interaction pattern. The detailed algorithm for generating interaction patterns is
described in Chapter 4.

**Phase 4: Construction of Reduced Regression Test Suite $RRTS$**

In this phase, a Reduced RTS output file, ".rr.ts" and an IP output file, ".rip" are
constructed. For each elementary modification, we use Pattern1Set, pattern2Set, and
Pattern3Set to store three types of interaction patterns. From Phase 3, for each test case,
we obtained three interaction patterns, namely Pattern1, Pattern2, and Pattern3, which
will be used to reduce the original test suite. If there exists an interaction pattern that is
not in the Pattern1Set, Pattern2Set, or Pattern3Set, we insert this test case into the
reduced regression test suite. RTSR reports the reduced regression test suite in an output
file with ".rr.ts" extension. For each elementary modification, RTSR also identifies sets
of equivalent test cases wrt a certain type of interaction pattern, and report it in an IP
output file with ".rip" extension. For example, consider the addition of transition T9 to
the EFSM of the simplified ATM system (Figure 2.1). The $tut$ is T9 that represents an
elementary modification of adding transition T9. Suppose the regression test suite
contains the following two tests:

Test\_1 = $T1$, $T4$, $T9$, $T7$, $T5$, $T7$, $T9$, $T7$, $T8$, and

Test\_2 = $T1$, $T2$, $T4$, $T9$, $T7$, $T5$, $T7$, $T9$, $T7$, $T8$,

A part of $RRTS$ for Test\_1 and Test\_2 is shown in Table 5.8. A part of IP output file
shown in Table 5.9 represents the case that for transition under test T9, Test\_1 and
Test\_2 are equivalent wrt the affecting interaction pattern.
Table 5.8: An Example of Reduced Regression Test Suite in RRTS File (.rr.ts file)

<table>
<thead>
<tr>
<th>ATM_System</th>
</tr>
</thead>
<tbody>
<tr>
<td>T9</td>
</tr>
<tr>
<td>t1, T4, T9, T7, T5, T9, T7, T8</td>
</tr>
</tbody>
</table>

Table 5.9: An Example of Affecting Interaction Pattern for T9 in IP Output File (.rip file)

<table>
<thead>
<tr>
<th>ATM_System</th>
</tr>
</thead>
<tbody>
<tr>
<td>T9</td>
</tr>
<tr>
<td>ip_affecting RT9_0 Test_1 Test_2</td>
</tr>
<tr>
<td>node 0 T1</td>
</tr>
<tr>
<td>node 1 T4</td>
</tr>
<tr>
<td>inc 0 dat</td>
</tr>
<tr>
<td>node 2 T5</td>
</tr>
<tr>
<td>inc 1 ctl</td>
</tr>
<tr>
<td>inc 0 dat</td>
</tr>
<tr>
<td>node 3 T9</td>
</tr>
<tr>
<td>inc 2 dat</td>
</tr>
<tr>
<td>inc 1 ctl</td>
</tr>
<tr>
<td>inc 0 dat</td>
</tr>
</tbody>
</table>

5.5 Application of RTSR to an Example

We have applied RTSR developed in this thesis to the simplified ATM system of Figure 2.1. The requirements of the simplified ATM system are described in English in
Appendix A.1 [28]. The EFSM model for the simplified ATM system is presented in Appendix A.2.

Consider adding a balance inquiry transaction to the simplified ATM system, and deleting the deposit transaction from the simplified ATM system. The added balance inquiry transaction is represented by transition T9 and the deleted deposit transaction is represented by transition T6. The modified EFSM model of the simplified ATM system with added balance transaction and deleted deposit transaction is shown in Appendix A.3. From the modified EFSM model, the regression test suite is derived and represented in Appendix A.4.

RTSR accepts three inputs files: an EFSM input file (shown in Appendix A.6), a MOD input file (shown in Appendix A.7), and an RTS input file (shown in Appendix A.4). The RTS input file is constructed manually according to the IPO-df-Chains coverage criteria [37].

After applying RTSR, the interaction patterns for T9 and T6dummy with the equivalent test cases wrt a certain interaction pattern are shown in Table 5.10 and Table 5.11, respectively.

<table>
<thead>
<tr>
<th>Number of Interaction Patterns</th>
<th>Interaction Pattern</th>
<th>Equivalent Test Cases wrt a Certain Interaction Pattern (specified by test ids)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Affecting Interaction pattern #1</td>
<td>Test_2, Test_3, Test_5, Test_6</td>
</tr>
<tr>
<td>2</td>
<td>Affecting Interaction pattern #2</td>
<td>Test_7, Test_10, Test_13, Test_46, Test_49, Test_61, Test_64, Test_76, Test_79, Test_91, Test_92</td>
</tr>
<tr>
<td>3</td>
<td>Affecting Interaction Pattern #3</td>
<td>Test_8, Test_9, Test_11, Test_12, Test_14, Test_15</td>
</tr>
<tr>
<td>Page</td>
<td>Affecting Interaction Pattern #4</td>
<td>Affecting Interaction Pattern #5</td>
</tr>
<tr>
<td>------</td>
<td>--------------------------------</td>
<td>---------------------------------</td>
</tr>
<tr>
<td>4</td>
<td>Test_17, Test_18, Test_20, Test_21, Test_22, Test_23, Test_24, Test_25, Test_26, Test_27, Test_28, Test_29, Test_30, Test_32, Test_33, Test_35, Test_36, Test_37, Test_38, Test_39, Test_40, Test_41, Test_42, Test_43, Test_44, Test_45</td>
<td>Test_47, Test_48, Test_50, Test_51, Test_52, Test_53, Test_54, Test_55, Test_56, Test_57, Test_58, Test_59, Test_60, Test_62, Test_63, Test_65, Test_66, Test_67, Test_68, Test_69, Test_70, Test_71, Test_72, Test_73, Test_74, Test_75, Test_77, Test_78, Test_80, Test_81, Test_82, Test_83, Test_84, Test_85, Test_86, Test_87, Test_88, Test_89, Test_90</td>
</tr>
<tr>
<td>Number of Interaction Patterns</td>
<td>Interaction Pattern</td>
<td>Equivalent Test Cases wrt a Certain Interaction Pattern (specified by test ids)</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-------------------------------------</td>
<td>--------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>1</td>
<td>Affecting Interaction Pattern #1</td>
<td>Test_3, Test_12</td>
</tr>
<tr>
<td>2</td>
<td>Affecting Interaction Pattern #2</td>
<td>Test_4, Test_5, Test_7, Test_8, Test_13, Test_14, Test_31, Test_32, Test_37, Test_38, Test_46, Test_47, Test_52, Test_53, Test_76, Test_77, Test_82, Test_83, Test_92</td>
</tr>
<tr>
<td>3</td>
<td>Affecting Interaction pattern #3</td>
<td>Test_6, Test_9, Test_15</td>
</tr>
<tr>
<td>4</td>
<td>Affecting Interaction pattern #4</td>
<td>Test_18, Test_19, Test_20, Test_21, Test_24, Test_25, Test_26, Test_27, Test_28, Test_29, Test_30, Test_63, Test_64, Test_65, Test_66, Test_69, Test_70, Test_71, Test_72, Test_73, Test_74, Test_75</td>
</tr>
<tr>
<td>5</td>
<td>Affecting Interaction pattern #5</td>
<td>Test_33, Test_34, Test_35, Test_36, Test_39, Test_40, Test_41, Test_42, Test_43, Test_44, Test_45, Test_48, Test_49, Test_50, Test_51, Test_54, Test_55, Test_56, Test_57, Test_58, Test_59, Test_60, Test_78, Test_79, Test_80, Test_81, Test_84, Test_85, Test_86, Test_87, Test_88, Test_89, Test_90</td>
</tr>
<tr>
<td>6</td>
<td>Affected Interaction Pattern #1</td>
<td>Test_3, Test_12, Test_18, Test_24, Test_63, Test_69, Test_92</td>
</tr>
<tr>
<td>7</td>
<td>Affected Interaction Pattern #2</td>
<td>Test_4, Test_13</td>
</tr>
<tr>
<td>8</td>
<td>Affected Interaction Pattern #3</td>
<td>Test_5, Test_14</td>
</tr>
<tr>
<td>9</td>
<td>Affected Interaction Pattern #4</td>
<td>Test_6, Test_8, Test_9, Test_15</td>
</tr>
<tr>
<td>10</td>
<td>Affected Interaction Pattern #5</td>
<td>Test_7</td>
</tr>
<tr>
<td>11</td>
<td>Affected Interaction Pattern #6</td>
<td>Test_19, Test_31, Test_34, Test_64, Test_76, Test_79</td>
</tr>
<tr>
<td>12</td>
<td>Affected Interaction Pattern #7</td>
<td>Test_20, Test_28, Test_29, Test_32, Test_35, Test_37, Test_38, Test_43, Test_44, Test_65, Test_73, Test_74, Test_77, Test_80, Test_82, Test_83, Test_88, Test_89</td>
</tr>
<tr>
<td>13</td>
<td>Affected Interaction Pattern #8</td>
<td>Test_21, Test_25, Test_26, Test_27, Test_30, Test_33, Test_36, Test_39, Test_40, Test_41, Test_42, Test_45, Test_47, Test_48, Test_50, Test_51, Test_52, Test_53, Test_54, Test_55, Test_59, Test_60, Test_64, Test_65, Test_66, Test_69, Test_70, Test_71, Test_72, Test_73, Test_74, Test_75, Test_76, Test_77, Test_78, Test_79, Test_80, Test_81, Test_84, Test_85, Test_86, Test_87, Test_88, Test_89, Test_90</td>
</tr>
</tbody>
</table>
The results of regression test suite reduction for T9 and T6dummy are shown in Table 5.12.

Let

# RTS denote the number of regression test cases in a test suite,

# RRTS denote the number of test cases in a reduced regression test suite, and

% denote the percentage of reduction

<table>
<thead>
<tr>
<th>TUT</th>
<th>#RTS</th>
<th>#RRTS</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>T9</td>
<td>86</td>
<td>5</td>
<td>94</td>
</tr>
<tr>
<td>T6dummy</td>
<td>79</td>
<td>11</td>
<td>86</td>
</tr>
</tbody>
</table>

The results of the above example show that RTSR can be used to reduce the size of the regression test suite successfully and significantly. For example, for the simplified ATM system wrt the elementary modifications T6 and T9, a reduction of 86% to 94% is achieved.

Our example also shows that when applying regression test suite reduction based on the EFSM dependence analysis, the size of the reduced regression test suite is bounded by the number of possible interaction patterns associated with modifications.

For example, in the simplified ATM system, the possible number of affecting interaction pattern, affected interaction pattern, and side-effect interaction pattern wrt T9 are shown in Table 5.13, and the possible number of affecting interaction pattern,
affected interaction pattern, and side-effect interaction pattern wrt T6dummy are shown in Table 5.14.

<table>
<thead>
<tr>
<th>Number of Affecting Interaction Patterns</th>
<th>Number of Affected Interaction Patterns</th>
<th>Number of Side-effect Interaction Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 5.14 Possible Number of Interaction Patterns wrt T6dummy

<table>
<thead>
<tr>
<th>Number of Affecting Interaction Patterns</th>
<th>Number of Affected Interaction Patterns</th>
<th>Number of Side-effect Interaction Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>9</td>
<td>0</td>
</tr>
</tbody>
</table>

Interaction patterns are used to reduce the original test suite. For each elementary modification, three interaction patterns are computed during traversal of a test case. A test case is included in the reduced test suite if at least one of its interaction patterns does not exist for any of the tests in the reduced test suite. Therefore, the minimum size of the reduced test suite wrt T9 is equal to 5, and the minimum size of the reduced test suite wrt T6dummy is equal to 9. In the other words, the number of possible interaction patterns designates the minimum size of the reduced regression test suite regardless of the test strategy used in the regression test suite generation.

In the next chapter, we present our conclusions, with a summary of contributions and directions for future research.
Chapter 6

Conclusion

6.1 Final Remarks

Testing a software product is a very laborious and expensive process, especially for large software systems with extensive regression test suites. It has been estimated that software maintenance activities account for as much as two-thirds of the overall cost of software production. One costly but necessary maintenance activity is regression testing, which is performed on modified software to provide confidence that modifications are correct and do not have adverse effects on the unmodified portions of the software [29]. Selective regression test generation techniques are often used to generate regression test suites. The size of these regression test suites still may be very large even for relatively small systems [19].

Through this thesis, we have presented an extension to an existing approach [19] of requirement-based regression test suite reduction, and demonstrated that this approach could significantly reduce the size of regression test suites. This approach is based on the differences between the original and modified model expressed as a set of elementary modifications: elementary addition of a transition and elementary deletion of a transition. For each elementary modification, the data and control dependencies are used to capture potential interactions between EFSM transitions. The potential interactions are then used to reduce an existing regression test suite by eliminating repetitive test cases.
6.2 Summary of Contributions

In this thesis, we present an extension to an existing requirement-based regression testing approach that uses EFSM dependence analysis to reduce a given regression test suite [19]. We have defined four new dependencies called affecting data dependence, affected data dependence, affecting control dependence, and affected control dependence. We proposed algorithms for obtaining dependencies introduced by elementary modifications of the EFSM model; for generating three types of interaction patterns; for reducing a given regression test suite. These algorithms have been implemented as the Regression Test Suite Reduction tool (RTSR) which is integrated into a Test Suite Reduction and Generation tool.

We have shown that RTSR automatically reduces a given regression test suite derived by using system models in EFSM. The RTSR also reports all interaction patterns that have been covered by reduced regression test suite wrt each transition under test.

6.3 Directions for Future Research

In this thesis, we concentrate on the EFSM system models. However, this approach can be extended to other models described in formal description languages like SDL. EFSM and SDL are very popular for modeling state-based systems like computer communication, telecommunication, and industrial control systems. It would be worthwhile to adapt the presented approach of regression test suite reduction to different types of system models.

We have made an assumption that each requirement can uniquely be represented by a single transition in the given EFSM. It would be interesting to expand this approach to those EFSMs where some requirements may be represented by more than one transition.
Also, it would be interesting to determine the effectiveness of the presented approach to see whether there would be any significant loss of effectiveness when a given regression test suite is reduced.

Our approach can reduce the regression test suite significantly. However, it does not find the minimal regression test suite. It is would be valuable to investigate the construction of minimally reduced regression test suites.

In this thesis, we only study regression test suite reduction by using static dependence analysis, which disregards the repetitions of the same dependence between transitions. More sophisticated interaction patterns may be obtained by using dynamic dependence analysis [38], which is similar to the approach described in this thesis except that repetitions of the same dependence during the traversal of a test case are taken into account. It would be interesting to extend the study of regression test suite reduction from static dependence analysis to dynamic dependence analysis.
References:


17. http://www.site.uottawa.ca/~ural/TSR


Appendix A: Simplified ATM System

A.1 Requirements of the Simplified ATM System

\[ R = \{ r \mid r \text{ is a requirement which is a transition} \} \text{ or} \]

\[ R = \{ r \mid r \text{ is a requirement which is a sequence of transitions} \} \].

\[ R = \{ r_1, r_2, r_3, r_4 \} \] for the simplified ATM EFSM where:

\( r_1 \): User fails to enter a correct pin that matches with the PIN stored in the ATM card in less than four attempts. If pins don’t match, and less than four attempts were performed, the system displays an error message, increments the number of attempts and prompts for pin. At the fourth attempt, if user still fails to enter a correct pin, the system prints an error message and ejects the user’s ATM card.

\[ r_1 = T_2 \ T_2 \ T_2 \ T_3 \]

\( r_2 \): After entering a correct pin in less than four attempts, user selects withdrawal function. The system adjusts the balance, and displays a menu with withdrawal, deposit, and exit functions.

\[ r_2 = x \ T_4 \ T_5 \]

\( r_3 \): After entering a correct pin in less than four attempts, user selects deposit function. The system adjusts the balance, and displays a menu with withdrawal, deposit, and exit functions.

\[ r_3 = x \ T_4 \ T_6 \]

\( r_4 \): After entering a correct pin in less than four attempts, user selects exit function and the system ejects the user’s ATM card.

\[ r_5 = x \ T_4 \ T_8 \]

\( x: T_2 \text{ may be inserted } 0, 1, 2 \text{ or } 3 \text{ times} \)

\( T# \text{ is the transition under test (tut)} \)
A.2 The EFSM Model of the Simplified ATM System

A.3 The Modified EFSM Model of the Simplified ATM System
### A.4 Regression Test Suite for the Simplified ATM System wrt the Elementary

#### Modifications

<table>
<thead>
<tr>
<th>Test id(s)</th>
<th>Test Sequences</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T1 T4 T5 T7 T8</td>
</tr>
<tr>
<td>2</td>
<td>T1 T4 T5 T7 T9 T7 T8</td>
</tr>
<tr>
<td>3</td>
<td>T1 T4 T5 T7 T6dummy T9 T7 T8</td>
</tr>
<tr>
<td>4</td>
<td>T1 T4 T6dummy T5 T7 T8</td>
</tr>
<tr>
<td>5</td>
<td>T1 T4 T6dummy T5 T7 T9 T7 T8</td>
</tr>
<tr>
<td>6</td>
<td>T1 T4 T6dummy T5 T7 T6dummy T9 T7 T8</td>
</tr>
<tr>
<td>7</td>
<td>T1 T4 T6dummy T9 T7 T5 T7 T8</td>
</tr>
<tr>
<td>8</td>
<td>T1 T4 T6dummy T9 T7 T5 T7 T9 T7 T8</td>
</tr>
<tr>
<td>9</td>
<td>T1 T4 T6dummy T9 T7 T5 T7 T6dummy T9 T7 T8</td>
</tr>
<tr>
<td>10</td>
<td>T1 T4 T9 T7 T5 T7 T8</td>
</tr>
<tr>
<td>11</td>
<td>T1 T4 T9 T7 T5 T7 T9 T7 T8</td>
</tr>
<tr>
<td>12</td>
<td>T1 T4 T9 T7 T5 T7 T6dummy T9 T7 T8</td>
</tr>
<tr>
<td>13</td>
<td>T1 T4 T9 T7 T6dummy T5 T7 T8</td>
</tr>
<tr>
<td>14</td>
<td>T1 T4 T9 T7 T6dummy T5 T7 T9 T7 T8</td>
</tr>
<tr>
<td>15</td>
<td>T1 T4 T9 T7 T6dummy T5 T7 T6dummy T9 T7 T8</td>
</tr>
<tr>
<td>16</td>
<td>T1 T4 T5 T7 T5 T7 T8</td>
</tr>
<tr>
<td>17</td>
<td>T1 T4 T5 T7 T5 T7 T9 T7 T8</td>
</tr>
<tr>
<td>18</td>
<td>T1 T4 T5 T7 T5 T7 T6dummy T9 T7 T8</td>
</tr>
<tr>
<td>19</td>
<td>T1 T4 T5 T7 T6dummy T5 T7 T8</td>
</tr>
<tr>
<td>20</td>
<td>T1 T4 T5 T7 T6dummy T5 T7 T9 T7 T8</td>
</tr>
<tr>
<td>21</td>
<td>T1 T4 T5 T7 T6dummy T5 T7 T6dummy T9 T7 T8</td>
</tr>
<tr>
<td>22</td>
<td>T1 T4 T5 T7 T9 T7 T5 T7 T8</td>
</tr>
<tr>
<td>23</td>
<td>T1 T4 T5 T7 T9 T7 T5 T7 T9 T7 T8</td>
</tr>
<tr>
<td>24</td>
<td>T1 T4 T5 T7 T9 T7 T5 T7 T6dummy T9 T7 T8</td>
</tr>
<tr>
<td>25</td>
<td>T1 T4 T5 T7 T6dummy T9 T7 T5 T7 T8</td>
</tr>
<tr>
<td>26</td>
<td>T1 T4 T5 T7 T6dummy T9 T7 T5 T7 T9 T7 T8</td>
</tr>
<tr>
<td>27</td>
<td>T1 T4 T5 T7 T6dummy T9 T7 T5 T7 T6dummy T9 T7 T8</td>
</tr>
<tr>
<td>28</td>
<td>T1 T4 T5 T7 T9 T7 T6dummy T5 T7 T8</td>
</tr>
<tr>
<td>29</td>
<td>T1 T4 T5 T7 T9 T7 T6dummy T5 T7 T9 T7 T8</td>
</tr>
<tr>
<td>30</td>
<td>T1 T4 T5 T7 T9 T7 T6dummy T5 T7 T6dummy T9 T7 T8</td>
</tr>
<tr>
<td>31</td>
<td>T1 T4 T6dummy T5 T7 T5 T7 T8</td>
</tr>
<tr>
<td>32</td>
<td>T1 T4 T6dummy T5 T7 T5 T7 T9 T7 T8</td>
</tr>
<tr>
<td>33</td>
<td>T1 T4 T6dummy T5 T7 T5 T7 T6dummy T9 T7 T8</td>
</tr>
<tr>
<td>34</td>
<td>T1 T4 T6dummy T5 T7 T6dummy T5 T7 T8</td>
</tr>
<tr>
<td>35</td>
<td>T1 T4 T6dummy T5 T7 T6dummy T5 T7 T9 T7 T8</td>
</tr>
<tr>
<td>36</td>
<td>T1 T4 T6dummy T5 T7 T6dummy T5 T7 T6dummy T9 T7 T8</td>
</tr>
<tr>
<td>37</td>
<td>T1 T4 T6dummy T5 T7 T9 T7 T5 T7 T8</td>
</tr>
<tr>
<td>38</td>
<td>T1 T4 T6dummy T5 T7 T9 T7 T5 T7 T9 T7 T8</td>
</tr>
<tr>
<td>39</td>
<td>T1 T4 T6dummy T5 T7 T9 T7 T5 T7 T6dummy T9 T7 T8</td>
</tr>
<tr>
<td>40</td>
<td>T1 T4 T6dummy T5 T7 T6dummy T9 T7 T5 T7 T8</td>
</tr>
<tr>
<td>Transition Under Test</td>
<td>Number of regression test cases</td>
</tr>
<tr>
<td>-----------------------</td>
<td>---------------------------------</td>
</tr>
<tr>
<td>T9</td>
<td>86</td>
</tr>
<tr>
<td>T6dummy</td>
<td>79</td>
</tr>
</tbody>
</table>
A.6 "efsm" File for the EFSM of the Simplified ATM System

```
ATM_System
5 0 4

transition T1
0 1
Card(pin, b) /

Prompt_for_PIN()

attempts := constant

transition T2
1 1
PIN(p)

p, pin, attempts /* [p!=pin) and (attempts<3)) */ /

Display_error()

attempts := function(attempts)

Prompt_for_PIN()

transition T3
1 4
```
PIN(p)
p, pin, attempts /* [(p!=pin) and (attempts==3)] */ / Display_error()
Eject_card()

transition T4
1 2
PIN(p)
p, pin /* [p==pin] */ / Display_menu()

transition T5
2 3
Withdrawal(w) /
b := function(b, w)

transition T6
2 3
Deposit(d) /
b := function(b, d)

transition T7
3 2
Continue() /
print(b)
Display_menu()

transition T8
2 4
Exit() /
Eject_card()

A.7 “.mod” File for the EFSM of the Simplified ATM System

mtransition 0 T9
2 3
Balance() /
Display(b)

mtransition 1 T6
2 3
Deposit(d) /
b := function(b, d)

A.8 An Example “.ts” File for the EFSM of the Simplified ATM System

ATM_System
T9 T6dummy

test Test_1 T1 T4 T9 T7 T5 T7 T9 T7 T8

test Test_2 T1 T2 T4 T9 T7 T5 T7 T9 T7 T8

test Test_3 T1 T4 T6dummy T5 T7 T8

test Test_4 T1 T2 T4 T6dummy T5 T7 T8
A.9 An Example “.rip” File for the EFSM of the Simplified ATM System

ATM_System

T9

ip_affecting RT9_0 Test_1 Test_2
node 0 T1
node 1 T4
inc 0 dat
node 2 T5
inc 1 ctl
inc 0 dat
node 3 T9
inc 2 dat
inc 1 ctl
inc 0 dat

ip_affect ed RT9_1 Test_1 Test_2
node 0 T7
node 1 T9
out 0 ctl

ip_sideEffect RT9_2 Test_1 Test_2
node 0 T1
out 1 activation
node 1 T7
node 2 T9

ATM_System
T6dummy

ip_affecting RT6dummy_0 Test_3 Test_4
node 0 T1
node 1 T6
inc 0 affectingGhostDat

ip_affected RT6dummy_1 Test_3 Test_4
node 0 T5
out 2 dat
out 2 ctl
node 1 T6
out 0 affectedGhostDat
node 2 T7