MODEL-BASED REGRESSION TEST SELECTION AND OPTIMIZATION FOR EMBEDDED PROGRAMS

Swarnendu Biswas

MODEL-BASED REGRESSION TEST SELECTION AND OPTIMIZATION FOR EMBEDDED PROGRAMS

Thesis submitted to the
Indian Institute of Technology, Kharagpur
For the award of the degree

of

Master of Science (M.S.)

by

Swarnendu Biswas

under the guidance of

Prof. Rajib Mall

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

JUNE 2011

To my dear parents

APPROVAL OF THE VIVA-VOCE BOARD

___/___/_____

Certified that the thesis entitled "**Model-Based Regression Test Selection and Optimization for Embedded Programs**" submitted by **Swarnendu Biswas** to the Indian Institute of Technology, Kharagpur, for the award of the degree Master of Science (M.S.) has been accepted by the external examiners and that the student has successfully defended the thesis in the viva-voce examination held today.

_____	_____
Prof. Rajib Mall	**Prof. Anupam Basu**
(Supervisor)	(Member of DAC)
_____	_____
Prof. Pallab Dasgupta	**Prof. Arobindo Gupta**
(Member of DAC)	(Member of DAC)
_____	_____
.....	**Prof. Jayanta Mukhopadhyay**
External Examiner	Chairman

CERTIFICATE

This is to certify that the thesis entitled "**Model-Based Regression Test Selection and Optimization for Embedded Programs**", submitted by **Swarnendu Biswas** to the Indian Institute of Technology, Kharagpur, is a record of bona fide research work under my supervision and I consider it worthy of consideration for the award of the degree of Master of Science (M.S.) of the Institute.

_____

Prof. Rajib Mall
Supervisor

Date:

DECLARATION

I declare that

a. The work contained in the thesis is original and has been done by myself under the general supervision of my supervisor(s).

b. The work has not been submitted to any other Institute for any degree or diploma.

c. I have followed the guidelines provided by the Institute in writing the thesis.

d. I have conformed to the norms and guidelines given in the Ethical Code of Conduct of the Institute.

e. Whenever I have used materials (data, theoretical analysis, and text) from other sources, I have given due credit to them by citing them in the text of the thesis and giving their details in the references.

f. Whenever I have quoted written materials from other sources, I have put them under quotation marks and given due credit to the sources by citing them and giving required details in the references.

_____

Swarnendu Biswas

Acknowledgements

My stay at IIT Kharagpur has been a long and eventful journey with its own gamut of emotions, sometimes thrilling, and full of despair and exasperating at certain other times. However, there has always been a few well-wishers who strived to make my MS. journey enjoyable and a success. I would be bordering upon blasphemy if I do not acknowledge their contributions.

I would like to take this opportunity to express my gratitude and indebtedness to my supervisor, Prof. Rajib Mall, for his continued support in all aspects of my research work. His guidance and motivation has been invaluable to me during the ups and downs of my work. I really admire and hope to emulate the zeal, dedication, and sincerity that Prof. Mall brings to his work. I consider myself very fortunate to have had Prof. Rajib Mall as my MS. supervisor.

I am also grateful to all the other faculty members of the Department of Computer Science and Engineering, IIT Kharagpur. In particular, I would like to thank Prof. P. Dasgupta, Prof. A. Gupta and Prof. A. Basu for their constructive feedback and insightful comments on my work. I would like to thank Prof. P. Dasgupta for helping me during my MS. admission at IIT Kharagpur. I would like to take this opportunity to thank the software laboratory staff Mr. Prasun Bhattacharjee and Mr. Shibabroto Banerjee for their co-operation and support in providing suitable infrastructure. I also appreciate the contribution of Mayank Mittal and Vipin Kumar K. S. (MTech. Batch of 2009, CSE) during the initial stages of this research work.

I would like to thank all the friends and acquaintances that I have made during my time at IIT Kharagpur. My MS. journey of three years has been made all the more memorable and rich only because of their constant support and companionship in both technical and non-technical areas. I would like to thank Sirsendu, Shibendu, Sourya, Soumyadip, Surajit, Rajarshi, ESF Najumudheen, Sujan, Dhole, Kunal and many others who have helped me in different ways during my stay at IIT Kharagpur. Those aimless and unending discussions that we used to have at our lab, our VSRC mess, etc. were really awesome, and were the best stress relievers

available in IIT Kharagpur. I will really cherish all those memories that I will be carrying with me for the rest of my life.

Finally, this page would never be complete without acknowledging the contributions of my parents. My work would not have reached its final outcome without the unconditional love, care and support of my parents.

Overall, my life and my MS. journey at IIT Kharagpur have been full of learning and unforgettable experiences. A very big thank you to one and all!

Best regards,
Swarnendu Biswas,
Department of CSE,
IIT Kharagpur.

Date: 16th May, 2011

Abstract

A safe regression test selection technique for embedded programs needs to take into account additional execution dependencies that arise among program elements due to features such as tasks, task deadlines, task precedences, and inter-task communication besides the traditionally considered data and control dependencies. In this context, we propose a model-based regression test selection technique for embedded C programs. Our technique first constructs a graph model for the embedded program to be regression tested. Our proposed graph model can capture the identified features which are important for regression test selection of embedded programs. Our regression test selection technique selects regression test cases based on an analysis of the constructed model. In addition to data and control dependencies, our technique also takes into account the task execution dependencies and the special semantics of the different features in an embedded program. We have implemented our regression test selection technique to realize a prototype tool. Our experimental studies show that though using our approach on the average about 28.33% more regression test cases were selected as compared to existing approaches, it did select all the fault-revealing test cases whereas 36.36% fault-revealing test cases were overlooked by the existing approaches.

However, the selected regression test suite may still be prohibitively large to be executed within the available time or resources. In this context, we propose a model-based multi-objective regression test suite optimization technique for embedded programs. Our technique targets to ensure that the thoroughness of regression testing of an embedded application using the optimized test suite is not compromised. This is achieved by defining optimization constraints such that the test cases that execute affected tasks and the critical functionalities are not omitted. In addition to these constraints, we aim to minimize the cost of regression testing, maximize the observable reliability, and remove redundant test cases during optimization. Experimental studies carried out by us show that the test suites optimized by our method include all the fault-revealing test cases from the initial regression test suite and at the same time achieve substantial savings in regression testing effort.

Keywords: *Regression testing, embedded systems, slicing, test case selection, test suite optimization.*

List of Abbreviations and Symbols

Abbreviations

CDG	-	Control Dependence Graph
CFG	-	Control Flow Graph
COTS	-	Commercial Off-The-Shelf
DDG	-	Data Dependence Graph
DFA	-	Deterministic Finite Automaton
DPC	-	Deferred Procedure Call
FLIH	-	First-level Interrupt Handler
GA	-	Genetic Algorithms
ISR	-	Interrupt Service Routine
MTest	-	Model-based Test Case Selector
PDG	-	Program Dependence Graph
PUT	-	Program Under Test
RTS	-	Regression Test Selection
RTSEM	-	Regression Test Selection for Embedded Programs
RTSO	-	Regression Test Suite Optimization
SDG	-	System Dependence Graph
SDGC	-	System Dependence Graph with Control Flow
TCP	-	Test Case Prioritization
TSO	-	Test Suite Optimization
TSM	-	Test Suite Minimization
WCET	-	Worst Case Execution Time

Symbols and Notations

$ET(P(t))$	-	Execution trace of a test case t on a program P
ET_{SDG}	-	All edge types of an SDG
ET_{SDGC}	-	All edges types of an SDGC
P	-	Original Program
P′	-	Modified Program
t	-	A test case in T
T	-	Initial Test Suite
VT_{SDG}	-	All node types of an SDG
VT_{SDGC}	-	All node types of an SDGC
τ	-	A task in an embedded program

Contents

Chapter 1

Introduction

Over the last decade or so, there has been a rapid surge in the usage of embedded applications. A large variety of embedded applications now touch our daily lives. These include home appliances, communication systems, entertainment systems, and automobiles just to name a few. In addition to the rapid increase in the popularity of embedded systems, an unmistakable trend is their increasing size and sophistication [107]. The enhanced capabilities of embedded systems coupled with the demands for flexibility have contributed to prolific usage of these systems even in safety-critical areas such as nuclear power stations, health-care, avionics, etc [103]. Embedded systems used in safety-critical applications are required to have far greater reliability than conventional applications. Seo *et al.* have reported [107] that though embedded programs on an average implement less than 20% of the functionalities of an embedded system, yet more than 80% of the reported failures could be attributed to software bugs. In this context, effective *regression* testing of evolving embedded software assumes increased significance.

Maintenance of an embedded program is frequently necessitated to fix bugs, to enhance or adapt existing functionalities, or to port it to different environments. After the necessary changes have been made, *resolution* testing is carried out to check whether the required changes have been carried out properly. Regression testing is carried out to ensure that no new errors have been introduced due to the changes made [63]. The model of a popular maintenance process carried out after a software is released is shown in Figure 1.1. As shown in Figure 1.1, after a software is released, the failure reports and change requests of the software are compiled and the software is modified to make the necessary changes. Resolution tests are carried out to verify the directly modified parts of the code, while regression testing is carried out to test the unchanged parts of the code that may be affected by the

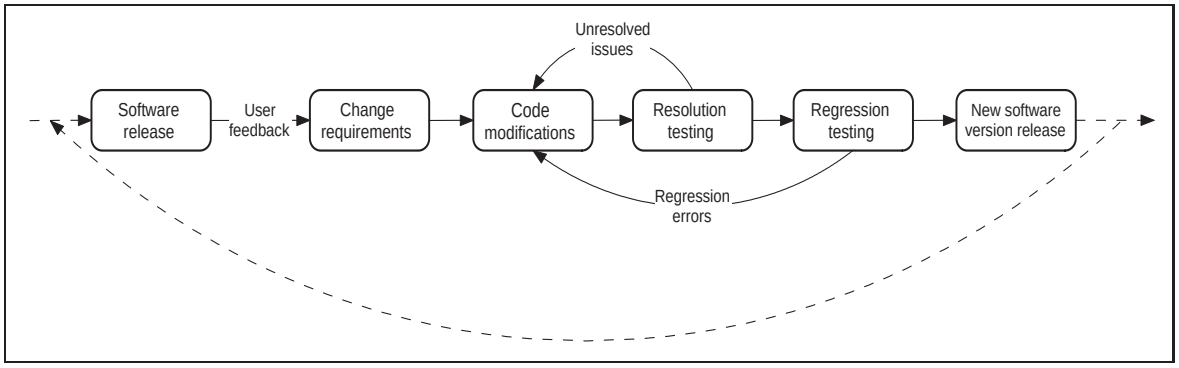


Fig. 1.1: Maintenance process model.

code change. After testing is complete, a new version of the software is released, which then undergoes a similar cycle.

Regression testing is carried out during different phases of software development: at unit, integration, and system testing, as well as during the maintenance phase [63]. Regression testing of an evolving application is a crucial activity, and consumes significant amount of time and effort. The extent of time and effort that are being spent on regression testing is exemplified by a study [30] that reports that it took twenty-seven days on an average to execute 135 regression test cases for an embedded real-time application of size of about 300 KLOC. In fact, regression testing has been estimated to account for almost half of the total software maintenance costs [55, 63]. To reduce regression testing costs, it is necessary to eliminate all those test cases that do not have any chance of detecting a bug, and at the same time ensuring that no test case that has a potential to detect a regression bug is overlooked. Accurate regression test selection is, therefore, considered to be an issue of considerable practical importance, and has the potential to substantially reduce software maintenance costs [41].

1.1. Motivation for Our Work

Regression test selection (RTS) techniques select a subset of valid test cases from an initial test suite (T) to test the affected but unmodified parts of a program [63, 94]. Regression test selection essentially consists of two major activities:

- Identification of the affected parts - This involves identification of the unmodified parts of the program that are affected by the modifications.
- Test case selection - This involves identification of a subset of test cases from

the initial test suite T which can *effectively* test the unmodified parts of the program. The aim is to be able to select the subset of test cases from the initial test suite that has the potential to detect errors induced on account of the changes.

Rothermel and Harrold [92] have formally defined the regression test selection problem as follows: *Let P be an application program and P′ be a modified version of P. Let T be the test suite developed initially for testing P. An RTS technique aims to select a subset of test cases $T' \subseteq T$ to be executed on P′, such that every error detected when P′ is executed with T is also detected when P′ is executed with T′.*

A large number of RTS techniques for procedural, object-oriented, component-based, aspect-oriented, and web-based applications have been reported in the literature [15, 19, 44, 82, 94, 138]. However, research results on RTS for embedded programs have scarcely been reported in the literature. Possibly this is one of the reasons why in industry regression test cases for embedded programs are selected based either on expert judgment, or on some form of manual program analysis [24, 41]. However, the effectiveness of such approaches tends to decrease rapidly as the complexity of software increases [24]. Also manual test selection often leads to a large number of test cases to be selected and rerun even for minor program changes, leading to unnecessarily high regression testing costs. What is probably more disconcerting is the fact that many test cases which could have potentially detected regression errors could get overlooked during manual selection.

As compared to traditional applications, regression testing of embedded programs poses several additional challenges [80,104,111]. In the following, we briefly highlight the main complicacies that surface while selecting regression test cases for embedded applications. Embedded applications usually consist of concurrent and co-operating tasks having real-time constraints. Apart from verifying the functional correctness of an embedded program, satisfaction of timing properties of the tasks also needs to be tested. A cursory analysis of this situation reveals that analysis of only data or control dependencies would not be satisfactory for selection of regression test suites for embedded programs. Unless timing issues are carefully analyzed and taken into consideration, several potentially *fault-revealing* test cases may be omitted during RTS for embedded programs.

It can be easily argued that existing RTS techniques [19, 94, 119] reported for procedural programs are unsafe for embedded programs. Traditional RTS techniques ignore the implications of important features of embedded programs, such

as task concurrency and time-constraints on tasks, on regression test selection. In an embedded application, it is possible that the execution of a task may get delayed due to changes made to the code of other tasks. For example, when two tasks are communicating using a shared variable, the access to the shared variable is usually guarded with a semaphore. If a task blocks the semaphore for a longer duration due to a change made to the task code, then the execution of the other tasks using that semaphore may get delayed. An unmodified task can also get delayed due to a modification made to the code of some other task due to issues such as message passing, precedence ordering and priorities. Whenever the code of one task is changed, it becomes necessary to test those tasks whose execution time can potentially get affected due to implicit *task execution* dependencies. Therefore, in addition to traditional data and control dependency-based analysis, an RTS technique for embedded programs needs to take into account the execution dependencies existing among the various tasks.

In this context, it is important to note the difference between task timing analysis and execution dependency analysis. While timing analysis deals with prediction of worst case execution time (WCET) for tasks, the aim of task execution dependency analysis is to identify all those tasks in an application which can affect the timing behavior of a given task.

We present an example of a regression error caused due to task execution dependency. Consider an embedded program composed of three tasks τ_1, τ_2 and τ_3 as shown in Figure 1.2a. The task τ_2 has a deadline but can start only after task τ_1 completes, while there is no such constraint on task τ_3. Suppose the task τ_1 is changed as shown in Figure 1.2b for fixing a bug. A change to task τ_1 can increase its execution time, and this would subsequently cause τ_2 to get delayed and miss its deadline. Existing RTS techniques select test cases based on only data and control dependencies, and therefore, are likely to omit test cases that could detect check the temporal failures of task τ_2.

Programmers extensively use exception handling mechanisms to increase the reliability and robustness of embedded software. Examples of commonly raised exceptions in embedded systems are expiration of timers, `NULL` pointer exception, etc. According to a study, the code implementing exception handling mechanisms is usually tested much less compared to the other parts of the code, and is, therefore, much more likely to have bugs, and according to one study accounts for as much as two-thirds of the system crashes [75]. When an exception arises, it results in

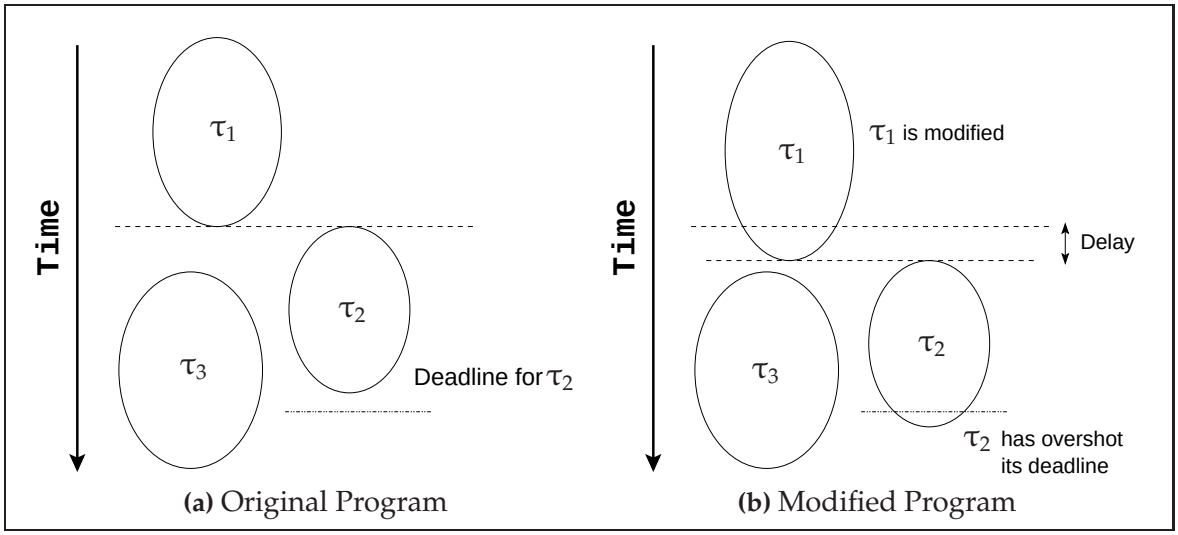


Fig. 1.2: An example to show the need of task execution dependency analysis in regression test selection for embedded programs.

transfer of control from the point where it is raised (within the *exception* block) to the corresponding exception handler routine. Exceptions raised in a program can also change the dependence relationships for some variables [54]. In such cases, the data dependencies may also be affected as the exception handling mechanism may alter the definition-use sequences for some variables. Therefore, satisfactory regression test selection of embedded programs requires explicit analysis of flow of control due to exception handling [110].

Regression testing of embedded programs is further constrained due to restrictions on resources such as time, budget, personnel, etc [34, 135]. Often, paucity of time appears as the primary obstacle to testers during regression testing [121]. Furthermore, regression testing of embedded programs is usually extremely expensive because the test cases are run on specific hardware and require setting up specific execution/simulation environments. As a result, the set of selected regression test cases may be prohibitively large to be executed with the program under test (PUT) and still meet the given constraints on regression testing. Therefore, in the industry, testers often have to manually *optimize* the selected test cases from experience to meet the testing goals. Experienced testers manually optimize the regression test suite by using their knowledge of the requirement specification document (SRS), the changes that have been incorporated, and the previous testing history. However, this process becomes difficult and error-prone for non-trivial programs and can severely compromise the thoroughness of regression testing [34, 133].

1.2. Objectives and Scope of Our Work

The primary goal of our work is to investigate the possibility to realize improved RTS and regression test suite optimization (RTSO) techniques for embedded C programs that takes into consideration the special features of an embedded program. We have chosen C as the programming language since the inherent flexibility and the ease of porting across a wide range of hardware platforms has made the C programming language [87] a popular choice for developing real-time and safety-critical embedded applications [7]. Towards this goal, we have set the following objectives:

- We plan to design a suitable graph model for representing all the features of an embedded program that are important in the context of regression test selection and optimization.
- We plan to develop a model-based RTS technique for embedded programs. We also plan to implement a prototype tool to evaluate the effectiveness of our proposed RTS technique as compared to existing approaches.
- We plan to develop a multi-objective RTSO technique for embedded programs. We plan to develop a prototype tool to verify the efficacy of our approach.

1.3. Contributions of This Thesis

In light of the discussed inadequacies of the existing approaches, we have proposed improved regression test selection and optimization techniques for embedded programs. We first propose a graph representation for modeling those features of an embedded program that are relevant to regression testing. Our proposed model, in addition to capturing data and control dependencies, also represents control flow information and embedded program features such as tasks, task precedences and inter-task communication using message queues and semaphores. We have proposed an RTS technique based on an analysis of the constructed models. We have implemented a prototype tool to validate the effectiveness of our proposed RTS technique. We have also proposed a multi-objective regression test suite optimization technique for embedded programs. We report an implementation of a prototype tool for optimizing the set of selected regression test cases, and present an

experimental evaluation of our approach. From our experimental studies, we observe that our regression test case selection and optimization techniques include all potentially fault-revealing test cases and at the same time achieve savings in terms of regression test effort without compromising on the thoroughness of testing.

We can elaborate the above mentioned general contributions into the following specific contributions of this thesis:

1. ***A graph model for embedded programs*** - The models proposed in the literature for use in regression test selection ignore many features that are important for embedded programs such as tasks, task precedence orders, time outs, inter-task communication using message queues and semaphores, interrupts and exception handling. Our proposed model has been designed to capture these important features of embedded programs, and is, therefore, an original contribution.
2. ***Model-based regression test selection technique for embedded programs*** - Existing regression test selection techniques for procedural programs are usually based on analysis of data and control dependencies and control flow information. However, modifications to a task in an embedded program can affect the completion times of other tasks. We, therefore, select regression test cases by analyzing the execution dependencies that exist among tasks in addition to control and data dependency analysis. Our technique determines execution dependencies among tasks that can arise due to various issues such as task precedence orders, task priorities, inter-task communication using message queues and semaphores, exception handling, and execution of interrupt handlers.
3. ***Model-based multi-objective regression test suite optimization technique for embedded programs*** - The thoroughness of regression testing an embedded program may be compromised if existing optimization techniques are used. This is because the existing techniques may ignore test cases that execute critical functionalities and the affected time-constrained tasks. We have proposed a program model-based multi-objective regression test suite optimization technique for embedded programs that aim to minimize the cost of regression testing, maximize the reliability, and remove redundant test cases. Our proposed optimization technique also ensures that test cases that execute affected tasks and critical functionalities of an embedded program are not excluded so that the thoroughness of regression testing with the optimized test

suite is not compromised.

1.4. Organization of the Thesis

The rest of this thesis is organized as follows:

Chapter 2 discusses the basic concepts that have been used in the subsequent chapters of the thesis. More specifically, we discuss several concepts that are being used extensively in the context of regression testing. We discuss some popular intermediate program representations which are used for RTS, and discuss some features of embedded programs that are relevant to regression test selection and optimization. We also provide a brief overview of genetic algorithms.

Chapter 3 reviews the related work in the field of regression test selection and optimization.

Chapter 4 presents the different types of execution dependencies that can arise among the tasks in an embedded program, and discusses how these can systematically be analyzed.

Chapter 5 introduces our proposed graph model for embedded programs. Our discussion also includes a model construction technique and an analysis of the construction complexity.

Chapter 6 presents our model-based RTS technique for embedded programs. In this chapter, we first discuss our technique to select regression test cases based on data, control and task execution dependencies that exist among the code elements of an embedded program. Subsequently, we discuss a prototype implementation of our RTS technique and the experimental results obtained using the prototype implementation.

Chapter 7 presents our proposed SDGC model-based multi-objective RTSO technique for embedded programs. In this chapter, we first discuss the optimization objectives and constraints implemented in our optimization technique. Subsequently, we discuss a prototype implementation of our RTSO technique, and the experimental results obtained using our prototype tool.

Chapter 8 concludes the thesis by highlighting the important contributions made, and possible extensions to the work.

Chapter 2

Basic Concepts

Over the last two decades or so, a significant amount of research results related to regression testing have been published in the literature. These results have laid the foundation of systematic regression test selection and have contributed significant knowledge and a plethora of terminologies.

In this chapter, we first discuss a few important concepts that are extensively being used in the context of regression testing. Subsequently, we discuss some popular intermediate program representations which have been used by researchers for RTS. We also discuss some features of embedded programs that are relevant to regression test selection and optimization, and then briefly discuss about genetic algorithms.

For notational convenience throughout the rest of the thesis unless otherwise specified, the original and the modified programs shall be denoted by P and P' respectively. The initial regression test suite will be denoted by T, and t will denote any test case in T.

This chapter has been organized as follows: We discuss concepts related to regression testing in Section 2.1. In Section 2.2, we discuss few popular program models that have been proposed for procedural programs. We discuss relevant embedded program features that are important in the context of regression testing in Section 2.3. We discuss few important concepts in genetic algorithms in Section 2.4. We conclude the chapter in Section 2.5.

2.1. Regression Testing Concepts

In this section, we discuss a few important notations and concepts relevant to regression testing.

2.1.1. Obsolete, Retestable and Redundant Test Cases

According to Leung and White [63], test cases in the initial test suite can be classified as obsolete, regression and redundant (or reusable) test cases. Obsolete test cases are no more valid for the modified version of the program. Regression test cases are those test cases which execute the affected parts of the system and need to be rerun during regression testing. Redundant test cases execute the unaffected parts of the system. Hence although they are valid test cases (i.e., not obsolete), they can still be omitted from the regression test suite.

Figure 2.1 shows the classes into which test cases from the initial test suite can be partitioned after a modification has been made to a program [63, 74]. In this context, it should be noted that only the regression test cases shown in Figure 2.1 should be used to revalidate a modified program.

2.1.2. Execution Trace of a Test Case

The execution trace of a test case t on a program P (denoted by $ET(P(t))$) is defined as the sequence of statements in P that are executed when P is executed with t [94]. The execution trace information for P can be generated by appropriately instrumenting the source code.

2.1.3. Fault-revealing Tests

A test case $t \in T$ is said to be *fault-revealing* for a program P, if and only if it can cause P to fail by producing incorrect outputs for P [93].

2.1.4. Modification-revealing Tests

A test case $t \in T$ is said to be *modification-revealing* for P' if and only if it produces different outputs for P and P' [93].

2.1.5. Modification-traversing Tests

A test case $t \in T$ is *modification-traversing* for P and P', if and only if the execution traces of t on P and P' are different [93, 94]. In other words, a test case t is said to be modification-traversing if it executes the modified regions of code in P'. For a

given original program and its modified version, the set of modification-traversing test cases is a superset of the set of the modification-revealing test cases.

Figure 2.2 [93] depicts the inclusion relationship among the different classes of test cases discussed.

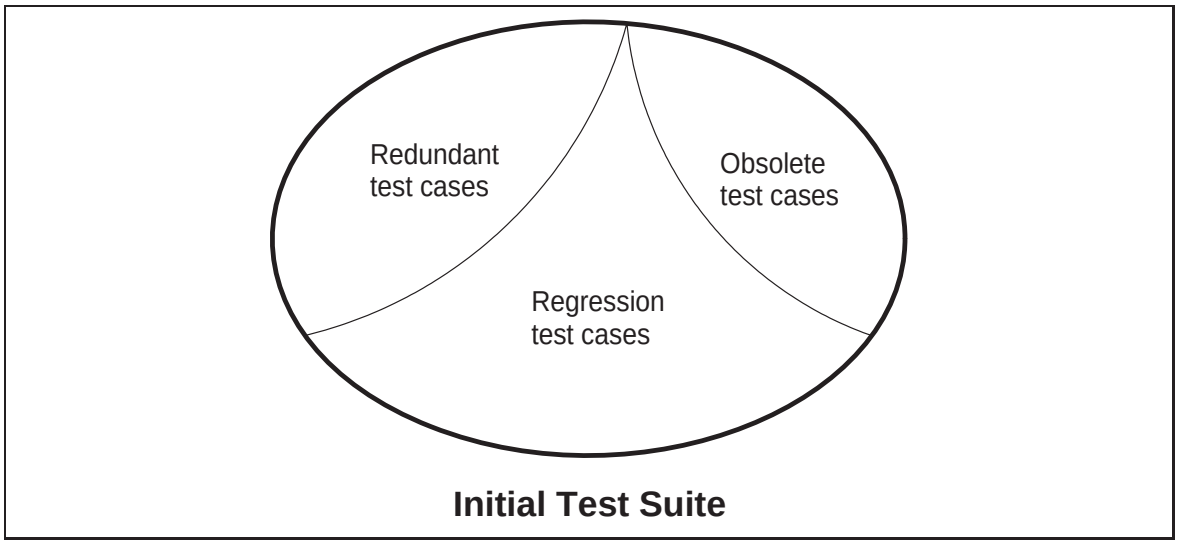


Fig. 2.1: A partition of the initial test suite.

2.1.6. Inclusive, Precise and Safe Regression Tests

Inclusiveness measures the extent to which an RTS technique selects modification-revealing tests from an initial regression test suite T [93]. Let us consider an initial test suite T containing n modification-revealing test cases. If an RTS technique M selects m of these test case, the inclusiveness of the RTS technique M is expressed as $(m/n) * 100$ [93].

A *safe* RTS technique selects all those test cases from the initial test suite that are modification-revealing [93]. Therefore, an RTS technique is said to be safe if and only if it is 100% inclusive. Regression test cases that are relevant to a change but are not selected by an RTS technique are called *false negatives*. Therefore, a safe RTS technique does not select false negatives [26].

Precision measures the extent to which an RTS algorithm ignores test cases that are non-modification-revealing [93]. Test cases that are selected by a technique but are not relevant to the changes made are called *false positives*. An RTS technique is, therefore, precise if and only if there are no instances of false positives in the regression test cases selected by it [26].

2.1.7. Program Slicing

Program slicing is a program analysis technique. It was introduced by Weiser [124] to aid in debugging programs. A program slice is usually defined with respect to a slicing criterion. A slicing criterion SC is a pair $< p, V >$, where p is a program point of interest and V is a subset of the program's variables. A slice of a program P with respect to a slicing criterion SC is the set of all the statements of the program P that might affect the slicing criterion for arbitrary inputs to the program.

Since the publication of Weiser's seminal work, the concept of slicing has been extended in many ways and have been applied to other areas such as program understanding, compiler optimization, reverse engineering, etc. Comprehensive surveys on program slicing can be found in [116, 130].

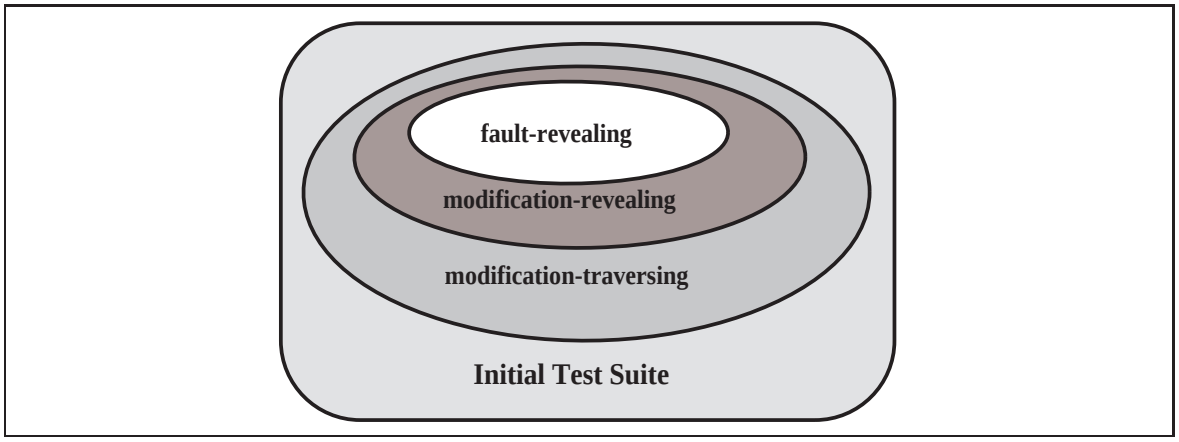


Fig. 2.2: Inclusion relationship between various classes of test suites.

2.1.8. Regression Test Suite Minimization and Prioritization

Regression test suite minimization (TSM) [43, 68, 70] and regression test case prioritization [30, 97, 121] are two other approaches proposed in the literature to reduce the effort involved in regression testing. TSM techniques aim to reduce the size of the regression test suite by eliminating redundant test cases while ensuring that the coverage achieved by the minimized test suite is identical to the initial test suite. Different studies published in the literature [68, 96, 128] report conflicting results on the impact of TSM techniques on the fault detection capabilities of test suites. Lin et al. observed [68] that the TSM problem is *NP-complete*, since the minimum set-covering problem [28] can be reduced to the TSM problem in polynomial time.

Test case prioritization (TCP) techniques order test cases on some considerations

such as test cases that have a higher fault detection capability are assigned a higher priority and can gainfully be taken up for execution earlier. TCP approaches usually aim to improve the rate of fault detection by the ordered test suite [30, 97]. The main advantage of ordering test cases is that bugs are detected and can be reported to the development team early so that they can get started with fixing the bugs [97]. Also TCP techniques provide testers with the choice of executing only a certain number of higher priority test cases to meet the given time or cost considerations. This is advantageous especially in the case of unpredicted interruptions to testing activities on account of delivery, resource or budget constraints.

Many different TSM and TCP approaches have been proposed in recent years, and they are active areas of research by themselves. However, our current work focuses on RTS and RTSO techniques in the context of embedded programs. More detailed information about TSM and TCP approaches can be found in [29, 30, 134].

2.2. Procedural Program Models

Graph models of programs have extensively been used in many applications such as program slicing [66, 109], impact analysis [60], reverse engineering [27], computation of program metrics [122], etc. Some of the popular graph models reported in the literature include Control Flow Graphs (CFG) [10, 94], Program Dependence Graphs (PDG) [35], and System Dependence Graphs (SDG) [49]. In the following, we briefly review some of these graph models that are relevant to our work.

2.2.1. Control Flow Graph

A control flow graph (CFG) [10, 94] is a directed graph that represents the sequence in which the different statements in a program get executed. The CFG of a program P is the flow graph $G = (N \cup Start \cup Stop, E)$ where N is the set of nodes and E is the set of edges. Two additional nodes *Start* and *Stop* are used to represent the entry and the exit points of a CFG. Each node $n \in N$ represents an assignment statement or a predicate in P. An edge $(m, n) \in E$ indicates a possible flow of control from the node m to the node n. Edges in a CFG are of two types, `TRUE` and `FALSE`, representing the possible flow of control when a predicate evaluates to `TRUE` or `FALSE` respectively. A CFG, therefore, captures all possible flows of control in a program.

Figure 2.4 represents the CFG of the program in Figure 2.3. Note that the existence of an edge (x, y) in a CFG does not necessarily mean that control *must* transfer from x to y during a program run.

2.2.2. Data Dependence Graph

Dependence graphs are used to represent potential dependencies between the elements of the program. In the following, we discuss data and control dependencies between program elements and their graph representations.

```
     int a, b, sum;
1.   read( a );
2.   read( b );
3.   sum = 0;
4.   while ( a < 8 ) {
5.       sum = sum + b;
6.       a = a + 1; }
7.   write( sum );
8.   sum = b;
9.   write( sum );
```

Fig. 2.3: A sample program.

Data Dependence: Let G be the CFG of a program P. A node $n \in G$ is said to be data dependent on a node $m \in G$, if there exists a variable *var* of the program P such that the following hold:

1. The node m defines *var*,
2. The node n uses *var*,
3. There exists a directed path from m to n along which there is no intervening definition of *var*.

Consider the sample program shown in Figure 2.3 and its CFG shown in Figure 2.4. From the use of the variables *sum* and *b* in line 5, it is evident that node 5 is data dependent on nodes 2, 3 and 5. Similarly, node 8 is data dependent on only node 2. However, node 8 is not data dependent on either nodes 3 and 5.

Data Dependence Graph: The data dependence graph (DDG) of a program P is the graph $G_{DDG} = (N, E)$, where each node $n \in N$ represents a statement in the program P and if x and y are two nodes of G, then $(x, y) \in E$ if and only if x is data dependent on y.

2.2.3. Control Dependence Graph

The execution of certain statements in a program is dependent on the result of the evaluation of the associated conditional statements and loop conditionals. The concept of control dependence [10] was designed to capture these relations in a program.

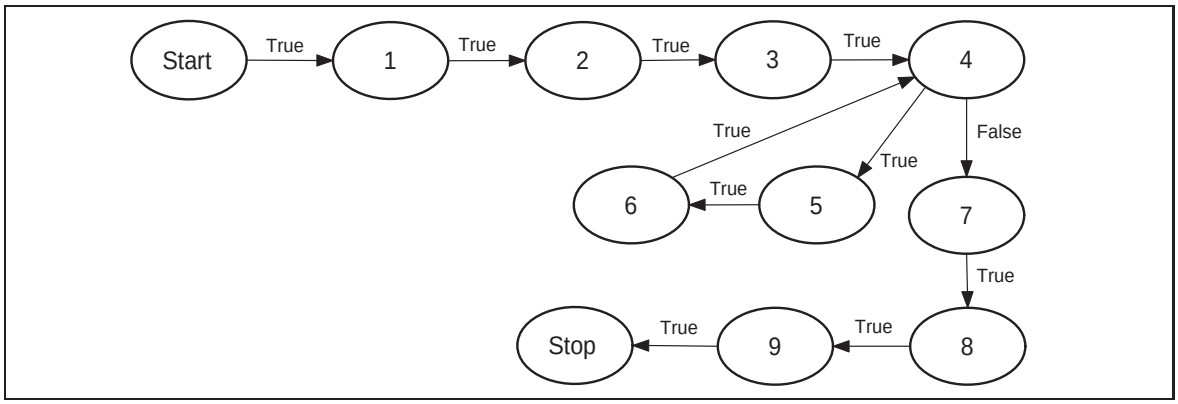


Fig. 2.4: CFG for the example program shown in Figure 2.3.

Dominance: If x and y are two nodes in a flow graph, then x dominates y if and only if every path from *Start* to y passes through x. Similarly, y post-dominates x if and only if every path from x to *Stop* passes through y.

Let x and y be two nodes in a flow graph G. Node x is said to be the immediate post-dominator of node y if and only if x is a post-dominator of y, $x \neq y$ and every other post-dominator $z \neq x$ of y post-dominates x. The post-dominator tree of a flow graph G is the tree that consists of the nodes of G, has *Stop* as the root node, and has an edge (x, y) if and only if x is the immediate post-dominator of y.

Control Dependence: Let G be the CFG of a program P, and let x and y be two arbitrary nodes in G. The node y is said to be control dependent on another node x if the following hold:

1. There exists a directed path Q from x to y,
2. y post-dominates every z in Q (excluding x and y),
3. y does not post-dominate x.

The concept of control dependence implies that if y is control dependent on x, then x must have multiple successors in G. Conversely, if x has multiple successors, then at least one of its successors must be control dependent on it. Consider the

program of Figure 2.3 and its CFG in Figure 2.4. Each of the nodes 5 and 6 is control dependent on node 4. Note that although node 4 has two successor nodes 5 and 7, only node 5 is control dependent on node 4.

The control dependence graph (CDG) of a program P is the graph $G_{CDG} = (N, E)$, where each node $n \in N$ represents a statement of the program P, and $(x, y) \in E$ if and only if x is control dependent on y.

2.2.4. Program Dependence Graph

The program dependence graph (PDG) [35] for a program P explicitly represents both control and data dependencies. This graph contains two kinds of directed edges: control and data dependence edges. A control (or data) dependence edge (m, n) indicates that the node m is control (or data) dependent on the node n. Therefore the PDG of a program P is the union of a pair of graphs: the data dependence graph of P and the control dependence graph of P. The *PDG* for the program in Figure 2.3 is shown in Figure 2.5.

2.2.5. System Dependence Graph

A major limitation of the PDG representation is that it can model a single procedure only and cannot model inter-procedural calls. Horwitz et al. [49] enhanced the PDG representation to handle procedure calls and introduced the system dependence graph (SDG) representation to model a main program together with all its non-nested procedures.

Let $\mathbf{VT_{SDG}}$ be the set of all types of nodes of an SDG. Then, $\mathbf{VT_{SDG}}$ can be expressed as follows:

$$\mathbf{VT_{SDG}} = \{V_{assign}, V_{pred}, V_{call}, V_{A-in}, V_{A-out}, V_{F-in}, V_{F-out}\},$$

where each member of the set $\mathbf{VT_{SDG}}$ represents a particular node type. In the following, we explain the different types of nodes in an SDG:

- Node types V_{assign} and V_{pred} represent assignment statements and control predicates respectively.
- *Call-site* nodes (V_{call}) represent the procedure call statements in a program.

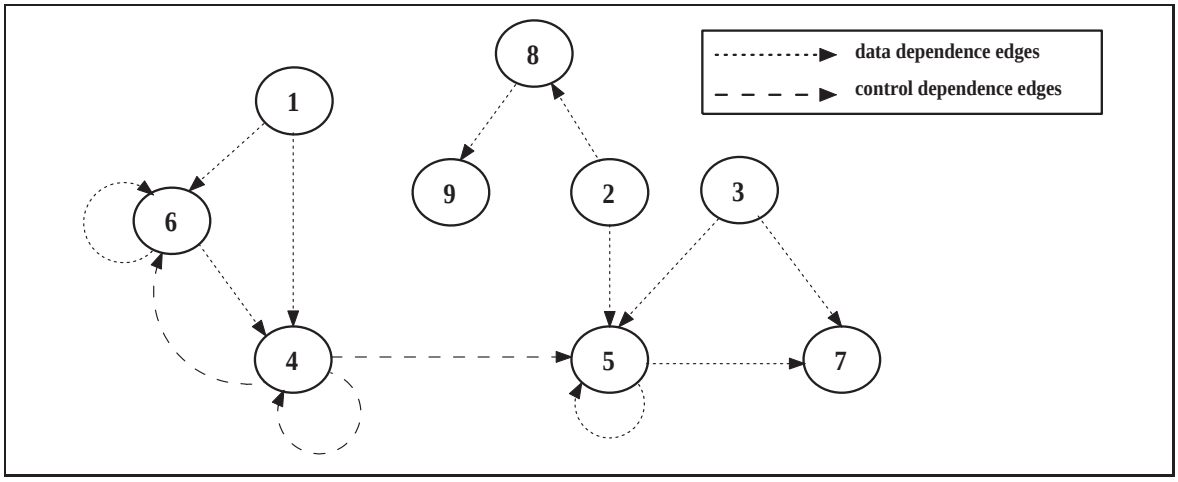


Fig. 2.5: PDG of the program in Figure 2.3.

- *Actual-in* (V_{A-in}) and *actual-out* (V_{A-out}) nodes represent the input and output parameters at a call site. They are control dependent on the corresponding *call-site* node.
- *Formal-in* (V_{F-in}) and *formal-out* (V_{F-out}) nodes represent the input and output parameters at the called procedure. These nodes are control dependent on the procedure's entry node.

Let $\mathbf{ET_{SDG}}$ denote the different types of edges of an SDG. Then, $\mathbf{ET_{SDG}}$ can be expressed as:

$$\mathbf{ET_{SDG}} = \{E_{cd}, E_{dd}, E_{ce}, E_{Par-in}, E_{Par-out}, E_{Sum}\},$$

where each individual member of the set $\mathbf{ET_{SDG}}$ represents a particular edge type. In the following, we explain the different types of edges of an SDG:

- *Control* (E_{cd}) and *data* (E_{dd}) dependence edges represent control and dependence relationships among the nodes of an SDG respectively.
- *Call* edges (E_{ce}) link the *call-site* nodes with the corresponding procedure entry nodes.
- *Parameter-in* edges (E_{Par-in}) connect the *actual-in* nodes with the respective *formal-in* nodes.
- *Parameter-out* edges ($E_{Par-out}$) connect the *formal-out* nodes with the respective *actual-out* nodes.
- *Summary* edges (E_{Sum}) are used to represent the transitive dependencies that arise due to function calls. A summary edge is added from an *actual-in* node

a to an *actual-out* node *b* if the value associated with *b* can get affected by the value associated with the node *a* due to control or data dependence. That is, a summary edge is added from *a* to *b* if there exists either a control or data dependence edge from the corresponding *formal-in* node *a'* to the *formal-out* node *b'*.

SDG is a generalization of the PDG representation. In fact, the PDG of the main procedure of a program is a sub-graph of the SDG. In other words, for a program without procedure calls, the PDG and SDG models are identical. The technique for constructing an SDG consists of first constructing a PDG for every procedure, including the main procedure, and then interconnecting the PDGs at the call sites.

Example 2.1: Figure 2.6 shows a simplified version of a C program of an automotive application developed on a VxWorks [127] platform. The corresponding SDG model for the program has been shown in Figure 2.7. In Figure 2.7, control dependence edges are represented by dash-dot-dash edges, while the dotted edges represent data dependence edges. The other types of SDG edges such as *parameter-in*, *parameter-out*, *call* edge, etc. have been represented by uniformly-spaced dashed edges. Please note that we have not shown all *actual-in* and *actual-out* nodes in the figure to avoid clutter.

2.3. Concepts Related to Embedded Software

In the following, we briefly review the standard task models that are popularly being used in the development of embedded applications. We also discuss the precedence relationships that may exist among tasks.

2.3.1. Task Model

A periodic task τ_i is represented by a four tuple, $\tau_i = \langle \phi_i, p_i, e_i, d_i \rangle$, where ϕ_i is the phase of τ_i, p_i is the period of τ_i, e_i is the WCET of τ_i, and d_i is the relative deadline (with respect to ϕ_i) of τ_i. In our work, without any loss of generality, we assume the phases of all tasks to be zero. Therefore, we represent a task τ_i by the three tuple $\langle p_i, e_i, d_i \rangle$.

OSEK/VDX [83] and POSIX RT [51,69] are two real-time operating system standards that are popularly being used in the development of embedded applications.

```
D0  MSG_Q_ID g_msgq=NULL;

E1  int main( void )
    {
S2    int l_monitor = ERROR;
S3    int l_varyspeed = ERROR;

S4    l_monitor = taskSpawn("tMonitor",100,0,
        10000, (FUNCPTR)monitor,0, 0, 0, 0, 0, 0,
                  0, 0, 0, 0);
S5    l_varyspeed = taskSpawn("tVarySpeed",100,0,
        10000, (FUNCPTR)vary_speed,0, 0, 0, 0, 0,
                  0, 0, 0, 0, 0);

S6    if(ERROR!=l_monitor && ERROR!=l_varyspeed)
S7      g_msgq = msgQCreate(50, 4, MSG_Q_FIFO);

S8    return 0;
    }

E2  void vary_speed(void)
    {
S9    float vel;
S10   while(true)
      {
S11     msgQReceive(g_msgq,(char *)&vel,
                    50,WAIT_FOREVER);
        /*increase/decrease the acceleration*/
S12     if(-1==vel)
S13       increase_acc();
S14     else if(1==vel)
S15       decrease_acc();
      }
    }

E3  void monitor(void)
    {
S16   float speed;
S17   int flag=0;

S18   while(true)
      {
        /*read the current speed*/
S19     speed=get_speed();
S20     if(speed>75)
        {
S21       flag=1;
S22       msgQSend(g_msgq,(char *)&flag,4,
            WAIT_FOREVER, MSG_PRI_NORMAL);
        }
S23     else if (speed<25)
        {
S24       flag=-1;
S25       msgQSend(g_msgq,(char *)&flag,4,
            WAIT_FOREVER, MSG_PRI_NORMAL);
        }
        else
S26       flag=0;
      }
    }
```

Fig. 2.6: A sample VxWorks program incorporating inter-task communication.

In this paper, we consider a task model that adopts features from both these standards. We assume that the tasks are statically created. This of course is usually the case for many embedded applications. The tasks have statically assigned priorities. The tasks are periodic and are scheduled using a priority-driven preemptive task scheduler. The tasks are assumed to communicate using either shared memory or message passing. During inter-task communication, synchronization among tasks is typically achieved by using the following two techniques:

- Shared memory: Shared memory is an important means of communication among producer and consumer tasks. The producer tasks share data by writing on to some shared variables; and the consumer tasks read data by reading those shared variables. To achieve deterministic task execution, it is a general practice in embedded programming to guard access to shared memory through the use of synchronization primitives such as semaphores and locks [117].
- Message passing: To achieve predictable results, embedded application developers usually restrict themselves to using the synchronous message passing mechanism [117]. Synchronous message passing requires that the sender

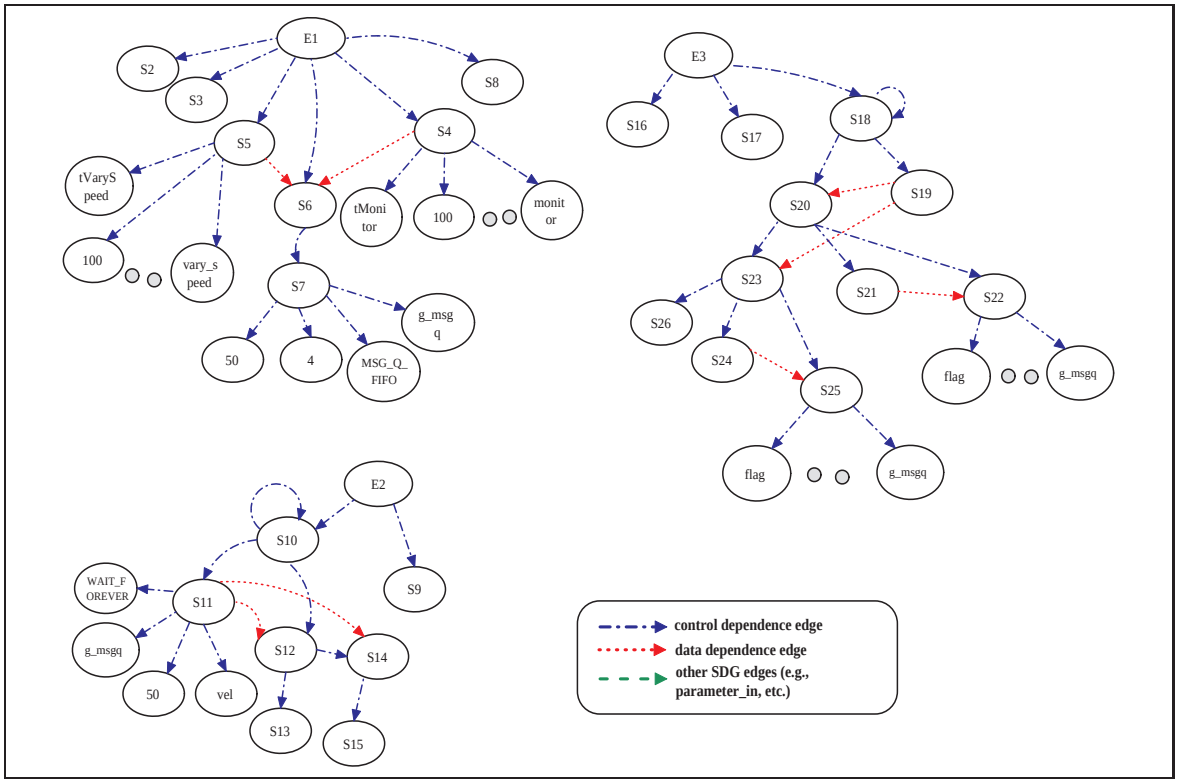


Fig. 2.7: SDG model for the program of Figure 2.6.

and the receiver tasks wait for each other before a message transfer occurs.

Our task model reflects the assumptions frequently made in the development of small embedded applications. For example, an adaptive cruise controller (ACC) module in an automotive application is usually implemented with about a dozen periodic real-time tasks with statically assigned priorities. The tasks are scheduled using the rate monotonic scheduling algorithm [69]. Some of the important concurrently executing tasks in a typical ACC implementation include controlling the host vehicle speed (HVSM task), and processing the radar information (RIP task).

2.3.2. Task Precedence

Two tasks in an embedded application are *precedence ordered* when one task is dependent on the actions or the results produced by the other task. For example, if τ_i is a producer task and τ_j is the consumer task, then τ_i must precede τ_j. When a task τ_i precedes another task τ_j, then each instance of task τ_i precedes the corresponding instance of τ_j. Precedence relationships define a partial ordering among tasks. An example of precedence ordering among tasks has been shown in

Figure 2.8. The circles in Figure 2.8 represent tasks while the edges among them represent precedence relationships. A directed edge from a task τ_i to τ_j indicates that τ_j is dependent on τ_i. From Figure 2.8, it can be inferred that τ_1 and τ_4 precede τ_2 and τ_5 respectively. However, we cannot ascribe any precedence ordering between the tasks τ_1 and τ_4 or the tasks τ_1 and τ_5.

We denote the precedence ordering of a task τ_i with other tasks by using two functions: $Pred(\tau_i)$ is the set of all those tasks whose execution must be complete before execution of task τ_i can be started. $Succ(\tau_i)$ is the set of tasks whose execution can start only after the task τ_i has completed. For the example shown in Figure 2.8, $Succ(\tau_1) = \{\tau_2, \tau_3\}$, and $Pred(\tau_5) = \{\tau_4\}$.

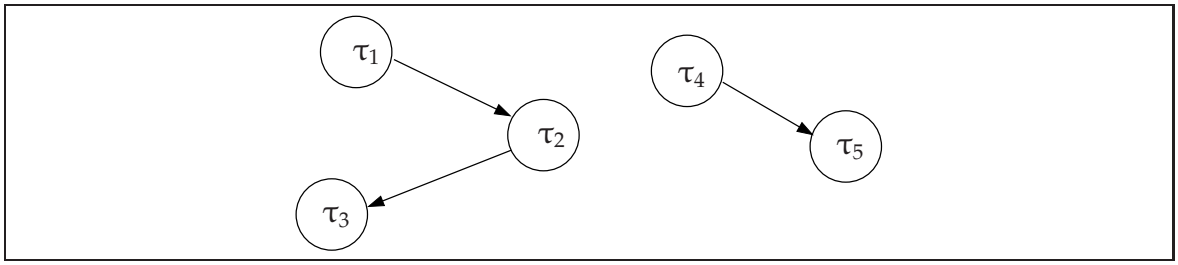


Fig. 2.8: A representation of the precedence relations among tasks.

2.3.3. Exception Handling in Embedded Systems

Invocation of exception handlers may delay the execution of certain tasks in an embedded program. In this context, a few studies have reported techniques to minimize the overhead of exception handling [88]. Since the C programming language does not directly support exception handling, therefore many embedded programs implement exception handling using jumps and `switch-case` constructs or through use of specific libraries [106]. In this work, we assume that exception handling in embedded programs is implemented using the C++ `try-catch` model. In this model, exception handling is achieved by using `try`, `catch` and `throw` statements. In C++, the code which can potentially raise an exception (known as *throwing*) is enclosed within a `try` block. When an exception is raised, the `catch` block corresponding to the raised exception is executed. The type of exception raised can be any valid data type including user-defined classes. In the absence of a matching `catch` block, the default `catch` block (which is indicated using an ellipsis ...) is executed. If a matching handler is not found, then the program execution gets terminated. However, if no exceptions are thrown in a `try` block, then the corresponding `catch` blocks are not executed. The different execution

paths that can be followed when an exception is thrown is discussed in more detail in [21,54].

2.4. Genetic Algorithms

Many search and optimization problems in engineering tend to be multi-objective in nature, i.e., it requires simultaneous optimization of a number of possibly conflicting objectives. Traditional algorithms such as linear programming and gradient-based methods when applied to solve multi-objective optimization (MOO) problems may produce sub-optimal results and get stuck at a local optima [105]. For these traditional class of algorithms, the convergence to an optimal solution depends on the initial solution. Evolutionary algorithms such as genetic algorithms (GA) [38], particle swarm optimization [56], etc. are the preferred choice for ssolving MOO problems.

Genetic algorithms (GA) are a part of the evolutionary family of algorithms and are extensively used to solve search and multi-objective optimization problems [38]. They are derivative-free stochastic optimization methods and are based on the principle of evolution and natural genetics. GAs were first proposed by Holland in 1976. A few reasons which have contributed to the popularity of GAs are its broad applicability, ease of understanding, and intuitive appeal. GAs are based on the principle of survival of the fittest. GAs encode a potential solution to a specific problem using a chromosome-like data structure. Each chromosome encodes possible solution and is represented as a binary string. Evolution across generations is incorporated using a structured yet randomized information exchange among possible solutions. The possible genetic operations include *selection*, *crossover* and *mutation*. Selection involves replication of chromosomes to the next generation. Crossover involves exchanging part of the string between to mating chromosomes. Mutation simply flips one bit in the binary chromosome string based on a probability.

The interested reader is referred to [38] for more detailed information of GAs.

2.5. Conclusion

In this chapter, we have presented an overview of the concepts which form the background of our research investigations. The idea was to provide some basic

concepts and definitions that would help the reader to understand the work presented in the subsequent chapters. We started with a review of the basic concepts related to regression testing. Subsequently, we discussed a few popular graph models proposed for procedural programs. We also discussed the basic assumptions regarding the task models that will be used in our work, and provided a brief overview of genetic algorithms.

Chapter 3

Review of Related Work

A large number of research results are available in the general areas of RTS, TSM, and TCP, for procedural and object-oriented programs. However, in spite of our best efforts, we could not find any reported results on selection and optimization of regression test suites for embedded applications. In the absence of any directly comparable work, in this chapter we review the related work on RTS and RTSO and discuss few important techniques that have an indirect bearing on our work.

This chapter has been organized as follows: We first review RTS techniques proposed in the context of different programming paradigms in Section 3.1. We focus on RTS techniques proposed for procedural programs in subsection 3.1.1. Subsequently, we discuss RTS techniques for object-oriented and component-based applications in subsections 3.1.2 and 3.1.3 respectively. Since in this work, we have assumed that procedural languages are used for embedded software development, therefore we have kept our discussions on different RTS techniques for object-oriented and component-based software to a minimum, and only highlight some challenges in carrying out satisfactory RTS. We briefly detail the different surveys carried out in the field of RTS in Section 3.1.4. We review RTSO techniques in Section 3.2, and finally conclude the chapter in Section 3.3.

3.1. Regression Test Selection Techniques

A large number of RTS techniques have been reported for procedural [13, 15, 19, 42, 45, 46, 62, 64, 65, 94] and object-oriented programs [11, 22, 44, 82, 95], each aimed at leveraging certain optimization options. Different techniques trade-off differently with regards to the cost of selection and execution of test cases and fault detection effectiveness. In the recent past, the problem of RTS has actively been

investigated and new approaches have emerged to keep pace with the newer programming paradigms. During the last decade, there has been a proliferation in the use of different programming paradigms such as component-based development, aspect-oriented programming, embedded and web applications, etc. It is therefore not surprising that a number of RTS techniques have been proposed for recent programming paradigms such as component-based [37,71,72,81,137–139], aspect programs [131,136], and web applications [67,99,100,114,132].

3.1.1. RTS Techniques for Procedural Programs

RTS techniques were first studied in the context of procedural programs [63,64]. RTS for procedural programs is therefore an extensively researched topic and many techniques have been proposed over the years [13,15,19,25,42,45,62,64,65,90,94, 113,119,120]. These techniques select relevant test cases using either control flow, data or control dependence analysis, or by textual analysis of the original and the modified programs. Depending on the type of the program analysis technique used and to aid in understanding, we have grouped the different RTS techniques into the following major classes:

1. Dependency-based techniques - These consist of the following sub-classes:
   (a) Data dependence-based techniques [42,45,113]
   (b) Slicing-based techniques [15,19]
2. Firewall-based techniques [64,65]
3. Differencing based approaches - These consist of the following techniques:
   (a) Modified code entity-based technique [25]
   (b) Textual differencing-based technique [119,120]
4. Control flow analysis-based techniques [13,62,94]

In the following, we discuss in detail the control flow analysis-based safe RTS technique proposed by Rothermel and Harrold [94], the textual diferencing-based technique proposed by Vokolos and Frankl [119], and the SDG-based slicing technique proposed by Binkley [19].

3.1.1.1. Dependency-Based Techniques

In the following, we review the RTS techniques that have been proposed based on various types of dependency analysis such as data and control [19,42,45,64,65,113].

Data Dependence-Based Techniques: These techniques [42, 45, 113] explicitly detect definition-use pairs for variables that are affected by program modifications, and select test cases that exercise the paths from the definition of modified variables to their uses. The use of a variable is further distinguished into computation uses (c-uses) and predicate uses (p-uses). A c-use occurs for a variable if it is used in computations, and a p-use occurs when it is used in a conditional statement. A c-use may have an indirect effect on the control flow of the program, while a p-use may either directly affect the flow of control or may also indirectly affect some other program statements. The dataflow analysis-based RTS techniques identify the def-use pairs for all variables that are affected by a change and select only those test cases, which when executed on P, exercise those affected def-use pairs in P'.

Data dependence-based RTS techniques reported in [45, 113] usually carry out analysis either by processing the changes one by one and then incrementally updating the data dependence information for P', or compute the full data dependence information for P and P', and compare the differences between def-use pairs. Both these approaches require saving the data dependence information across testing sessions or recompute them at the beginning of each testing session. The RTS technique proposed by Gupta et al. [42] is based on inter-procedural slicing which does not require saving or recomputing the dataflow information across testing sessions. The technique uses the concepts of backward and forward slices to determine the affected def-use pairs that must be retested. The program is sliced to select test cases that execute the affected def-use pairs.

Dependence Graph-Based Slicing Techniques: Slicing-based techniques for RTS of procedural programs identify all program elements that are indirectly affected by a modification to the original program P.

A PDG-based slicing approach for procedural programs was proposed by Bates and Horwitz [15]. However, the PDG-based slicing technique did not support inter-procedural regression testing. In [19], Binkley proposed an inter-procedural RTS technique based on slicing SDG models of P and P'. Two components are said to have equivalent execution patterns if and only if they are executed the same number of times on any given input [19]. The concept of *common* execution patterns [19] has been introduced as an inter-procedural extension of the *equivalent* execution patterns proposed in [15]. Code elements are said to have a common execution pattern if they have the same equivalent execution pattern during some call to procedures. The common execution patterns capture the semantic differ-

ences among code elements [19]. The semantic differences between P and P' are determined by comparing the expanded version (i.e., with every function call expanded in place) of the two programs. The expanded versions of the two programs are analyzed to find out affected program elements which need to be regression tested.

Critical Evaluation: The techniques reported in [42,45] are based on computing dataflows in a program and do not consider control dependencies among program elements for selecting regression test cases. These techniques are also not able to detect changes that do not cause changes to the dataflow information [134]. Hence, these techniques are unsafe. Dataflow techniques are also imprecise because the presence of a affected definition or use in a new block of code does not guarantee that all test cases which execute the block will execute the affected code [93]. Examples illustrating the unsafe and imprecise nature of dataflow-based techniques are available in [93].

According to the studies reported by Rothermel and Harrold [93], the PDG [15] and SDG-based [19] slicing techniques are not safe when the changes to the modified program involve deletion of statements. The techniques are also imprecise. The slicing-based RTS techniques are computationally more expensive than the dataflow analysis-based techniques. However, slicing-based RTS techniques can be applied to select test cases for both intra- and inter-procedural modifications.

3.1.1.2. Module Level Firewall-Based Techniques

The firewall-based approach, first proposed by Leung and White [64,65], is based on analysis of data and control dependencies among modules in a procedural program. A *firewall* is defined as the set of all the modified modules in a program along with those modules which interact with the modified modules. The firewall is a conceptual boundary that helps in limiting the amount of retesting required by identifying and limiting testing to only those modules which are affected by a change. The firewall techniques use a *call graph* to represent the control flow structure of a program [64]. Module A is called an ancestor of module B, if there exists a path (a sequence of calls) in the call graph from module A to B, and module B is then called a descendant of module A. The direct ancestors and the direct descendants of the modified modules are also considered during the construction of a firewall to account for all possible interactions with the modified modules. The test coverage information for P is used to select the subset of test cases $t \in T$ which

exercise the affected modules included in the firewall.

Critical Evaluation: The firewall technique is not safe as it does not select those test cases from outside the firewall that may also execute the affected modules within the firewall [93]. The firewall techniques are imprecise because all test cases which execute the modules within the firewall do not necessarily execute the modified code within modules. However, the firewall techniques are efficient because the approaches consider only the modified modules and relationships with other modules in the firewall, and hence limits the total amount of the source code that needs to be analyzed. The firewall techniques handle RTS for inter-procedural program modifications but are not applicable for intra-procedural modifications [93].

3.1.1.3. Differencing-Based Techniques

In this subsection, we discuss RTS techniques [25,119] that are based on an analysis of the differences between the original and the modified programs.

Modified Code Entity-Based Technique: A modified code entity-based RTS technique was proposed by Chen *et al.* [25] for C programs. They have decomposed program elements into functional and non-functional code entities. A code entity is defined as either a directly executable unit of code such as a function or a statement, or a non-executable unit such as global variable and macro. The original program P is executed with each test case $t \in T$. The test coverage information is analyzed to determine the set of executable code entities that are exercised by each test case $t \in T$. For each function that is executed by a test case t, the transitive closure of the global variables, macros, etc. referenced by the function is computed. When the original program P is modified, all the code entities which were modified to create the revised program P' are identified. Test cases that exercise any of the modified entities are selected for regression testing of P'.

Technique Based on Textual Differencing: Vokolos and Frankl [119,120] have proposed an RTS technique which is based on a textual differencing of the original and the modified programs (i.e., P and P'), rather than using any intermediate representation of the programs. A naive textual differencing of the programs will include trivial differences between the two versions such as insertion of blank lines, comments etc. Therefore, their technique first converts a program to its *canonical*

form [118, 119] before comparison. This conversion ensures that the original and the modified programs follow the same syntactic and formatting guidelines. The canonical version of P is instrumented to generate test coverage information. The test coverage information includes the basic blocks that are executed by each test case instead of the program statements. The canonical versions of P and P' are syntactically compared to find out modifications to the code. The test coverage information is then used to identify test cases which execute the affected parts of the code.

Critical Evaluation: The modified code entity technique is safe because it identifies all possible affected code entities, and selects regression test cases based on test coverage [16, 93]. The technique proposed in [119] is also safe because it identifies all the basic blocks that are affected due to modifications and selects regression test cases that execute those basic blocks. However, both the techniques are imprecise. For example, if a function f is modified, the modified code entity technique selects all those test cases which execute f. But there might be tests which execute f without executing the modified code in f. The textual differencing technique can be highly imprecise when code changes are arbitrary since differentiation is based on only syntax and the test cases are selected based on coverage of basic blocks. The code entity technique is considered to be the most efficient and safe RTS technique for procedural programs [93], and its time complexity is bounded by the size of T and P. The time complexity of the textual differencing technique is $O(|P|*|P'|*log|P|)$ which may not be scalable for large programs.

3.1.1.4. Control Flow Analysis-Based Techniques

A few RTS techniques [13, 62, 94] have been proposed which analyze control flow models of the input programs for selecting regression test cases. We briefly discuss these RTS techniques in the following.

Cluster Identification Technique: The main concept used in the cluster identification technique proposed by Laski and Szermer [62] is localization of program modifications into one or more areas of the code referred to as *clusters*. Clusters are defined as single-entry, single-exit parts of code that have been modified from one version of a program to the next. The cluster identification technique models programs P and P' as CFGs (denoted by G and G'). The nodes in G and G', which correspond to the modifications in the code, are identified, and the set of all such

identified nodes in G and G' are marked as clusters. A cluster identification-based technique uses control dependence information of the original and the modified procedures to compute the clusters in the two graphs.

Once the clusters have been identified in the CFGs, each cluster is then represented by a single node to form a *reduced* CFG. Analysis of the reduced flow graphs is based on the assumption that any complex program modification can be achieved by one of the following three operations: inserting a cluster into the code, deleting a cluster, or changing the functionality of a cluster. Test cases are classified into two categories: local to the clusters and global in the entire program. The former includes test cases which execute modified clusters, and the latter includes test cases which execute other areas of the program affected due to the modified clusters based on control dependencies. The test coverage information is then used to select regression test cases.

Graph Walk-Based Technique: Rothermel and Harrold have proposed an RTS technique based on traversal of CFGs of the original and the modified programs [94]. The approach proposed in [94] involves constructing CFGs G and G' for programs P and P' respectively. The execution trace information for each test case t, $ET(P(t))$, is recorded. This is achieved by instrumenting P. In [94], a simultaneous depth-first traversal of the two CFGs G and G' is performed corresponding to each modified procedure in P and P'. The traversal is performed according to the test case execution trace for each test case in T. For each pair of nodes n and n' belonging to G and G' respectively, the technique finds out whether the program statements associated with the successors of n and n' along identically-labeled edges of G and G' are equivalent or not. If a pair of nodes n_1 and n'_1 is found such that the statements associated with n_1 and n'_1 are not identical, then the edges that lead to the non-identical nodes are identified as *dangerous* edges. Test cases which execute the set of identified dangerous edges are assumed to be modification-revealing. Therefore, a test case $t \in T$ is selected for retesting P' if $ET(P(t))$ contains node n_1.

DFA Model-Based Approach: Ball [13] has proposed a more precise RTS technique as compared to [94] by modeling the CFG G for a program P as a deterministic finite state automaton (DFA). A DFA M for a CFG G can be constructed such that the following conditions hold:

1. Each node v in G corresponds to two nodes $v1$ and $v2$ of M, such that $v1 \rightarrow_{BB(v)} v2$, where $BB(v)$ is the basic block associated with node v in G.

2. $L(M)$ = the set of all possible complete paths in G.

Ball introduced an intersection graph model for a pair of CFGs G and G' corresponding to the original and modified programs. The intersection graph also has an interpretation in terms of a DFA. Ball's RTS technique is based on reachability of edges in the intersection graphs. The techniques use edge coverage criterion as the basis for RTS analysis.

Critical Evaluation: The RTS techniques proposed in [13,62,94] are safe. Among the three techniques, the cluster identification technique is comparatively more imprecise because test cases are selected based on whether they execute a cluster rather than the affected statements. The techniques proposed in [13,94] are the two most precise procedural RTS techniques. However, Ball's DFA-based approach is computationally more expensive than [94]. The time complexity of the cluster identification technique is bounded by the time required to compute the control scope of decision statements and is dependent on the input program size.

Ball has proposed another technique [13] which uses path coverage criterion and is still more precise than [13,94]. The higher precision is attributable to the fact that path coverage is stronger than an edge coverage criterion. This increase in precision is however accompanied by an increase in the computation effort. Additionally, it cannot analyze control flows across procedures and hence cannot be applied for RTS of inter-procedural code modifications.

An important difference between graph walk and slicing-based techniques is that the latter uses dependence relationships to analyze the source code and identify the modified regions in the source code. Regression test selection is performed by monitoring the execution of the sliced region of code on T. On the other hand, the graph walk techniques use comparison of graphical models of the program to identify the modifications [90,94].

3.1.2. RTS Techniques for Object-Oriented Programs

The object-oriented paradigm is founded on several important concepts such as encapsulation, inheritance, polymorphism, dynamic binding, etc. These concepts lead to complex relationships between the various program elements, and make dependency analysis more difficult [126]. Moreover, in object-oriented development, reuse of existing libraries, class definitions, program executables (black-box

components), etc. are emphasized to facilitate faster development of applications. These libraries and components frequently undergo independent modifications to fix bugs and enhance functionalities. This creates a new dimension in regression testing of object-oriented programs that use these third-party components or libraries, since the source code for such libraries are often not available. These features, therefore, raise challenging questions on how to effectively select regression test cases that are safe for such programs [17,77].

The reported RTS techniques for object-oriented programs can broadly be classified into the following three major categories:

1. Firewall-based techniques [9,50,52,61]
   (a) Class firewall technique [61]
   (b) Method level firewall technique [52]
2. Program model-based techniques [44,82,91,95]
3. Design model-based techniques [11,22,33,39,79]

3.1.3. RTS Techniques for Component-Based Software

In the component-based software development model, a software product is developed by integrating different components developed either in-house or by third-party vendors. The reliability of a component-based software application, to a large extent, depends on the reliability of the individual components. These blackbox components are often modified by the concerned vendor to fix bugs and incorporate enhancements. Hence, regression testing of component-based software needs to address how the changes made to a component might affect the execution of application programs which use those modified components. Techniques which perform RTS of traditional programs cannot meaningfully be used for RTS of software using COTS (Commercial Off-The-Self) components because the code for the components are usually not available. RTS for component-based software is a challenging research problem due to the following reasons [37,81]:

- In a component-based development environment, often there is a lack of adequate information about the changes made to each release of a component. Relevant information such as control and data flow relationships among the modules are usually not supplied to the application programmer. Moreover, there is also a lack of adequate documentation for third-party components.

- A change made to a component may be reflected both at the component level and at the system level functioning of the software. Even trivial changes made to a component in a system may at times affect the proper working of the software as a whole.
- There is a lack of test tools which can be used to identify changes in a component and its impact on the software.

Depending on the type of program analysis, the RTS techniques for component-based software can be grouped into the following classes:

1. Metacontent-based RTS approaches
   (a) Code coverage-based approaches [81]
   (b) Enhanced change information-based approaches [71,72]
2. Model-based techniques
   (a) UML model-based techniques [101,129]
   (b) Component model-based techniques [37]
   (c) Dynamic behavior and impact analysis using models [85]
3. Analysis of executable code [137–139]

3.1.4. Survey of RTS Techniques

RTS techniques have been reviewed by several authors [14,16,31,40,93,134]. In [93], Rothermel and Harrold have proposed a framework (i.e., a set of parameters) to evaluate the effectiveness of different RTS techniques. Baradhi and Mansour [14], Bible et al. [16], and Graves et al. [40] have performed experimental studies on the performance and effectiveness of different RTS techniques proposed for procedural programs. Based on these studies, it is difficult to choose any technique as the best because these empirical studies have been performed on different categories of programs and also under different conditions [32]. This lead Engström et al. to perform a qualitative study [31,32] of the nature of the empirical data considered. The studies reported in [31,32] are based on the similarities of the different RTS techniques and the quality of the empirical data used. Engström et al. [32] observe that that it is very difficult to come up with a RTS technique which is generic enough (i.e., can be applied to a different classes of applications) and is superior to all other techniques.

The interested reader is referred to [20] for a detailed and in-depth review of the different RTS techniques proposed in the literature.

3.2. Regression Test Suite Optimization Techniques

Most test suite optimization (TSO) techniques reported in the literature have been proposed in the context of traditional programs. In spite of our best efforts, we could not find any study which specifically addresses the problem of optimizing regression test suites of embedded applications. In the absence of any directly comparable work, we compare our RTSO technique with the conventional approaches [34, 135].

Farooq and Lam [34] have proposed a non-pareto min-max based TSO technique which removes redundant test cases and maximizes the branch coverage. Zhang et al. [135] have proposed a resource-aware RTSO approach by combining selection and prioritization of test cases. Their approach first selects the relevant test cases and then prioritizes the selected test cases based on the given resource constraints.

A few studies reported in the literature refer to TSM [84] and TCP [48, 121] techniques as optimization techniques aimed at improving the effectiveness of regression testing. TCP techniques are considered as optimization approaches for T because they aim to improve certain regression testing criteria: improving the rate of fault detection, improving the rate of coverage achieved, etc [48, 97, 121]. A drawback inherent in using these approaches is that these approaches primarily aim to optimize the rate of detection of faults. Usually the total number of faults detected during a typical regression testing session is much smaller than the number of faults detected during a product testing session. Unlike product testing, improving the rate of fault detection during regression testing can be considered to be less important as compared to the total number of faults detected.

The reported techniques [34,48,84,121,135] may not produce satisfactory results when used to optimize regression test suites of embedded applications. This is because these techniques do not consider the execution dependencies introduced among tasks in an embedded program due to modifications made to the other parts of the code. Neither the minimization nor the prioritization techniques distinguish test cases which execute critical or non-critical functionalities of an embedded program. Most of these approaches also do not aim to optimize the objectives of execution cost of test cases and total path coverage *at the same time*.

3.3. Conclusion

In this chapter, we have presented a brief review of the various RTS techniques proposed for procedural, object-oriented and component-based software. We have highlighted the relative merits and demerits of procedural RTS techniques. We have also discussed existing TSO approaches proposed in the context of optimizing regression test suites for traditional programs. In the absence of any directly comparable work, we have also discussed those techniques which are only marginally related to our work.

Chapter 4

Task Execution Dependencies in Embedded Programs

Embedded software developers not only target to meet the stringent reliability requirements of embedded software, but also aim to address a a set of diverse and conflicting issues such as extensibility, efficiency, cost, etc [104]. This makes designing and development of embedded applications a challenging task.

After the development of an embedded software is complete, it is tested to check for possible bugs in the code. Unlike traditional programs, the failures of an embedded program arise from both *functional* errors as well as *timing* bugs. Therefore, in addition to functional correctness of an embedded application, it is also necessary to guarantee its temporal correctness. Considerable number of studies on issues such as computation of the WCETs of tasks and priority inversions arising on account of resource sharing has been reported in the literature. Though research results on such timing analysis of tasks are numerous, no studies on systematic identification of execution dependencies of tasks have been reported.

The completion time of a task of an embedded application can be affected by other concurrent and communicating tasks. Besides control and data dependencies, additional dependencies arise among tasks due to precedence relations, task priorities, and inter-task communications [73]. We call these *task execution* dependencies. Upon a code change, the effect of task execution dependencies can manifest as delays to task completion times or altered task execution sequences. Thus, it is a major challenge for software developers to always ensure that the performance constraints of real-time tasks are still satisfied even after modifications to parts of the code.

Systematic identification of task execution dependencies among tasks can be of

use in many software engineering activities of embedded programs such as RTS, task prioritization, debugging of timing errors, computation of complexity metrics, etc. For example in RTS, in addition to testing for traditional regression errors induced due to data and control dependencies after a change to an embedded program, it is imperative to test whether any timing errors have been induced. Similarly, while *debugging* a timing fault, it becomes necessary to first identify all those tasks that might have contributed to the unexpected timing behavior. The information about the number of tasks that are execution dependent on a particular task can also be used to *prioritize* testing effort.

It is important to note that task execution dependencies are captured neither by data nor by control dependencies. In this chapter, we discuss execution dependencies that can arise among tasks in an embedded program due to various factors such as task precedence, task priority, message passing, use of shared resources, etc., and their effect on the execution of tasks.

This chapter is organized as follows: In Section 4.1, we discuss task execution dependencies that arise due to precedence order. We discuss task execution dependencies that arise due to priorities in Section 4.2. We discuss task execution dependencies that arise due to message passing and use of shared resources in Sections 4.3 and 4.4. We discuss the task execution dependencies that are introduced due to interrupt handling in Section 4.5. We discuss a possible side-effect of task execution dependencies on the execution of embedded programs in Section 4.6. Finally, we conclude the chapter in Section 4.7.

4.1. Task Execution Dependency Due to Precedence Order

Given a set of time-constrained tasks, the completion of a task is dependent on the task precedence order (if any) among the set of tasks. A task τ_i executes only after the set of tasks in $Pred(\tau_i)$ have already completed their execution. Therefore, we can say that a task is execution dependent on each task in $Pred(\tau_i)$. The effect of the dependencies introduced due to precedence ordering on the execution of a task is stated in Proposition 1.

Proposition 1. *Modifications made to a task τ_i can affect the completion time of the tasks in the set $Succ(\tau_i)$.*

We explain the effect of execution dependencies introduced due to precedence order with the help of the following example.

Example 4.1: Let us consider an embedded program P consisting of three tasks $\tau_1 = \langle 10, 3, 4\rangle$, $\tau_2 = \langle 10, 3, 8\rangle$ and $\tau_3 = \langle 10, 2, 8\rangle$. Let us further assume that the task τ_1 precedes task τ_2 and τ_2 precedes task τ_3. A possible schedule of the tasks in P is shown in Figure 4.1a. Suppose task τ_1 is modified in P'. It is possible that τ_1 in P' takes longer to execute, say 3.5 units, due to the modification. This is represented in the schedule shown in Figure 4.1b. As a result of the change, the completion of the tasks in $Succ(\tau_1) = \{\tau_2, \tau_3\}$ will also get delayed by 0.5 time units.

Execution dependencies arising among tasks due to their precedence ordering are transitive in nature. The set of tasks that are execution dependent on a task τ_i due to precedence relations is same as $Succ(\tau_i)$. From Proposition 1, it can be inferred that it is important to select test cases for regression testing those tasks which are execution dependent due to precedence relationships on some task which has been directly modified in P'.

4.2. Task Execution Dependency Due to Priorities

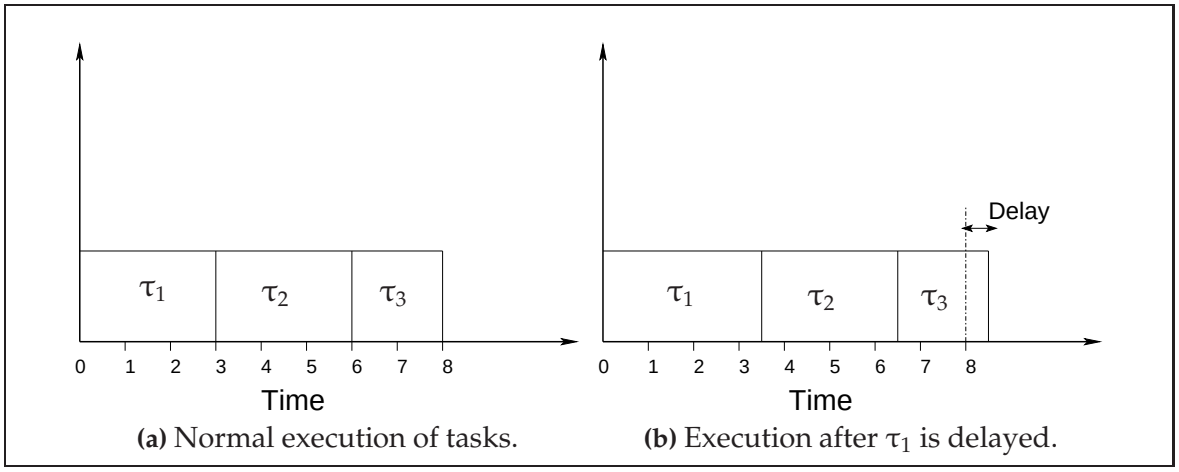
Execution dependencies among tasks can arise on account of task priorities. This is because the lower priority tasks will not be able to execute unless a higher priority task completes its execution.

Proposition 2. *A delay to the completion time of a higher priority task can lead lower priority tasks to miss their deadlines.*

We illustrate the effect of execution dependencies introduced due to task priorities with the help of the following example and Figure 4.1.

Example 4.2: Let us consider the three tasks τ_1, τ_2 and τ_3 shown in Figure 4.1. Let the priority of each task be as follows: priority(τ_1) > priority(τ_2) > priority(τ_3). Assume that the execution order of the tasks are τ_1, τ_2, τ_3. Suppose task τ_1 is changed and as a result takes longer to execute, say 3.5 units. This would delay the other two tasks as shown in Figure 4.1b.

Execution dependencies among tasks due to priorities are transitive in nature. For a given task τ_i, we denote the set of all lower priority tasks whose execution time can potentially be affected by τ_i by $Prior(\tau_i)$.



(a) Normal execution of tasks. **(b)** Execution after τ_1 is delayed.

Fig. 4.1: Task delay caused due to precedence relationships.

4.3. Task Execution Dependency Due to Message Passing

Synchronous message passing in an embedded program gives rise to execution dependencies among a pair of communicating tasks. The effect of task execution dependence due to message passing is expressed in the following proposition.

Proposition 3. *Modification to either the sender or the receiver task of a pair of tasks communicating using message queues may delay the completion of the other task.*

Since we have assumed synchronous mode of communication, therefore the message transfer will not take place unless the two tasks are ready to send and receive data respectively. Any delay to one task will force the other task (sender/receiver) to wait.

Task execution dependencies arising due to message passing are both symmetric and transitive in nature. Task execution dependence due to message passing is a symmetric relation under the assumption of synchronous message passing as both the sender and the receiver tasks can delay each other. For a task τ_i, we denote the set of tasks that can possibly get delayed by it due to the execution dependencies arising due to messaging passing by $ITC_{mp}(\tau_i)$.

4.4. Task Execution Dependency Due to Use of Shared Resource

Execution dependencies can arise among a set of tasks when they share a resource. The effect of task execution dependence introduced due to use of a shared resource is stated in the following proposition.

Proposition 4. *A task locking a synchronization variable for an unusually long duration may delay other tasks sharing the same variable.*

We assume that access to shared variables are usually guarded using synchronization primitives such as semaphores or locks. The program statements for accessing such primitives are implicit synchronization points between the concerned tasks. Therefore, any increase to the duration for which one task locks a synchronization variable may cause the other tasks needing to lock the variable to get delayed.

Example 4.3: In Figure 4.2, we show an example of execution dependencies that can exist among two tasks communicating using a shared variable. The tasks τ_1 and τ_2 in Figure 4.2 communicate using the shared variable *var* and use a semaphore variable *sem* to guard the access to *var*. Let us assume that task τ_2 computes and writes a new value for *var* which is later read by τ_1. Suppose the statement '`compute var`' in τ_2 is modified in P'. This change may cause τ_2 to block the semaphore for a longer duration, thereby delaying τ_1.

Task execution dependencies arising due to the use of synchronization primitives are transitive in nature and are also symmetric. For a task τ_i, we denote the set of tasks that are execution dependent on it due to access to a shared resource by $ITC_{syn}(\tau_i)$.

Without any loss of generality, in this work we have assumed semaphores are used as synchronization primitives.

4.5. Task Execution Dependency Due to Execution of Interrupt Handlers

Interrupts are profusely used in embedded applications to notify occurrence of events. Interrupts are raised by the sensors or peripheral devices. On receiving an

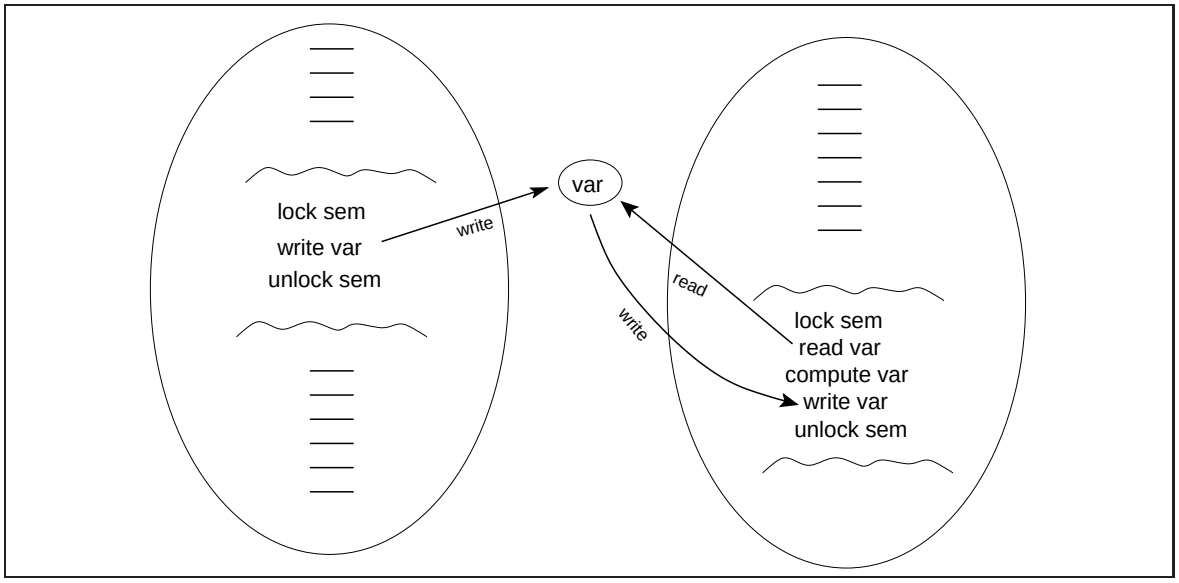


Fig. 4.2: Two tasks communicating using shared memory.

interrupt, the corresponding interrupt service routine (ISR) is invoked which performs operations that are necessary to handle the interrupt. The work performed by an ISR is usually split into two parts: the first-level interrupt handler (FLIH) and the deferred procedure call (DPC) [102, 108]. The role of FLIH is to service the interrupt quickly by executing a few instructions only, and the DPC is executed later. A DPC is scheduled as a normal task in many operating systems such as the Symbian [102].

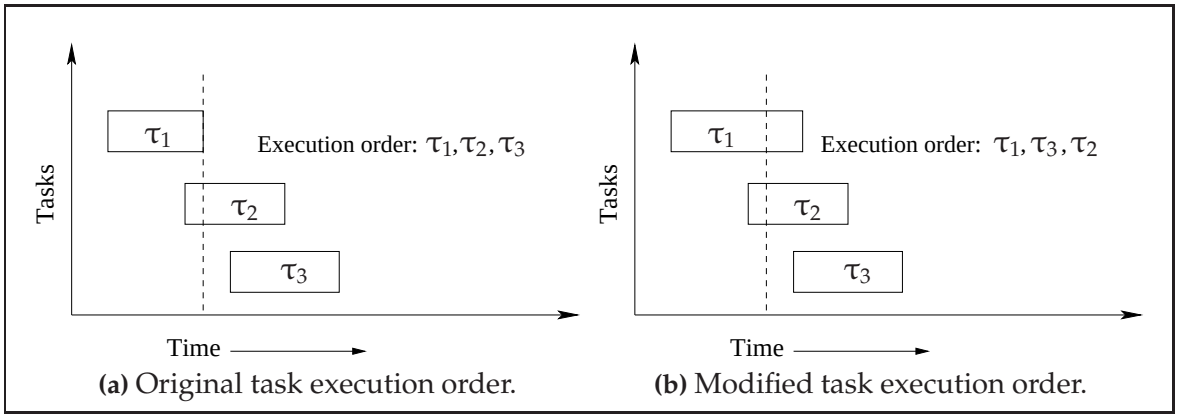
Interrupt handling can cause unpredictable delays (called *jitter*) to the execution of some tasks which are usually unacceptable for hard real-time embedded tasks. In this work, we assume that the execution of a FLIH is fast and the delay can be ignored, and it is only the effect of the DPCs which need to be considered. A DPC usually inherits the priority of the interrupted task, otherwise it is scheduled by the operating system at the priority level of normal tasks. Execution of a DPC can delay other tasks of the same or lower priority.

4.6. A Possible Side-Effect Due to Task Execution Dependencies

The delays suffered by tasks which are execution dependent on some directly modified task can also lead to altered task execution sequences or a different output being produced by the embedded program. We explain a possible side-effect due

to execution dependencies among the tasks in an embedded program with the help of the following example.

Example 4.4: Figure 4.3a shows three tasks and their times of arrival. The relative priorities of the three tasks are as follows: $priority(\tau_3) > priority(\tau_2) > priority(\tau_1)$. The execution order of the tasks in Figure 4.3a are: τ_1, τ_2, τ_3. Let us suppose that task τ_1 is modified and its WCET is increased as shown in Figure 4.3b. The modified task execution sequence now becomes: τ_1, τ_3, τ_2. The modified task execution sequence in P' as a result of the execution dependencies introduced due to task priorities can lead to a different output being produced by P'.



(a) Original task execution order. **(b)** Modified task execution order.

Fig. 4.3: Change in task execution order due to execution dependencies introduced due to task priorities.

4.7. Conclusion

Execution dependencies may arise among tasks of an embedded program due to task precedence ordering, task priorities, inter-task communication, and execution of interrupt handlers. Such dependencies can cause one task to delay another task's execution, change its execution order, etc. In this chapter, we have reported our investigations on the possible reasons why execution dependencies arise among tasks in an embedded program and their effects. These results can help to identify those tasks that can get affected in an embedded program due to code changes to a task.

Chapter 5

SDGC: A Model for RTS of Embedded Programs

CFG and SDG-based representations of programs are extensively being used in diverse applications such as program slicing [66], impact analysis [60], computation of program metrics [122], reverse engineering [27], regression test case selection [15, 94], etc. However, as far as modeling embedded programs is concerned, the semantics of task creation, message passing, semaphore access, timers, etc. are largely ignored by these models and are simplified into ordinary function calls. Moreover, SDG-based representations ignore control flow information. As a result, it becomes difficult to capture important features of embedded programs such as tasks, their execution dependencies, and exception handling. A task is a basic programming entity that is defined by a group of program statements tied together through control flow relations. Furthermore, as advocated by many researchers, analysis of timing properties requires representation of control flow information [47, 123]. However, none of the program models proposed in the literature can meaningfully represent a task or other embedded program constructs that are necessary for RTS. In order to overcome these shortcomings, we propose a graph model to represent embedded programs.

This chapter is organized into sections as follows: In Section 5.1, we discuss in detail our proposed graph representation for modeling embedded programs. We discuss the steps involved in constructing our model from a given embedded program in Section 5.2. We also present an analysis of the space and time complexities of constructing our proposed model in Section 5.3. We conclude the chapter in Section 5.4.

5.1. SDGC Model

Our proposed graph model is an extension of the standard CFG and SDG representations [15]. We have named our model SDGC, SDGC being an acronym for *S*ystem *D*ependence *G*raph with *C*ontrol flow.

Definition: ***SDGC Model*** - An SDGC model for a program P is a directed graph $G = (V, E)$, where V represents the set of nodes and E represents the set of edges. The various types of nodes and edges defined for an SDGC model are represented by the sets $\mathbf{VT_{SDGC}}$ and $\mathbf{ET_{SDGC}}$ respectively, where,

$$\mathbf{VT_{SDGC}} = \{V_{assign}, V_{pred}, V_{call}, V_{A-in}, V_{A-out}, V_{F-in}, V_{F-out}, V_{task}, V_{mp}, V_{sem}, V_{timer}, V_{eh}\}$$

$$\mathbf{ET_{SDGC}} = \{E_{cd}, E_{dd}, E_{ce}, E_{Par-in}, E_{Par-out}, E_{Sum}, E_{cf}, E_{tdef}, E_{prec}, E_{mp}, E_{sem}, E_{timer}\}$$

Since an SDGC model is an extension of a CFG and an SDG model, therefore all node and edge types defined for a CFG and an SDG model are also present in an SDGC model. The sets $\mathbf{VT_{SDGC}}$ and $\mathbf{ET_{SDGC}}$ are supersets of the corresponding node and edge sets discussed for an SDG model in subsection 2.2.5. That is, $\mathbf{VT_{SDG}} \subset \mathbf{VT_{SDGC}}$, and $\mathbf{ET_{SDG}} \subset \mathbf{ET_{SDGC}}$. For simplicity and to aid in understanding, an SDGC model can also be considered to be an *extended union* of the CFG, DDG and the CDG for each procedure in an embedded program.

We now discuss the additional node and edge types that we have introduced in an SDGC model to represent those constructs of an embedded program that are important for RTS.

5.1.1. Additional Node and Edge Types Introduced in an SDGC Model

We have introduced additional node types to represent tasks, message queues, semaphores, timers, and exception handling information. As illustration of the underlying program constructs, for each node type we provide an example of a

related API in VxWorks syntax [127].

Task nodes: We introduce a *task* node type (V_{task}) in an SDGC to model tasks of an embedded program. *Task* nodes are divided into the following sub-types:

- A *task create* node (denoted by V_{tc}) is used to model task creation using a construct such as `taskSpawn()`. The priority of a task is also stored in the *task create* node.
- A *task delay* node (denoted by V_{tdl}) is used to model task delay using a construct such as `taskDelay()`.
- A *task suspend* node (denoted by V_{ts}) is used to model task suspension using constructs such as `taskSuspend()`.
- A *task delete* node (denoted by V_{tdt}) is used to model task deletion using a construct such as `taskDelete()`.
- A *task exit* node (denoted by V_{tx}) is used to model task exit using a construct such as `exit()`.

Message passing nodes: We have introduced a node type called *message passing* (V_{mp}) to model the semantics of message passing. The *message passing* node type is divided into the following sub-types:

- A *message queue create* node (denoted by V_{qc}) is used to model message creation using a construct such as `msgQCreate()`.
- A *message queue send* node (denoted by V_{qs}) is used to model message sending using a construct such as `msgQSend()`.
- A *message queue receive* node (denoted by V_{qr}) is used to model message receiving using a construct such as `msgQReceive()`.
- A *message queue delete* node (denoted by V_{qd}) is used to model message queue deletion using a construct such as `msgQDelete()`.

Semaphore nodes: We have introduced a *semaphore* node type (V_{sem}) to model program statements associated with semaphore operations. The node type *semaphore* is divided into the following sub-types:

- A *semaphore create* node (denoted by V_{sc}) is used to model semaphore creation using a construct such as `semBCreate()`.
- A *semaphore req* node (denoted by V_{st}) is used to model semaphore request using a construct such as `semTake()`.

- A *semaphore rel* node (denoted by V_{sg}) is used to model semaphore release using a construct such as `semGive()`.
- A *semaphore delete* node (denoted by V_{sd}) is used to model deletion of a semaphore variable using a construct such as `semDelete()`.

Timer nodes: We have introduced a *timer* node type to model timer operations. The *timer* node type is divided into the following sub-types:

- A *timer create* node (denoted by V_{tmc}) is used to model timer creation using a construct such as `wdCreate`. Please note that the construct `wdCreate` is used to create a watchdog timer in VxWorks.
- A *timer start* node (denoted by V_{tms}) is used to model the start/setting of a timer using a construct such as `wdStart()`.
- A *timer stop* node (denoted by V_{tmp}) is used to model the reset/stop of a timer using a construct such as `wdCancel()`.
- A *timer delete* node (denoted by V_{tmd}) is used to model the deletion of a timer using a construct such as `wdDelete()`.

Exception handling nodes: Our approach to represent the exception handling information in an SDGC is based on the work reported in [12, 21, 54]. We have introduced the *exception handling* node type (V_{eh}) to model exception handling. The *exception handling* node type is divided into the following sub-types:

- A *try* node (denoted by V_{try}) is used to model the start of an exception block.
- A *catch* node (denoted by V_{catch}) is used to model a `catch` statement.
- A *throw* node (denoted by V_{throw}) is used to model a statement which can raise exceptions.
- A *normal return* node (denoted by V_{nr}) is used to model a return construct during normal execution of the program.
- An *exceptional return* node (denoted by V_{er}) is used to model an abnormal return.
- A *normal exit* node (denoted by V_{np}) is used to model normal termination of a program, i.e., when an exception is not raised.
- An *exceptional exit* node (denoted by V_{xp}) is used to model abnormal termination of a program when an exception is not caught.

We now list the additional edge types that we have introduced in an SDGC over those present in the SDG model.

Control flow edge: Control flow edges (denoted by E_{cf}) in an SDGC are used to model possible flow of control among nodes in the individual functions and tasks in an embedded program.

Task definition edge: A *task definition* edge (denoted by E_{tdef}) is used to connect a node of type V_{tc} to the *Start* node of the CFG for the task.

Task precedence edge: *Task precedence* edges (denoted by E_{prec}) are used to model precedence relations among tasks. A *task precedence* edge connects two nodes of type V_{tc} representing tasks τ_i and τ_j if there exists a predefined precedence ordering between τ_i and τ_j.

Message passing edge: A *message passing* edge (denoted by E_{mp}) is used to represent the execution dependency that arises between a pair of tasks when they communicate using message queues. A *message passing* edge connects a pair of nodes of type V_{qs} and V_{qr} in the sender and the receiver tasks respectively.

Semaphore edge: Dependencies arising due to use of semaphores between two communicating tasks is represented by a *semaphore* edge (denoted by E_{sem}). A *semaphore* edge connects a pair of nodes of type V_{st} and V_{sg} representing the sender and receiver tasks that access the same semaphore variable.

Timer edge: A *timer* edge (denoted by E_{tm}) is used to connect a *timer create* node with the *Start* node of its associated handler function which is invoked when the timer expires. Nodes of types *timer start*, *timer stop* and *timer delete* are connected with the preceding/subsequent nodes of the SDGC model using control flow edges.

Edges to model exceptions: To model changes to the normal control flow due to exceptions, the following control flow edges have been introduced to connect *exception handling* nodes.

- A *try* node is connected to the node representing the first statement in the exception block through a control flow edge.
- A *catch* node has two outgoing control flow edges: the `TRUE` edge connects the *catch* node to the first statement in the `catch` block, and the `FALSE` edge connects the *catch* node to the next `catch` statement if any.
- A *throw* node is connected to the corresponding *catch* node with a control flow edge. If a *throw* node is outside a *try* block, then it is connected to the *exceptional exit* node of the function using a control flow edge.

The `throw` and `catch` statements are treated as conditional statements which alter the flow of control depending on the evaluation of the conditional expression.

Example 5.1: The SDGC model of the program of Figure 2.6 is shown in Figure 5.1. It can be observed that the SDGC model in Figure 5.1 is an extension of the SDG model shown in Figure 2.7, and incorporates additional nodes and edges for modeling embedded program features such as tasks, message queues, etc. The solid edges in Figure 5.1 represent control flow edges for each function in the program. We have omitted the `TRUE` labels on control flow edges wherever the flow of control is obvious to avoid cluttering the figure.

The program in Figure 2.6 has two statements spawning two tasks in lines $S4$ and $S5$. The task names are '`tMonitor`' and '`tVarySpeed`' and the task bodies are the functions `monitor()` and `vary_speed()` respectively. Since task creation semantics are ignored by the SDG model, these constructs are shown as simple function calls in Figure 2.7. In the SDGC model shown in Figure 5.1, the two task creation statements are represented by the *task creation* nodes $S4$ and $S5$. These two nodes are connected to the corresponding function entry nodes $E3$ and $E2$ respectively by using *task definition* edges ($S4 \rightarrow E3$ and $S5 \rightarrow E2$).

The tasks `tMonitor` and `tVarySpeed` in the program shown in Figure 2.6 communicate using a message queue. The lines (and the corresponding nodes) related to message queue management are: $S7$, $S11$, $S22$ and $S25$. Node $S7$ in Figure 5.1 is of type V_{mc}, node $S11$ is of type V_{mr} while nodes $S22$ and $S25$ are of type V_{ms}. Two *message passing* edges connect nodes $S22$ and $S11$ and $S25$ and $S11$ respectively.

5.2. Construction of An SDGC Model

In the following, we discuss the construction of an SDGC model M through a static analysis of the source code of a program. The pseudocode for SDGC model construction has been shown in Algorithm 1. The input to the algorithm is the embedded program for which the corresponding SDGC model is to be constructed. The first step (line number 2 in Algorithm 1) constructs the CFG for each function in the program. This step involves parsing the input program, and constructing the nodes and edges by executing appropriate semantic actions corresponding to each grammar rule. All nodes including the nodes modeling specific embedded program features such as *task create*, *message queue send*, *timer start*, etc. are created in this step. For example, a *task create* node is created when a `taskSpawn()` statement

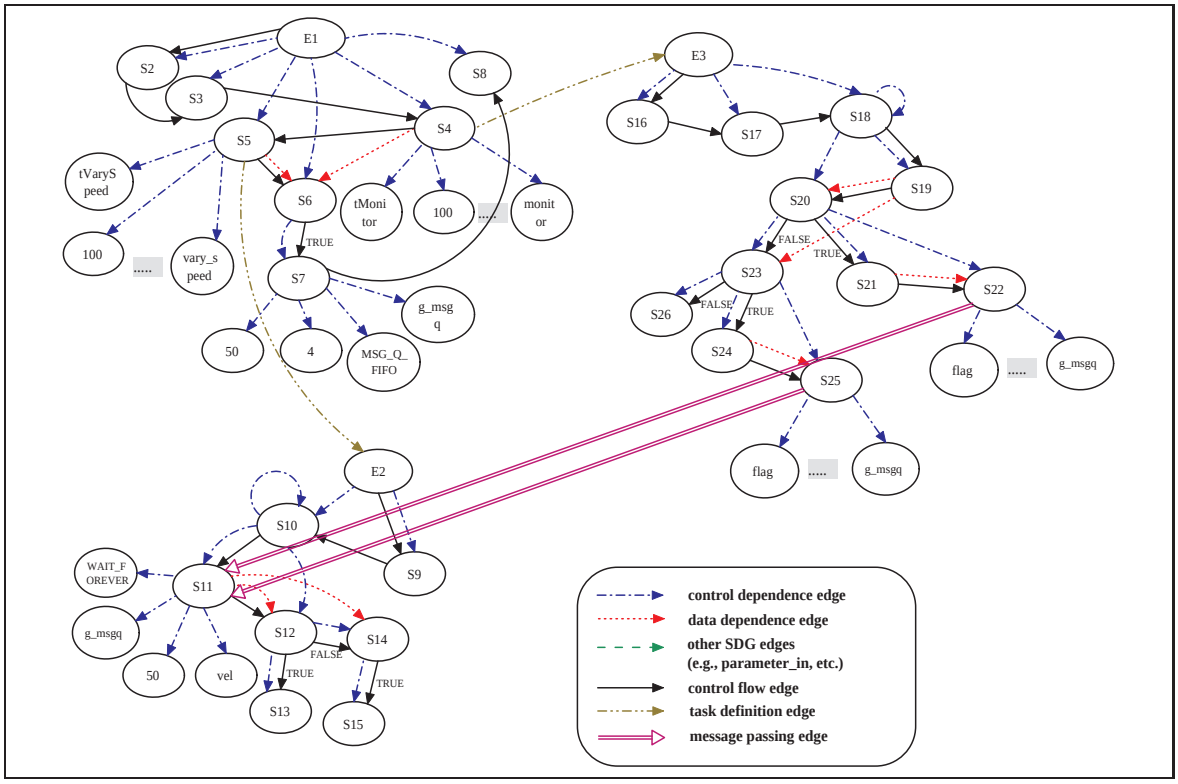


Fig. 5.1: SDGC model for the program of Figure 2.6.

is parsed. Once construction of the individual CFGs is complete, these are analyzed in lines 4 to 7 to compute and construct the data and control dependence edges.

In the next step, the partially constructed SDGC model is traversed to connect nodes of types *task create* and *timer create* to the corresponding functions which are either task definitions or timer expiry handlers using *task definition* edge or *timer* edge respectively. This helps in considering the special semantics of these program constructs by differentiating them from simple function calls. For example for the embedded C program shown in Figure 2.6, the name of the function `monitor` which is the definition for the task `tMonitor` is stored as an *actual-in* argument connected to the *task create* node *S*4. Subsequently, we connect every *message queue send* node to the corresponding *message queue receive* node with a *message passing* edge. The *message queue receive* node corresponding to a *message queue send* node can be identified from the message queue identifier over which the communication takes place. Similarly, *semaphore req* nodes are connected to the corresponding *semaphore rel* nodes using *semaphore* edges. A pair of *semaphore req* and *semaphore rel* nodes can easily be identified based on examination of the name/id of the semaphore variable.

The last step concerns identification of the precedence relationships among

tasks. For this, the source code is analyzed for the constructs `join()`/`wait()`. These two constructs indicate whether a task τ_j precedences another task τ_i. We then create a *task precedence* edge between the *task create* nodes of the two tasks τ_i and τ_j to model the precedence relationship between the two tasks.

Algorithm 1 Pseudocode for constructing SDGC model of an embedded program.

```
1: procedure ConstructSDGC(input)
                                          ▷ input = Input embedded C program
                                ▷ Output of ConstructSDGC is the SDGC model for input
2:     Construct CFG for each procedure in input
   ▷ New nodes such as timer create, task create, and exception handling are constructed in this step.
3:     Connect nodes across CFGs to create edges   ▷ Edges include Call, Parameter-in and
   Parameter-out edges
4:     for each CFG in the partially constructed SDGC model do
5:         Perform data dependence computation to add data dependence edges
6:         Perform control dependence computation to add control dependence edges
7:     end for
8:     for each task create node do
9:         Connect the task create node to the corresponding function definition node with a task
   definition edge
10:    end for
11:    for each timer create node do
12:        Connect the timer create node to the corresponding timer expiry handler with a timer edge
13:    end for
14:    for each message queue send node do
15:        Connect the node to the corresponding message queue receive nodes in all receiver tasks
   with a message passing edge
16:    end for
17:    for each semaphore req node do
18:        Connect the node to the corresponding semaphore rel node in all the other tasks which
   block on the same semaphore variable with a semaphore edge
19:    end for
                                ▷ Checking for precedence constructs join()/wait()
20:    Identify precedence order among tasks from input
21:    Add task precedence edges to the SDGC model
22: end procedure
```

Example 5.2: We now illustrate the construction of the SDGC model for a sample program shown in Figure 5.2. First the CFG for each procedure in the sample program is constructed. During creation of the CFGs, the *actual-in*, *actual-out*, *formal-in*, *formal-out*, *call-site*, *task create*, *timer create*, etc. nodes are also created. The partially constructed SDGC model at the completion of this step is shown in Figure 5.3. Subsequently, the data dependence and control dependence edges are computed and added to the partially constructed SDGC model. The SDGC model after this step is shown in Figure 5.4. Finally, for this example, adding the *task definition*, *timer*, and the *message passing* edges to the partially constructed SDGC

```
D0 WDOG_ID g_wdog = NULL;
D1 int g_cruise = ERROR;
D2 float g_distance;
D3 float g_speed;

E4 int main( void )
{
S5  int l_status = ERROR;
S6  g_wdog = wdCreate();
S7  if (NULL != g_wdog)
    {
S8    l_status = taskSpawn("tCruise",50,0,
         10000,(FUNCPTR)cruise,0, 0, 0, 0, 0,
             0, 0, 0, 0, 0);
  }
S9  return 0;
}

E10 void HandleTimeout(int val)
    {
S11   if ( 1 == val )
      {
S12     /*Reset Cruise Control parameters*/
S13     g_cruise = ERROR;
      }
   }
```

```
E14 void cruise( void )
    {
S15   while(true)
      {
        /*Timer Start*/
S16     l_status = wdStart( g_wdog
                    , TIMEOUT
                    , (FUNCPTR)HandleTimeout
                    , 1 );
S17     if(OK==l_status)
        {
S18       /*Compute Control Parameters:
                g_distance and g_speed*/
S19       if (g_distance > 30)
                  && (g_speed > 40 ))
          {
S20         /*Put into Cruise Control*/
S21         g_cruise = OK;
          }
S22       wdCancel(g_wdog);
          /*Timer End*/
        }
      }
}
```

Fig. 5.2: A sample VxWorks program incorporating usage of a watchdog timer.

model of Figure 5.4 completes construction of the the SDGC model shown in Figure 5.5.

In Figure 5.5, the node $S16$ is of type *timer start*. If the timer expires before it is reset in line $S22$, the control flows to the handler routine `HandleTimeOut`. This is modeled in the SDGC model of Figure 5.5 with a *timer* edge connecting the nodes $S16$ and $E10$.

5.3. Complexity Analysis

We now present an analysis of the space and time complexities of the algorithm for construction of an SDGC model.

Time Complexity: Let us assume that there are n statements in P and let the number of nodes and edges in the SDGC model M be m and e respectively. Let the number of functions in P be denoted by p. From the pseudocode ConstructSDGC presented in Algorithm 1, it can be observed that the primary steps in constructing an SDGC model are: (a) construction of the CFG, (b) computation of data dependence edges, (c) computation of control dependence edges, (d) incorporation of information related to semantics of tasks, message passing, semaphores, and timer

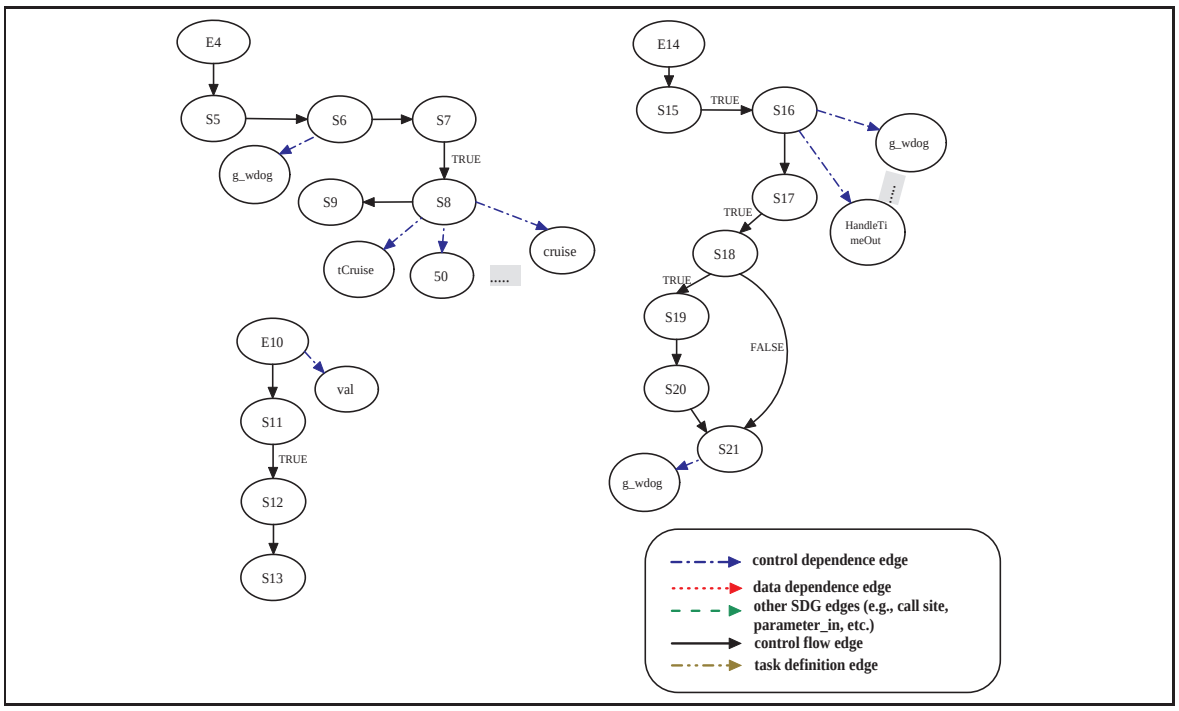


Fig. 5.3: Partially constructed SDGC model for the sample program in Figure 5.2.

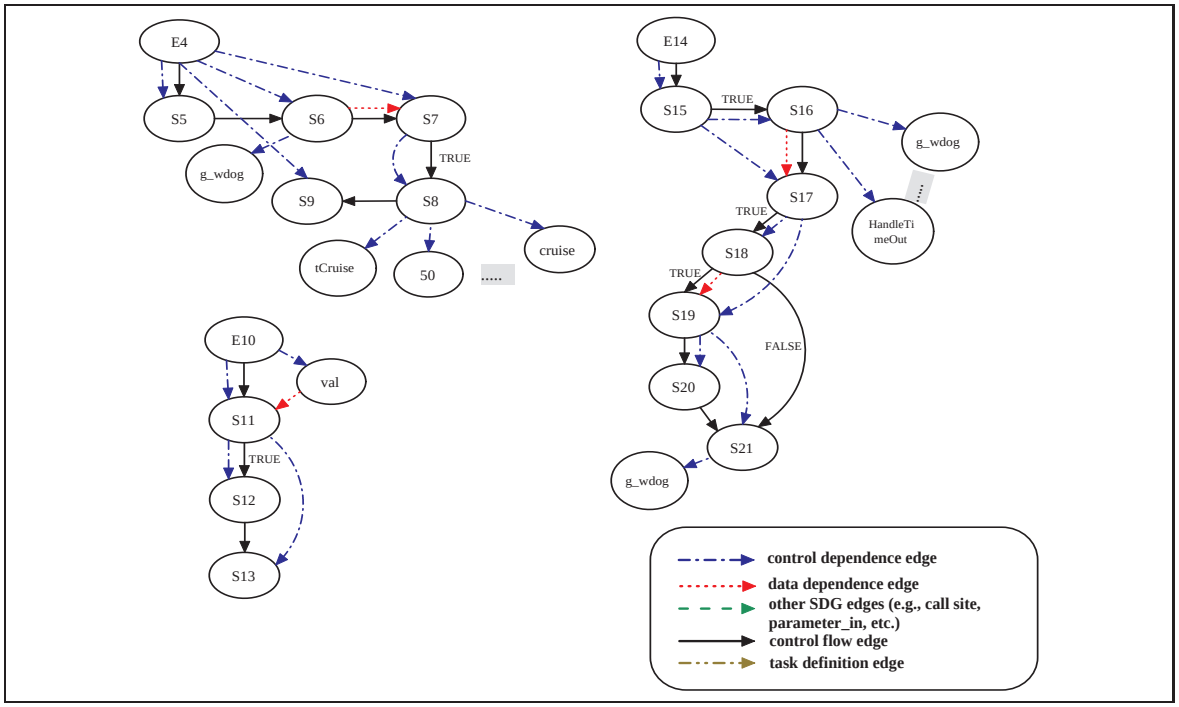


Fig. 5.4: Partially constructed SDGC model after computation of data and control dependences.

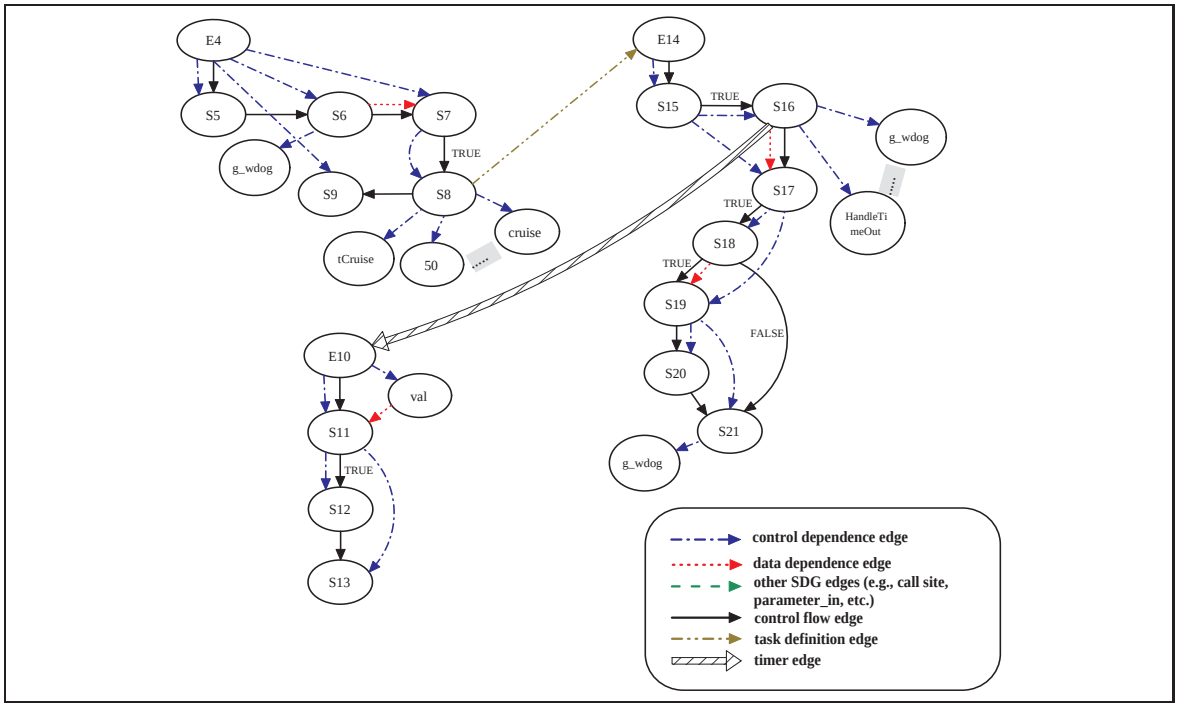


Fig. 5.5: Complete SDGC model for the sample program shown in Figure 5.2.

management. The time complexity of CFG construction is $O(n)$ [10], and the time complexity of control dependence computation is $O(n^2)$ [35]. Computation of the data dependence edges requires traversal of each CFG in the SDGC model and is bounded by $p * O(m^2)$. To create edges of type *semaphore*, *task precedence*, etc. requires traversing the SDGC model. This can be expressed by $O(m^2)$. Therefore, the time complexity of SDGC model construction is $O(m^2)$.

Space Complexity: Let us assume that there are n statements in P, and let n_f denote the number of functions in P (including task, timers, message queue and semaphore creation), and arg_{max} be the maximum number of arguments of any function in P. For a given embedded program, the SDGC model contains more number of edges than the SDG model because the SDGC model captures many additional embedded program features. The space complexity for representing a graph is of the order of $O(n^2)$ where n is the number of nodes in the graph. Therefore, the space requirement for constructing an SDGC model is $O((n + 2 * n_f * arg_{max})^2)$. Assuming that the maximum number of arguments is limited to a constant say 10, the space complexity expression reduces to $O(n^2)$.

5.4. Conclusions

In this chapter, we proposed an intermediate graph representation for embedded programs which we have named as SDGC. An SDGC captures important features of an embedded program such as tasks, task precedences, inter-task communication using message passing or semaphores, and exception handling. Subsequently, we also discussed an algorithm to construct an SDGC model from a given embedded C program, and presented an analysis of the time and space complexities of our proposed SDGC construction algorithm.

Chapter 6

RTSEM: An RTS Technique for Embedded Programs

We have named our proposed RTS technique as RTSEM (*R*egression *T*est *S*election for *EM*bedded programs). RTSEM is based on analyzing the SDGC model of an embedded C program. More specifically, RTSEM selects test cases based on an analysis of control, data and task execution dependencies among tasks. In this chapter, we discuss our proposed RTS technique and a prototype implementation of the same.

This chapter is organized as follows: In Section 6.1, we first list the assumptions made in RTSEM, and then discuss the types of program changes possible across program versions in Section 6.2. We discuss the different processing activities in RTSEM in Section 6.3. We discuss the steps to incrementally update an SDGC model in Section 6.4, and present our regression test selection algorithm in Section 6.5. We discuss about a prototype implementation of RTSEM and present the results obtained during our experimental studies in Section 6.6. In Section 6.7, we compare our approach with related work and conclude the chapter in Section 6.8.

6.1. Assumptions

Our approach is primarily intended to be applicable to small embedded applications. In the following, we list the assumptions made in our RTS technique.

- We assume that the embedded programs adhere to the MISRA C coding guidelines [7]. Although MISRA C guidelines were originally intended for the automotive industry, it is now widely being used for developing embedded

applications. We list a few important rules from [7] to indicate the types of restrictions imposed by MISRA C:

- **Rule 8.1**: Every function must have a prototype declaration.
- **Rule 9.1**: All automatic variables are assigned a value before being used.
- **Rule 12.10**: The comma operator is not used.
- **Rule 12.13**: The increment (++) and decrement (−−) operators are not mixed with other operators in an expression.
- **Rule 14.6**: Only one `break` statement can be used in any iteration construct.
- **Rule 20.4**: Dynamic memory allocation is not used.

We have made two exceptions to the MISRA guidelines in our implementation. MISRA guidelines do not recommend the use of `goto` and `continue` statements to promote structured programming practices. On the other hand, many studies [57, 59, 76] have argued that judicious use of `goto`s is beneficial for many types of common programming problems. Embedded programmers still make heavy use of constructs such as `goto`s. Code that is auto-generated using tools such as Matlab Real-Time Workshop also contain unstructured constructs such as `break`s. In our work, we therefore have assumed that `goto` (**Rule 14.4**) and `continue` (**Rule 14.5**) statements are allowed in the code.

- The tasks are statically created and scheduled, and dynamic creation of tasks is not considered.
- The tasks are assigned priorities statically and are scheduled using a preemptive and priority-driven operating system.
- DPCs are assumed to inherit the priority of the tasks which are interrupted.
- The tasks communicate using either shared memory or message passing mechanisms. We further assume that only the synchronous mode of message passing is used.
- Many embedded applications read inputs from sensors and the output computed is directed to an actuator. The inputs from the sensors are usually transferred to the program as a set of input variables, and the output variables computed are transferred to the actuator. We assume that the initial test suite T is available as a formatted text file. For each test case $t \in T$, the formatted file contains the following information: a unique identifier assigned to each test case, set of inputs, and the expected results.

6.2. Types of Program Changes

A change to a program can be classified into the following three types: (1) *addition* of a statement, (2) *deletion* of a statement, and (3) *modification* to a statement. A change to a program P can be confined to a line or can span multiple lines. A change to P might require addition and deletion of some nodes and edges of the corresponding SDGC model. Since a modification can be considered to be composed of a deletion operation followed by an addition operation, therefore in our work, we assume that only addition and deletion are the basic operations.

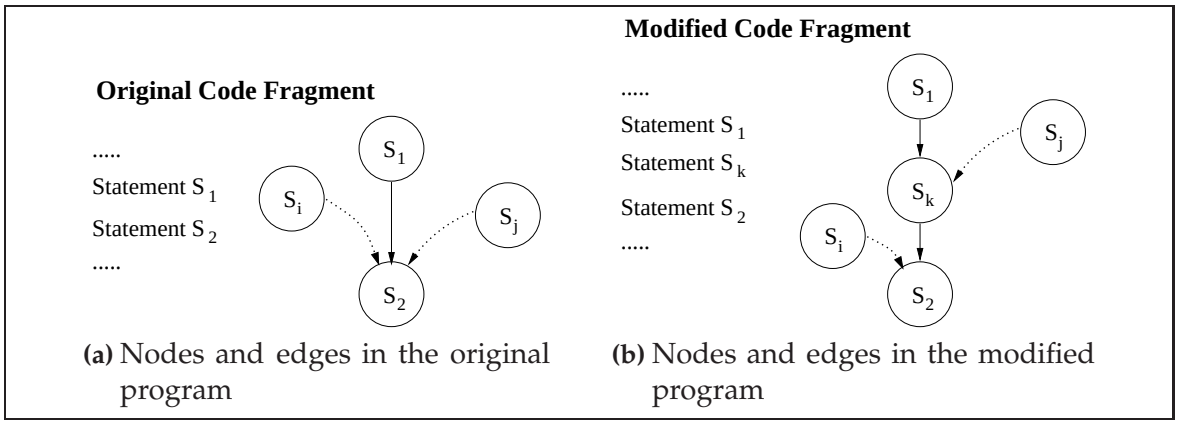
A statement-level change can affect the dependency relations among elements of a program in subtle ways. In the following, we elaborate on how the control flow and dependency relations are affected due to the two basic types of code changes - addition and deletion:

- *Addition of statements* - Adding new statements to P requires creating new nodes and edges in the SDGC model M. Additional control flow, control or data dependence, *parameter-in*, etc. edges may have to be created to model the addition of new statements. It may also be required to delete certain existing control flow and dependency edges during edge creation.
  We explain the effect of addition of statements in Figure 6.1. Figure 6.1a shows a sample code snippet consisting of two sequential program statements, S_1 and S_2. Therefore, in the corresponding partial model, the two nodes (also denoted by S_1 and S_2) are connected using a control flow edge (denoted by solid edges). The partial model also shows that the statement S_2 is data dependent on certain other program statements S_i and S_j (denoted by dotted edges). Now, suppose a statement S_k is added to the sample code snippet as shown in Figure 6.1b. The corresponding snapshot of the modified partial model is also shown in Figure 6.1b. Due to the addition of the statement S_k, the control flow edge between S_1 and S_2 is now deleted, and instead new control flow edges are introduced between the pairs S_1 and S_k, and S_k and S_2. Due to the added statement, the data dependency between S_j and S_2 ceases to exist, and instead a new data dependency is introduced between the nodes S_j and S_k.
- *Deletion of statements* - When a statement is deleted in P', the corresponding node n in M is also deleted. The different edges which were either incident on or emanated from n are also deleted. In addition, new dependency edges

may also be created.

It is important to note that deleting a statement can affect the other dependent parts of the code. For example, if a statement where a variable is defined is deleted, it can lead to a wrong evaluation of a predicate which uses the variable. Therefore, it is important to identify and regression test all those program statements which are data dependent on the deleted statement.

Figure 6.2a shows a code fragment and the corresponding program model consisting of control flow (solid) and data dependence (dotted) edges. The edge E_{ij} models the data dependency between the nodes S_i and S_j in the original code. Let us assume that the variables f and g are defined in the statements represented by the nodes S_b and S_a. Therefore, there exists data dependence edges from S_b to S_j, and from S_a to S_k. Suppose the statement S_j is deleted in the modified version of the program as shown in Figure 6.2b. Because of this change, a control flow (C_{ik}) and a data dependence edge E_{ik} is now created between the nodes S_i and S_k. An additional data dependency is now introduced between S_b and S_k.



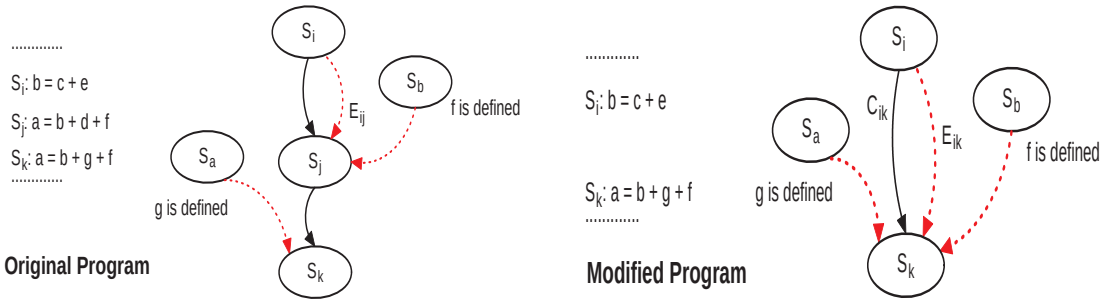
(a) Nodes and edges in the original program

(b) Nodes and edges in the modified program

Fig. 6.1: Effect of addition of statements on control flow and dependencies.

6.3. Processing Activities in RTSEM

Many existing RTS techniques proposed in the context of procedural and object-oriented programs advocate dividing the activities involved in RTS analysis into different phases in an attempt to reduce the run time of the selection process. However, RTSEM has only a single phase since we are restricting the applicability to only small embedded programs. The time complexity of an RTS approach is not



(a) Nodes and edges in the original program

(b) Nodes and edges in the modified program

Fig. 6.2: Effect of deletion on data dependencies.

a critical concern since an automated RTS technique can be used offline for selecting test cases once the modified program version is ready to be regression tested. The primary aim of an effective RTS technique is to reduce the test effort spent by a tester in manually executing the whole initial test suite by omitting those test cases that cannot potentially expose a bug in the program to be regression tested.

The important steps of RTSEM have been represented in the activity model of Figure 6.3. We now briefly describe these processing steps.

1. *Instrument and execute program*: In this step, the original program P is instrumented by inserting print statements. The print statements are inserted only for the purpose of gathering test coverage information. Therefore, even though the print statements may alter the timing properties of tasks, this is immaterial since these statements are stripped during actual regression testing. The instrumented code is executed with the entire test suite T to generate the execution trace for each test case. An execution trace essentially is a sequence of statements of P that is executed by a test case. Generating the test coverage information is an *one-time* activity for a given program during one testing cycle, and need not be repeated during the subsequent regression testing sessions in that cycle. The test coverage information generated in this step is saved for later processing.
2. *Test suite maintenance*: New test cases may be added and obsolete test cases are deleted from the initial test suite T during the maintenance phase. When T is modified, the test coverage information also needs to be appropriately updated. This activity is called test suite maintenance and is carried out during resolution testing. In this work, we only focus on the regression test selection problem and ignore the issues in test suite maintenance. We therefore assume

that the test coverage information generated during resolution testing is in accordance with the initial test suite T.

3. *Construct SDGC model*: In this step, the SDGC model M for the original program P is constructed by using the approach discussed in Section 5.2.
4. *Mark the SDGC model*: In this step, the test coverage information is marked on M. Marking an SDGC model involves adding information to each node in the SDGC model about the test cases that execute the corresponding program statement.
5. *Identify changes*: In this step, the exact changes that were made to the modified program P' are identified through a semantic analysis of P and P'. The statement-level changes between the two files are stored as a formatted ASCII file which we will refer to as the *change* file. Each entry in the *change* file corresponds to a statement-level change between P and P', and contains the changed program statements, the line numbers in both P and P', and the function name corresponding to the change. In this context, it is important to note that any changes due to commenting, formatting, etc. are ignored during semantic analysis.
6. *Update SDGC model*: In this step, the SDGC model is updated using information from the *change* file so as to make it correspond to the modified program P'. The detailed steps to update the original SDGC model M are explained in Section 6.4.
7. *Select test cases*: In this step, the relevant test cases are selected based on control, data and task execution dependency analysis. The details of this step are explained in greater detail in Section 6.5. The selected regression test cases are represented by the datastore *Regression Test Cases* in Figure 6.3.

6.4. Incremental Updation of an SDGC Model

Many existing RTS techniques construct program models of both the original and the modified programs [44, 94]. Construction of two program models during RTS analysis is unacceptably inefficient for large programs, especially when minor changes have been made to P. To overcome this source of inefficiency, we incrementally update the SDGC model M constructed for P once during each maintenance cycle to reflect the changes made to P. In the following, we discuss how incremental updation of an SDGC model is achieved.

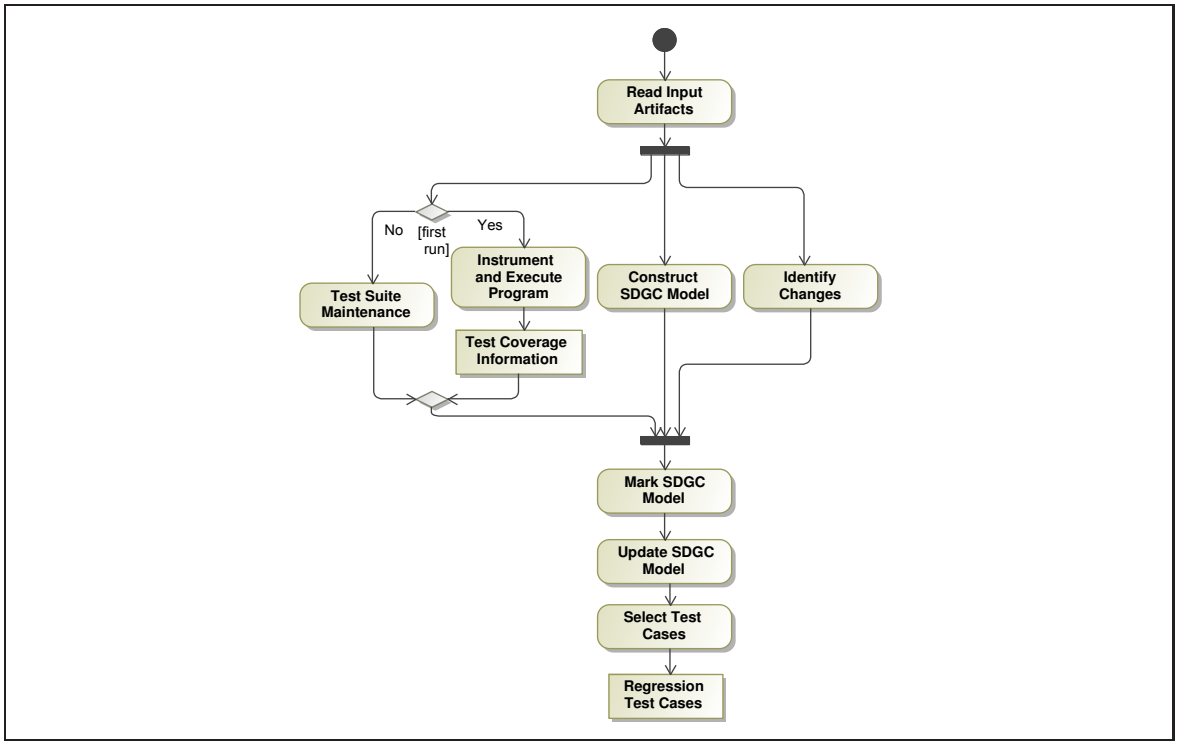


Fig. 6.3: Activity diagram representation of RTSEM.

- *Addition*: We create an additional node for every new statement that has been introduced in P'. Please note that though usually a single node needs to be created for a new statement, at times more than one node may need to be created. For example, for a function call statement, one *call-site* node and one or many *actual-in* and *actual-out* nodes may be created. Control flow edges are created to connect a newly created node to its control flow predecessors and successors nodes in the SDGC model. Creation of a new node in the existing SDGC model usually requires that existing control flow edges between the predecessor and the successor nodes be deleted.
- *Deletion*: For statements which have been deleted from P, we also delete the corresponding nodes from the SDGC model M. Deletion of a node n from M also deletes all the edges (control flow, data dependence, etc.) which are incident on n, or emanate from n. Before deleting a node n, we mark the set of nodes which are dependent on n as *affected*.
- *Dependency computation*: After the control flow information is updated on model M for all the entries of the *change* file, we recompute the data and control dependency information. The data dependency information is recomputed for all the functions in the modified program P' to take into account inter-procedural data dependencies. The control dependence information is

recomputed for only the directly modified functions. Data and control dependency computation for a changed function is done by analyzing the CFGs corresponding to the function.

6.5. Test Case Selection

After a program is changed, apart from selecting test cases based on an analysis of traditional dependence relationships, RTSEM also selects all test cases that execute tasks whose timing behavior may be affected. We can therefore express the set of regression test cases (T_{reg}) selected by RTSEM by the following relationship:

$$T_{reg} = T_{dep} \cup T_{time} \tag{6.1}$$

where T_{dep} and T_{time} denote the test cases selected through control and data dependence analysis and task execution dependency analysis respectively.

6.5.1. RTS Based on Control and Data Dependency Analysis

We select regression test cases based on control and data dependencies using forward slicing. We have proposed an algorithm to compute the SDGC model slice with respect to the statement-level changes between P and P' to identify all the affected model elements. Our forward slicing algorithm is an extension of the two phase SDG slicing algorithm proposed by Bates and Horwitz [15].

We have named our algorithm to slice SDGC models as *SDGCSlice*. The pseudocode of *SDGCSlice* is shown in Algorithm 2. Algorithm *SDGCSlice* takes as input the marked and updated SDGC model M and the *change* file, and computes the set of test cases (T_{dep}) relevant for regression testing. In addition to each entry (i.e., a program statement) in the *change* file, we also include the nodes affected due to deletion of statements as slicing points. Our SDGC model slice computation essentially performs a reachability analysis using data and control dependence edges [49]. Slicing the SDGC model based on data and control dependence edges helps to identify the set of model elements that may get affected due to the modifications. *SDGCSlice* computes the set (denoted by $AffectedSet$) of all nodes affected on account of data and control dependence relations by dynamically updating the set of potentially affected nodes. These steps are shown in lines 5 to 13 in Algorithm 2.

Once all the affected SDGC model elements are identified through forward slicing, the test cases executing those model elements are selected for regression testing. This step is trivial since the information about which test cases execute which model elements is already marked on the SDGC model M.

Algorithm 2 Pseudocode to select regression test cases by computing the SDGC model slice based on control and data dependence.

```
1: procedure SDGCSlice(M, change, T_dep)
                                                  ▷ M = updated marked SDGC model
        ▷ change = modifications between P and P'      ▷ T_dep = selected regression test cases
2:     T_dep ← NULL
3:     Dependent ← NULL                                  ▷ Initially, T_dep is empty
4:     for each entry in change do
5:         if modification type is added OR modification type is modified then
6:             Find the nodes in M corresponding to the statement
7:             for each node m do          ▷ Check for data or control dependence edges from m
8:                 AffectedSet = AffectedSet ∪ {Set of all nodes control or data dependent on m}
       ▷ Slicing should take into account special semantics of function calls related to tasks, timers,
   message queue and semaphore management
9:             end for
10:        else                          ▷ Check for dependencies introduced due to deletions
11:            Find the set of nodes affected due to deletion
             ▷ Check for data or control dependence edges from each node affected due to deletion
12:            AffectedSet = AffectedSet ∪ {Set of all nodes control or data dependent on the affected
   nodes}
13:        end if
14:        if AffectedSet ≠ Φ then
15:            for each node n ∈ AffectedSet do
16:                Add the list of test cases that execute n to T_dep
17:            end for
18:        end if
19:        AffectedSet ← NULL
20:    end for
21: end procedure
```

6.5.2. RTS Based on Task Execution Dependency Analysis

An important part of our regression test selection technique is the selection of those test cases that test execution dependent tasks. We have named our algorithm *TimingSelect*. The pseudocode for the algorithm is shown in Algorithm 3. Algorithm *TimingSelect* takes as input the SDGC model M, and the *change* file, and produces the selected set of test cases (denoted by T_{timing}) as the output. The working of the algorithm is explained briefly in the following: The tasks which are directly modified in P' are first identified from an analysis of the *change* file. *TimingSelect* then invokes four functions: *PrecSelect*, *PrioritySelect*, *MPSelect*, and *SemSelect*,

which compute the set of affected tasks $Succ(\tau_i)$, $Prior(\tau_i)$, $ITC_{mp}(\tau_i)$, and $ITC_{syn}(\tau_i)$ respectively.

The information about the test cases which execute the program statements corresponding to the affected tasks in P' are already stored in the nodes (corresponding to the statements) themselves during marking of the SDGC model M. The test cases executing the affected tasks are selected for regression testing.

Algorithm 3 Pseudocode to select regression test cases based on task execution dependencies.

```
1: procedure TimingSelect(M, change, T_timing)
        ▷ M = updated marked SDGC model   ▷ change = modifications between P and P'
                                          ▷ T_timing = selected regression test cases
2:     T_timing ← NULL
3:     Identify the directly modified tasks from change (denoted by Mod(τ))
4:     for each τ_i ∈ Mod(τ) do
5:         PrecSelect(M, τ_i)   ▷ Select regression test cases based on task precedence order
6:         PrioritySelect(M, τ_i)   ▷ Select regression test cases based on task priorities
                ▷ Select regression test cases based on dependencies due to message passing
7:         MPSelect(M, τ_i)
                ▷ Select regression test cases based on dependencies due to semaphores
8:         SemSelect(M, τ_i)
9:     end for
10: end procedure
11: procedure PrecSelect(M, τ_i)   ▷ Compute Succ(τ_i)
12:     Succ(τ_i) ← NULL
13:     Traverse M to reach task create node of τ_i
14:     for each task precedence edge emanating from τ_i do
15:         Traverse along the task precedence edge to the task create node for the task (denoted by τ_j)
16:         Succ(τ_i) = Succ(τ_i) ∪ τ_j   ▷ Add τ_j to Succ(τ_i)
17:     end for
18:     AddTest(T_timing, Succ(τ_i))
19: end procedure
20: procedure PrioritySelect(M, τ_i)   ▷ Compute Prior(τ_i)
21:     Prior(τ_i) ← NULL
22:     Traverse M to reach task create node of τ_i
23:     Let priority_i be the priority of task τ_i
                ▷ Task priority is stored in the task create node of an SDGC model
24:     Traverse M to find out all task create nodes   ▷ Traversal is along control flow and task
    definition edges
25:     for each task τ_j ∈ M i ≠ j do
26:         if priority_i > priority_j then Prior(τ_i) = Prior(τ_i) ∪ τ_j   ▷ Add τ_j to Prior(τ_i)
27:         end if
28:     end for
29:     AddTest(T_timing, Prior(τ_i))
30: end procedure
```

Algorithm 3 Pseudocode to select regression test cases based on task execution dependencies.

31: **procedure** AddTest(T_{timing}, $AffectedTasks$) ▷ $AffectedTasks$ is a set of affected tasks
32: **for** each $\tau_j \in AffectedTasks$ **do**
33: Add the test cases that execute the task τ_j to T_{timing}
34: **end for**
35: **end procedure**
36: **procedure** MPSelect(M, τ_i) ▷ Compute $ITC_{mp}(\tau_i)$
37: $ITC_{mp}(\tau_i) \leftarrow NULL$
38: Traverse M to reach *task create* node of τ_i
39: Traverse along *task definition* edge to the corresponding task function
40: Traverse the CFG for the function
▷ Check for nodes of type *message queue send* or *message queue receive*
41: **if** node type is V_{ms} OR node type is V_{mr} **then** ▷ Task τ_i is a send/receiver of data
42: Traverse along *message queue* edge to reach the *message passing* node (denoted by n_{dest}) of the other task (denoted by τ_j)
43: Traverse back along the CFG for τ_j starting from n_{dest} to reach the *task create* node for τ_j
44: $ITC_{mp}(\tau_i) = ITC_{mp}(\tau_i) \cup \tau_j$ ▷ Add τ_j to $ITC_{mp}(\tau_i)$
45: **end if**
46: AddTest(T_{timing}, $ITC_{mp}(\tau_i)$)
47: **end procedure**
48: **procedure** SemSelect(M, τ_i) ▷ Compute $ITC_{syn}(\tau_i)$
49: $ITC_{syn}(\tau_i) \leftarrow NULL$
50: Traverse M to reach *task create* node of τ_i
51: Traverse along *task definition* edge to the corresponding task function
52: Traverse the CFG for the function
▷ Check for nodes of type *semaphore req* or *semaphore rel*
53: **if** node type is V_{st} OR node type is V_{sg} **then** ▷ Task τ_i is a send/receiver of data
54: Traverse along *semaphore* edge to reach the *semaphore* node (denoted by n_{dest}) of the other task (denoted by τ_j)
55: Traverse back along the CFG for τ_j starting from n_{dest} to reach the *task create* node for τ_j
56: $ITC_{mp}(\tau_i) = ITC_{syn}(\tau_i) \cup \tau_j$ ▷ Add τ_j to $ITC_{syn}(\tau_i)$
57: **end if**
58: AddTest(T_{timing}, $ITC_{syn}(\tau_i)$)
59: **end procedure**

6.6. Experimental Studies

To study the effectiveness of our approach, we have implemented RTSEM to realize a prototype tool. We have named our prototype implementation `MTest` which stands for *Model-based Test case selector*. In the following, we first briefly discuss our tool implementation. Subsequently, we present the results of the experimental studies that we have conducted using `MTest`.

6.6.1. MTest: A Prototype Implementation of RTSEM

`MTest` has been developed using C++ programming language on a Microsoft Windows 7 environment using a Compaq SG3770IL desktop with a 2.8 GHz processor and 2 GB main memory. The code size of `MTest` is approximately 17 KLOC, excluding the external packages used. `MTest` currently has a rudimentary user interface developed using Microsoft Visual Basic 6.0. During execution, `MTest` takes an original program P, modified program P' and the initial test suite T as inputs. The information related to the initial test suite T is input as a formatted file. The file contains information about the test case identifier, the set of inputs, and the expected output for each test case $t \in T$. The output produced by `MTest` is a formatted text file containing the identifiers of the test cases selected for regression testing.

6.6.1.1. Open Source Software Packages Used in Implementation of MTest

Our implementation of `MTest` utilizes the following open-source software packages: Eclipse [3], MinGW [6], ANTLR [1], Graphviz [5]. In the following, we briefly describe the functionalities of these open-source software.

Eclipse - Eclipse [3] is a multi-language software development environment comprising an IDE and a plugin system to extend it. We have used Eclipse CDT (C/C++ Development Tools) as our IDE for implementation.

MinGW - MinGW (Minimalist GNU for Windows) is a distribution of the GNU Compiler Collection (GCC), and GNU Binutils, for use in the development of native Microsoft Windows applications [6]. We have used MinGW as our C/C++ compiler. It is not required to explicitly configure Eclipse with MinGW, as MinGW is automatically and seamlessly integrated with Eclipse.

ANTLR - ANTLR (ANother Tool for Language Recognition) is a top-down LL(k) parser generator [1]. ANTLR supports generating code in C, C++, Java, Python, C#, and Objective-C. ANTLR takes a grammar file for a language as an input and produces files that recognize and can parse the file. ANTLR mainly produces two files: a lexer and a parser file. These files may be in Java or C or C++. By default, ANTLR produces files in Java. To produce the output files in C++, we need to add the line `language = "Cpp"` in the `options` section of the ANTLR grammar file.

We have used ANTLR v2.7.7 as the parser generator for our implementation.

This is because the ANTLR grammar file [125] for C language that we have used is written with ANTLR v2.7 features. ANTLR plugin can be installed in Eclipse to include ANTLR capabilities in our application. The steps to install the ANTLR plugin in Eclipse is given in [2]. We have adapted the ANTLR grammar file for C language from [125], and have used a subset of the rules in the grammar file for C language which are compliant with MISRA C guidelines [7].

Graphviz - Graphviz (short for Graph Visualization Software) is a package of open source tools distributed by AT&T Research Labs for drawing graphs specified in DOT language scripts [5]. It also provides libraries for software applications to use the tools. We have used Graphviz to graphically display the SDGC models constructed by `MTest`.

6.6.1.2. Components of MTest

The architecture of `MTest` is shown in the component diagram in Figure 6.4. From Figure 6.4, it can be observed that the primary components of `MTest` are: *SDGC Model constructer*, *Test coverage generator*, *Model marker* and *Test case selector*. The ball and socket connections in between the components identifies the producer and consumer components. For example, the test coverage information generated by *test coverage generator* is used by the component *Model marker*. In the following, we describe the roles of the different components of `MTest`.

- *Model constructer*: *Model constructer* implements the algorithm *ConstructSDGC* presented in Section 5.2 for constructing SDGC models. As shown in Figure 6.4, the *Model constructer* component takes P as input and constructs the SDGC model M. *Model constructer* first constructs CFGs for each function of the input program. Once the construction of the CFGs is complete, the iterative dataflow computation technique described in [10] is performed on the CFGs to identify the data dependence edges. *Model constructer* constructs the CDG for a function using the approach proposed by Ferrante *et al.* [35].
- *Test coverage generator*: *Test coverage generator* generates test coverage information when the input program is executed with test cases. Our implementation of *test coverage generator* internally uses Gcov, an open source profiling tool [4]. The generated test coverage information is in the form of an ASCII file listing the functions, tasks and the line numbers covered by each test case in T for the original program P.

- *Model marker*: *Model marker* stores the test coverage information in the SDGC model *M*. A statement in the input files is usually represented by one or more nodes in the SDGC model. For each statement in the input program, the SDGC model *M* is first traversed along control flow edges to find out the corresponding node(s). Then for each node corresponding to a program statement, *model marker* stores the list of test cases that execute the node in the node data structure itself. This technique of storing the test coverage information on the model itself circumvents the use of a database or file storage.
- *Test case selector*: *Test case selector* selects regression test cases by using data, control and task execution dependency analysis as discussed in subsection 6.5. The output of this module is a file containing the identifiers of the test cases selected for regression testing.

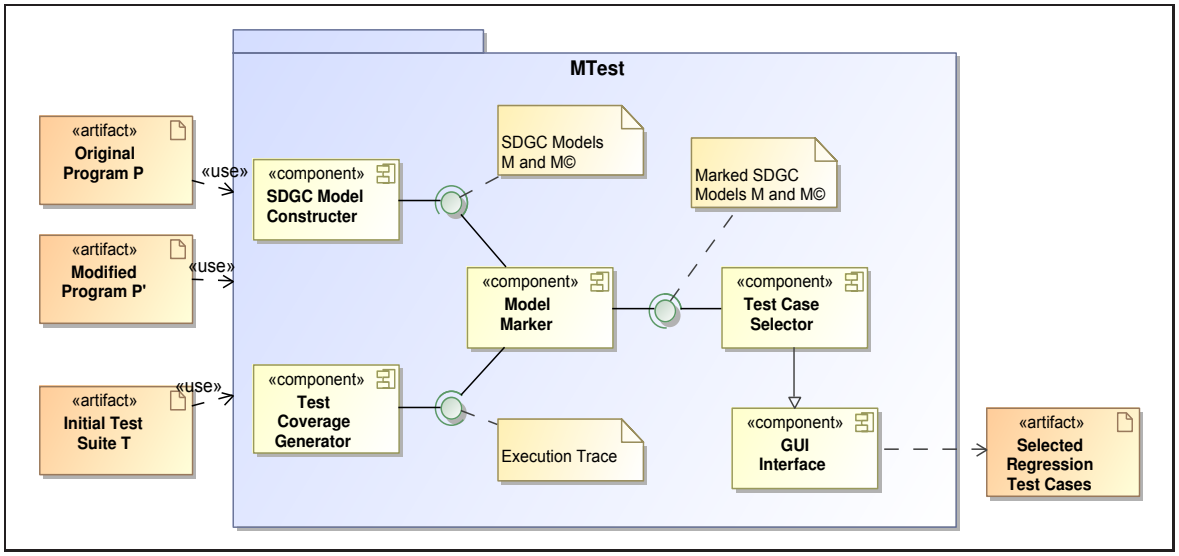


Fig. 6.4: Component model of `MTest`.

6.6.2. An Evaluation of the Effectiveness of MTest

The aim of our experimental studies using `MTest` was to evaluate the performance and effectiveness of our RTS approach RTSEM. We evaluated the effectiveness of our RTS technique based on the following two criteria:

- ***Percentage of test cases selected for RTS*** - This measure indicates the size of the regression test suite as compared to the original test suite. Obviously, it is desirable to have this number as small as possible.

- ***Fault-revealing effectiveness*** - A good RTS technique should select all those test cases that fail when the valid test cases in the initial test suite are run. Thus, the percentage of failed test cases selected by an RTS strategy can serve as a figure of merit. The fault-revealing effectiveness metric can be computed by computing the percentage of test cases selected by an RTS technique from the set of test cases that fail when the valid test cases in the initial test suite are run. That is, the fault-revealing effectiveness of the test suite selected by a safe RTS technique is equal to that of the initial test suite.

6.6.3. Experiments

We have used eight programs from the automotive control domain for our experimental studies. These include applications such as Adaptive cruise controller, Power window controller, and Climate controller. These C programs have been auto-generated from the corresponding Simulink [115] models using the Real-Time Workshop tool in MATLAB. The size of the programs range from 156 to 737 LOC. This size is the number of uncommented statements in the program. Table 6.1 summarizes the average size of the sample programs (LOC) and that of the corresponding SDGC models (in terms of the number of nodes and edges). A snapshot of the SDGC model for the Climate Controller program is shown in Figure 6.5. The figure has been generated with the *dotty* tool of Graphviz [5]. The black edges in the figure represent control flow edges, while the other SDGC model edge types are annotated in the figure. Please note that we have shown only a partial model in Figure 6.5 for the sake of readability.

For each program under test (PUT), we created several modified versions by systematically adding, modifying or deleting one or more lines of code. In order to avoid the possibility of making unrealistic changes to programs, we consulted several industry professionals involved in Simulink/Stateflow (SL/SF) based embedded program development. Based on their feedback on the types of changes usually made, we introduced the following types of modifications to the PUTs: a) changes were introduced in the Simulink model blocks and then the code was auto-generated to get the modified program, b) modifications were made directly to the auto-generated code. The changes made to the Simulink models were reflected as new functions in the source code, or as modified function prototypes, etc. An example of a change of type (a) is disabling the 'Window Up' functionality in one version of the modified Power Window Control program; and that for changes

Program Name	Size (LOC)	Average Size of SDGC Models (#nodes, #edges)
Power Window Controller	204	(263, 334)
Quasilinear Model	174	(245, 312)
Vector Calculator	156	(209, 283)
Cruise Controller	649	(783, 886)
Power Window Controller (with obstacle detection)	737	(852, 996)
ATC Disc Copier	588	(722, 836)
Climate Controller	318	(364, 429)
If Pattern	266	(302, 361)

Tab. 6.1: Characteristics of the programs used in our experimental studies.

of type (b) are changed predicates, changed datatype of variables, and delays to tasks. The relative frequency of occurrence of the different types of changes that we introduced are based on feedback from the industry experts. These are categorized into three levels: *Extremely frequent*, *Frequent*, and *Less frequent*. In Table 6.2 we list the different types of modifications that we introduced in the PUTs and their relative frequencies.

We designed test cases for each PUT to test the functional and temporal correctness of the programs. The functional test cases were designed using blackbox techniques of category partitioning and boundary value analysis, and performance test cases were designed to check whether the timing constraints of tasks are met. The test cases were executed with the original version of the PUTs to generate the test coverage information. The test cases from the initial test suite were also executed with each modified version to find out the number of test cases that failed*. Then, we selected regression test cases using `MTest`. To compare the performance and effectiveness of our approach with an established approach for RTS of procedural programs, we also selected regression test cases using the SDG-based RTS approach proposed by Binkley [18, 19]. We have chosen Binkley's approach [19] since we were unable to find any RTS technique specifically designed for embedded programs, and also we could not find any recent approaches that advance the

* A *failed* test case is one which produces incorrect results when run with the modified program under test.

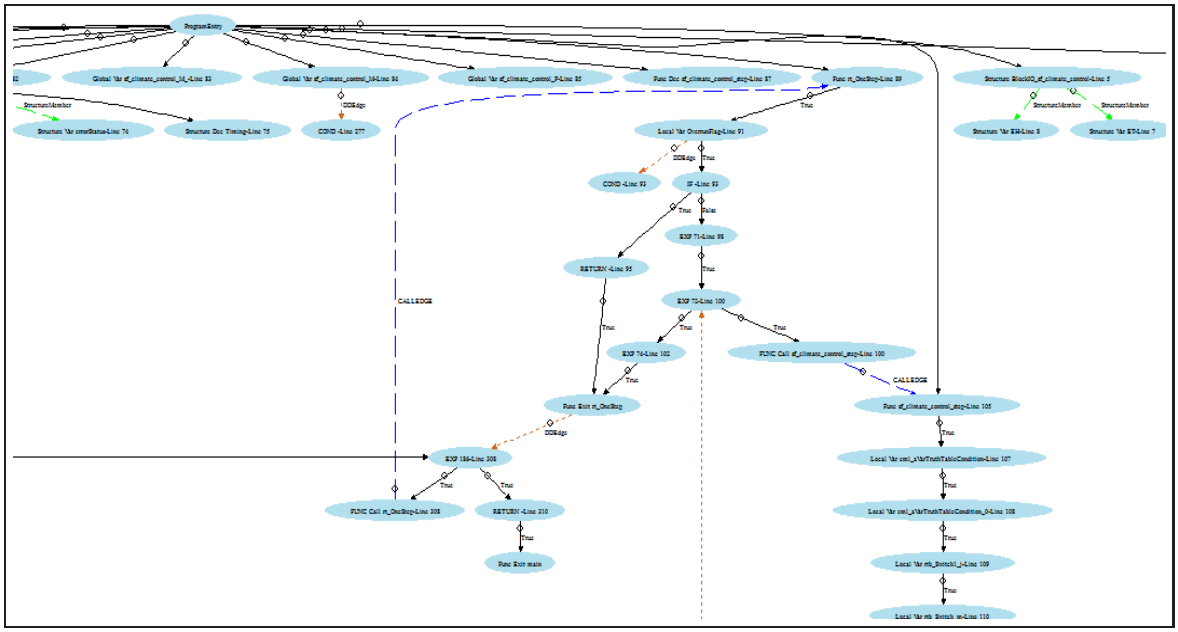


Fig. 6.5: A screenshot of the SDGC model for the climate controller program.

SDG-based RTS approach [19] in non-trivial ways.

6.6.4. Results and Analysis

The results obtained from our experiments have been summarized in Tables 6.3 and 6.4. Table 6.3 shows the number of test cases that were selected for regression testing by `MTest` and Binkley's approach. The first column in Table 6.3 shows the type of programs that were tested. Column 2 shows the number of test cases that were used to test the PUTs. Column 3 shows the number of test cases that were

Type of Change	Relative Frequency
Uninitialized variable declarations	Extremely frequent
Variables assigned wrong values	Extremely frequent
Changed predicates	Extremely frequent
Changed datatypes	Frequent
Changes made to Simulink blocks	Frequent
Changed function prototypes	Less frequent
Task delays	Less frequent

Tab. 6.2: Types of modifications introduced and their relative frequencies.

selected on the average by `MTest` from the initial test suite. Column 4 shows the number of test cases that were selected using Binkley's approach [19]. Column 5 shows the difference in the number of regression test cases selected by the two approaches as a percentage.

The results of Table 6.3 have been presented in the form of a bar graph in Figure 6.6 to visually show the relative performance of `MTest` and Binkley's approach. In the figure, the y-axis shows the percentage of selected test cases while the labels on the x-axis represent the different PUTs. It can be observed from Table 6.3 and Figure 6.6 that `MTest` on an average selected around 45% to 65.22% of test cases for regression testing of the PUTs. For all the PUTs, the number of test cases selected by `MTest` was greater than the SDG-based RTS approach. This increase can be explained by the fact that, in addition to data and control dependence, our approach also selects test cases based on task execution dependencies that are ignored by Binkley's approach. On the average, `MTest` selected 28.33% more test cases than Binkley's approach.

Program Name	**# Test Cases**	**% of Test Cases Selected**		**% Change**
		MTest	**Binkley's Approach**	
Power Window Controller	30	56.67	43.33	30.77
Quasilinear Model	25	48.00	36.00	33.33
Vector Calculator	20	45.00	30.00	50.00
Cruise Controller	42	61.90	47.62	30.00
Power Window Controller (with obstacle detection)	46	65.22	52.17	25.00
ATC Disc Copier	40	57.50	50.00	15.00
Climate Controller	35	54.29	40.00	35.71
If Pattern	30	60.00	46.67	28.57
Total	**268**	**57.46**	**44.78**	**28.33**

Tab. 6.3: Summary of experimental results.

Table 6.4 shows the number of failed test cases that were selected by `MTest` and Binkley's approach. In column 2 in Table 6.4, we list the number of test cases from the initial test suite that failed when run on the modified PUTs. The modifications to the programs were made based on the types of changes listed in Table 6.2. Each

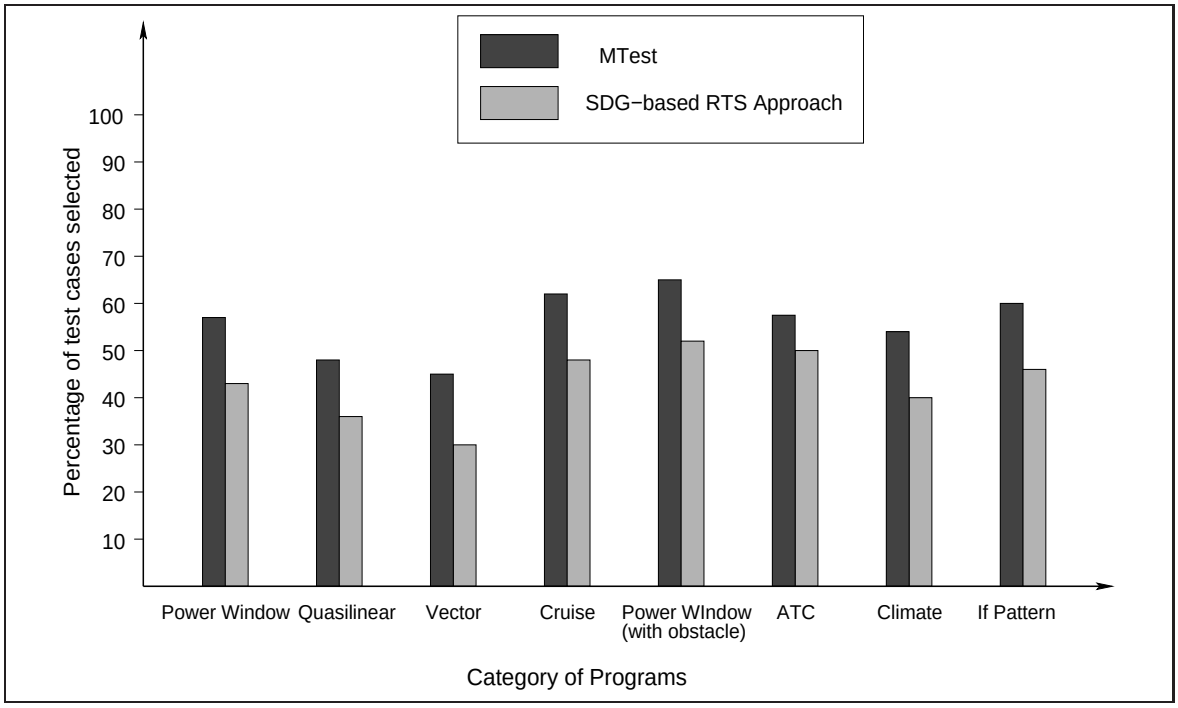


Fig. 6.6: Number of test cases selected for different categories of programs.

time after making a change, regression test cases were selected and the number of regression test cases showing failure was determined. This was done to remove any bias in the results introduced due to selection of a specific type of change. Only the summary data of the average of ten systematically selected changes of a given type have been presented in Table 6.4. For example, for a change of type *predicate change*, randomly selected predicates were changed one at a time. The number of regression test cases showing failure for Binkley's approach and the initial test suite was also determined. The results of Table 6.4 have been presented as a bar graph in Figure 6.7. In the figure, the y-axis shows the percentage of failed test cases selected while the labels on the x-axis represent the different PUTs. The results show that `MTest` is able to select all the fault-revealing test cases present in T. In other words, the regression test suite selected by `MTest` has the same fault-revealing effectiveness as the initial test suite. The fault-revealing effectiveness of Binkley's approach is lower by 36.36% on the average compared to `MTest`.

6.6.5. Threats to Validity

Although the initial results from our limited set of experiments in RTS of embedded programs are very encouraging, it should be noted that are certain limitations which need to be considered before the results can be generalized. We have considered

eight embedded C programs from the automotive domain, and the results that we have obtained during our experimental studies are limited to PUTs of a maximum size of approximately 740 LOC. However, the case studies used are real industry applications. It would be interesting to study the results obtained when `MTest` is applied to select regression test cases for more complex and larger embedded programs having large test suites.

Threats to internal validity can arise due to issues in the implementation of the prototype tool `MTest`. To eliminate issues in our implementation, we have tested the proper working of the different modules in `MTest` with several student programs before carrying out our experimental studies. Moreover, as we discussed in subsection 6.6.4, we have tried to remove any occurrences of bias in our studies by carrying out the tests a few times and averaging the results obtained.

Program Name	% Test Cases Failed	Fault-revealing Effectiveness		% Change
		MTest	Binkley's Approach	
Power Window	23.33	100	57.14	75.00
Quasilinear Model	24.00	100	66.67	50.00
Vector Calculator	25.00	100	80.00	25.00
Cruise Controller	30.95	100	76.92	30.00
Power Window (with obstacle)	30.43	100	78.57	27.27
ATC Disc Copier	27.50	100	72.73	37.50
Climate Controller	28.57	100	70.00	42.86
If Pattern	30.00	100	77.78	28.57

Tab. 6.4: Summary of results of fault-revealing effectiveness.

6.7. Comparison with Related Work

In spite of our best efforts, we could not find any reported results on selecting regression test cases for embedded applications. However, a few results have been reported for regression testing of embedded programs [23, 80]. Cartaxo *et al.* [23] have proposed a technique to select functional test cases for embedded applications. Given the initial test suite, their technique aims to *minimize* the test suite while still

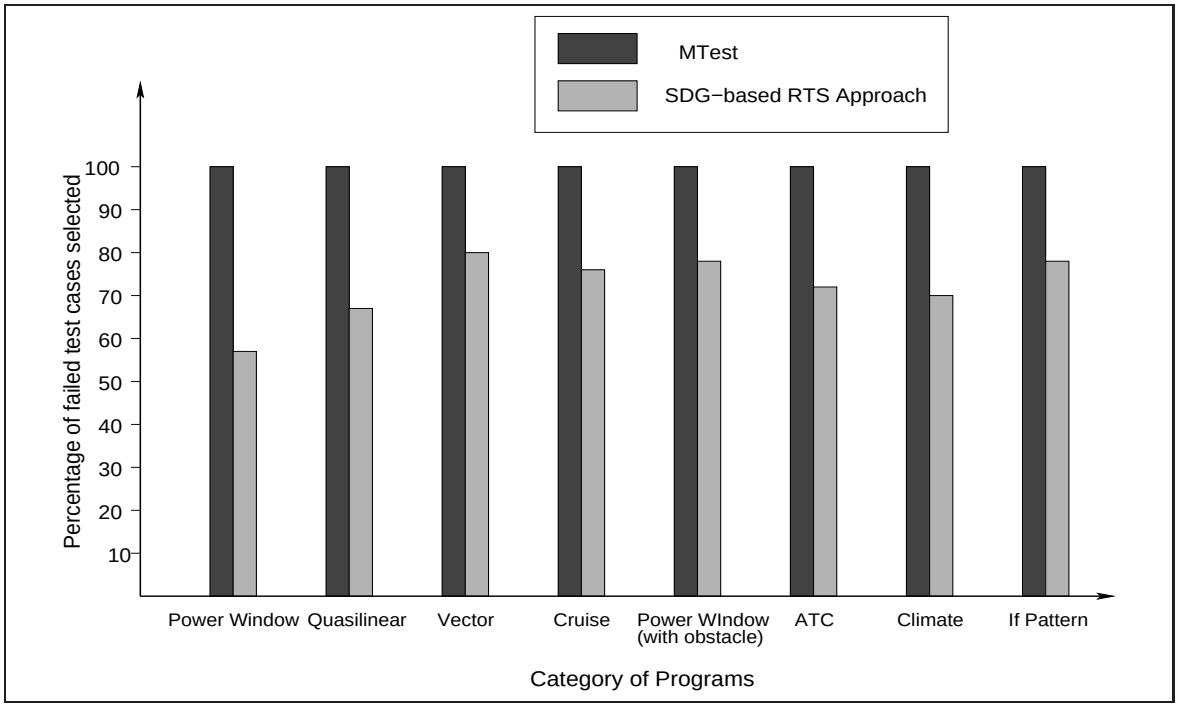


Fig. 6.7: A comparison of the fault-revealing effectiveness of RTSEM and Binkley's approach.

achieving a desired feature coverage. The technique [23], therefore, cannot be directly applied to select regression test cases for embedded programs because the technique does not consider testing of timing errors that are introduced in time-constrained tasks due to execution dependencies. In [80], Netkow and Brylow have proposed a framework called Xest for automating execution of regression test cases in a test-driven development environment. Their test setup helps to automatically execute regression test cases for kernel development projects on embedded hardware. However, their work does not address the problem of RTS, and is therefore, not directly related to our work.

In the absence of any directly comparable work, we compare our technique with a few important procedural RTS techniques that have indirect bearing on our work. Existing procedural RTS techniques [19, 94, 119] select regression test cases based mainly on analysis of either one or more of the following relations among program entities: control flow, control dependence and data dependence. These techniques do not capture important embedded program features such as tasks, timers, message passing, synchronization primitives, task precedence, exceptions, etc. These techniques [19, 94, 119] also ignore task execution dependencies during RTS. As a result, these approaches often omit test cases that can expose time-related

errors, and hence are unsafe. Our RTS technique models tasks, task precedence ordering, task priorities, inter-task communication, timers and exception handling using an extended SDG model. Apart from selecting test cases based on data and control dependencies, our RTS technique also selects test cases based on task execution dependencies that are identified by analyzing relations such as task precedence, task priority, and inter-task communication using message queues and semaphores. During our experimental studies, on the average an additional 28.33% test cases were selected for regression testing and there was an increase of 36.36% in the fault-revealing effectiveness as compared to existing techniques [19].

6.8. Conclusions

Existing RTS techniques do not take into account the implications of many embedded program features such as tasks, task precedences, inter-task communication, timers, etc., and as a consequence do not consider the execution dependencies that arise among tasks because of these features. In addition to data and control dependencies, safe RTS of embedded programs also needs to consider these additional execution dependencies among task elements. In this chapter, we have proposed an RTS technique for embedded programs which addresses the shortcomings of the existing procedural RTS approaches. During our experimental studies we observed an increase in the number of selected regression test cases by approximately 28.33%. Moreover, we observed that on an average 36.36% higher number of fault-revealing test cases that were selected for regression testing. In this context, it is promising to note that in our studies, we observed that RTSEM did not miss out on selecting any of the fault-revealing test cases for regression testing.

Chapter 7

GA-TSO: A Regression Test Suite Optimization Technique

We discussed in Chapter 1 that embedded program features such as tasks, task deadlines, and inter-task communication make testing of these applications a challenging task [104, 112]. A timing error which does not manifest while executing one test case may show up for another test case that has the same set of inputs, same start state but has different timing characteristics. Therefore, an embedded system needs to be tested for both the functional correctness as well as the temporal correctness. This makes it necessary to verify the correctness of an embedded system using a *large* number of test cases.

An important constraint on regression testing is the restriction on the availability of resources such as time, budget, personnel, etc [34, 135]. It is very expensive to execute a large number of test cases during regression testing of embedded programs because the test cases are usually run on specific hardware and require setting up specific execution/simulation environments. Paucity of time is considered to be the primary obstacle faced by testers during regression testing [121]. There are many reasons as to why aggressive schedules are often set for regression testing. A few common reasons for this are:

- Intense competition and increased client expectations force managers to set aggressive product release targets. The time available to carry out activities such as bug fixing, feature enhancements, regression testing, between releases gets severely compromised.
- Recent software development methodologies like extreme programming (XP) and lean promote shorter release cycles [86, 134]. As a result, it is often required to test and release product versions even on a daily basis [121].

An important characteristic of embedded applications that differentiates these from traditional applications is that their functionalities have different degrees of criticality. For example, an anti-lock braking system (ABS) in an automobile performs many activities, e.g., prevents wheel lock under braking (PWB), controls the front and rear wheel brake bias (WBB), etc. An ABS system may also simultaneously carry out data logging activity. For an ABS application, functionalities such as PWB and WBB are considered to be extremely critical as compared to the logging activity. Regression testing of embedded applications should take into account the different criticality levels of the functionalities. This is especially important in a time-constrained environment where regression testing of the more critical functionalities can be given more focus.

A conservative attempt to achieve safety [94] is to select a large number of test cases for regression testing [21, 89]. This of course increases the cost and the time of testing and is usually considered unacceptable. In this context, we propose a multi-objective RTSO technique for embedded applications. Our multi-objective RTSO technique selects a pareto optimal set of test cases that minimizes the cost of regression testing, maximizes the reliability of the frequently-executed non-critical functionalities of the PUT and can be executed within the time allotted for regression testing. The optimization constraints are that the test cases executing the affected tasks and critical functionalities should not be omitted, so as to ensure that the thoroughness of regression testing is not compromised.

This chapter is organized as follows: In Section 7.1, we discuss the regression test suite optimization problem in the context of embedded programs. We present a detailed discussion of our RTSO technique in Section 7.2. We discuss about a prototype implementation and present the results obtained during our experimental studies in Section 7.3. In Section 7.4, we compare our approach with related work and conclude the chapter in Section 7.5.

7.1. Regression Test Suite Optimization for Embedded Programs

We define the problem of regression test suite optimization (RTSO) of embedded applications as follows:

Definition: Let $Time_{max}$ denote the maximum time available for regression testing of a program P using the regression test suite R. Let $\mathcal{F} = \{F_1, \ldots, F_k\}$

represent the set of k objective functions which need to be optimized and let $C = \{C_1, \ldots, C_q\}$ represent the set of q optimization constraints. Then the test suite optimization problem can be stated as follows:

Find the pareto optimal set $S \subseteq R$ having minimum cardinality with respect to $\mathcal{F}$ and satisfying the set of constraints C such that the time to execute P with test cases in S is less than or equal to $Time_{max}$.

7.2. Our Proposed Regression Test Suite Optimization Technique

We have named our GA-based technique for optimizing regression test suites of embedded programs as GA-TSO. In the following, we present a detailed discussion of our technique, and also discuss the importance of each of the optimization objectives and constraints.

7.2.1. Definitions

In the following, we define a few terminologies and concepts related to our work on RTSO that we use in the rest of this thesis.

7.2.1.1. Combinationally Redundant Test Cases

The set of test cases R selected for regression testing P' may contain a test case t_i which does not test any additional program element that is not already tested by the other test cases in R. Such a test case t_i does not contribute to the total path coverage achieved by R. We classify such test cases as a *combinationally redundant* test cases. Although a few reports [24,34,53] have used the term *redundant* to refer to such test cases, we use *combinationally redundant* (CR) to avoid confusion with the term redundant test cases as used in regression testing literature [63].

The concept of CR test cases has been explained with as example shown in Figure 7.1. The oval shape in Figure 7.1 represents an application which is to be regression tested with three test cases T_1, T_2 and T_3. The freeform dashed and dotted lines within the oval represent the execution trace of T_1 and T_2, where A, B, C, D, X and Y are different program elements. It can be observed from the execution trace of T_3 given in Figure 7.1 that test case T_3 executes program elements that have

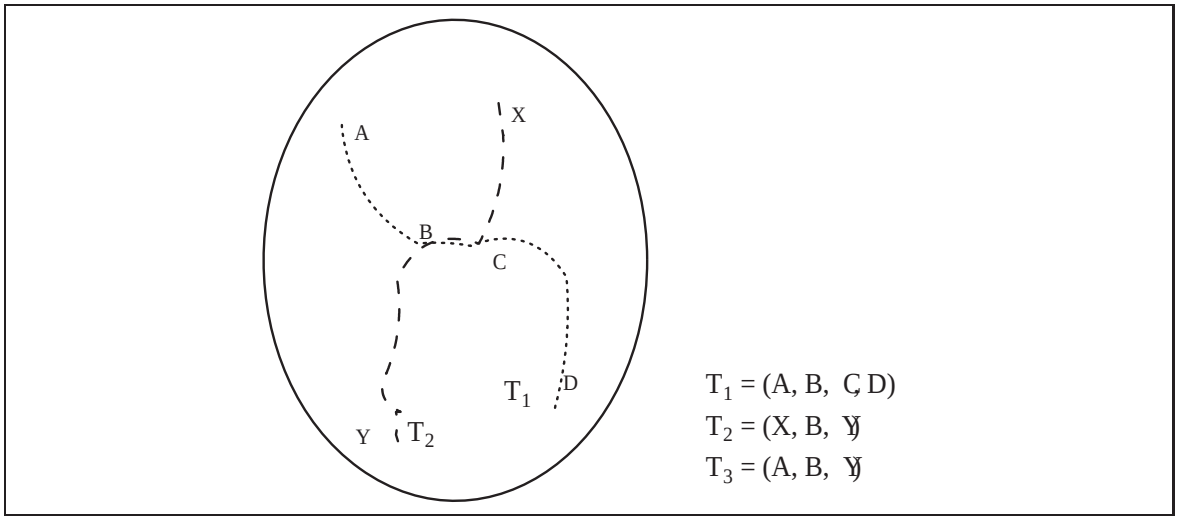


Fig. 7.1: Combinationally redundant test cases.

already been covered at least once by the other test cases. Therefore, T_3 does not contribute to the overall path coverage achieved by the regression test suite and hence can be omitted during optimization especially for a time-constrained regression testing environment. In this context, it is important to note that omitting T_1 instead of T_3 results in a lower overall coverage and, is therefore, not preferable.

7.2.1.2. Observable Reliability of Functions

Let us consider an application A which provides two functionalities: F_a and F_b, of which functionality F_a is executed very frequently while F_b is seldom executed. Suppose that parts of the application A implementing F_a has been thoroughly tested and is bug-free while parts of A implementing F_b contains a bug and is prone to occasional failures. If an user X only uses functionality F_a of the system, then he would conclude that the software is reliable. On the other hand, another user Y may experience failures while invoking functionality F_b and may perceive the system reliability to be relatively poor. Thus, the perceived reliability of a system is observer-dependent and is called its *observable* reliability.

7.2.1.3. Classification of Critical Functionalities

Embedded applications perform different functionalities of varying criticality levels (e.g., ABS application discussed earlier in the chapter). Based on the prevalent industry practices [8, 98], the criticality levels of the different functionalities in an embedded program can be classified as follows:

- ***Extremely Critical (EC)*** - These are functionalities whose failure during execution leads to a failure of the system potentially causing loss of lives, money, etc. An example of an EC functionality is the front and rear wheel brake bias (WBB) in an anti-lock braking system (ABS) in an automobile.
- ***Moderately Critical (MC)*** - Functionalities whose failure cause a system to fail but will not lead to loss of lives, money, etc. are categorized as MC applications. An example of an MC functionality is the diagnostics module in automobiles.
- ***Less Critical (LC)*** - Functionalities whose failures cause only a minor annoyance to the user are classified as LC functionalities. The software in a coffee vending machine is an example of a LC functionality.
- ***Not Critical (NC)*** - Functionalities, such as logging, whose failure does not noticeably hamper the functioning of the embedded system fall under this category.

7.2.1.4. Test Case Costing

The cost of carrying out regression testing of a PUT using a test case (called the cost of a test case) is dependent on the following two factors: cost of test setup, and the cost of executing the test case. Here the underlying assumption is that a different test setup may be required by each test case. Therefore, the cost of a test case t_i, denoted by $cost(t_i)$, is given by

$$cost(t_i) = cost_{setup}(t_i) + cost_{ex}(t_i) \tag{7.1}$$

In the following, we discuss how these two parameters influence the cost of a test case.

1. **Test Setup Cost** - For an embedded application, a major part of the cost of testing using a test case is the cost incurred in setting up the hardware and the environment for the test case to run. For example, consider an automobile adaptive cruise control software that is being tested for regression errors. For testing the adaptive scenario of the adaptive cruise control module, it is required that the system is already in cruise control mode or that the vehicle speed is above a minimum threshold. To execute a test case that checks the proper functionality of the adaptive control software, the system needs to be first put in the cruise control mode. In this context, the cost involved in

executing the step to set the car in cruise control mode is an example of setup cost. We denote the setup cost of a test case t_i by $cost_{setup}(t_i)$.

2. **Test Execution Cost** - Another component of the cost of a test case the time taken by the test case to execute and the inputs required during its execution. We denote the cost of executing a test case t_i by $cost_{ex}(t_i)$.

7.2.2. Optimization Objectives

We have already discussed that regression testing of embedded applications is characterized by high expenses that are incurred and stringent product release deadlines [121]. In this context, presence of CR test cases for testing non-critical functionalities in a test suite need to be eliminated especially under time-constrained regression testing. CR test cases not only increase the size of the regression test suite and the cost incurred in execution, it also does not contribute to the total coverage attained by the test suite.

For higher observable reliability, it is also important to ensure that the more frequently invoked non-critical functionalities of an application are relatively bug-free. This leads to an increased observable reliability of the PUT and helps elicit the confidence of the customer in the product. The functions that actually get invoked when a functionality (or a use case) is executed can be determined by appropriate source code instrumentation. In this context, it can be argued that simply tracking the frequency of invocation of each function during a typical usage of the application can be misleading and inaccurate, e.g, a function getting called iteratively. Therefore, the relative frequency of execution of each function is computed from the operation profile information [78] and by executing the instrumented code. Each function f_i is then assigned a *priority* value ($pv(f_i)$) within the range 1 to 10 to denote its frequency of execution relative to the other functions. Based on the feedback from the testers, we also assign a normalized value in the range of 1 to 10 to each test case to reflect its cost, where higher values indicate a higher cost. In summary, the optimization objectives of GA-TSO are the following:

- $\mathbf{F_1}$: Minimize the number of CR test cases,
- $\mathbf{F_2}$: Minimize the cost of regression testing, and
- $\mathbf{F_3}$: Maximize the observable reliability of the non-critical functionalities.

Optimization of regression test suites of embedded applications also needs to satisfy some constraints. Since embedded applications are being extensively used

in safety-critical and real-time application domains, therefore it is important that test cases that test critical functionalities and affected real-time tasks are not omitted during RTSO. Unless these constraints are met, the thoroughness of regression testing an embedded application using an optimized test suite may be severely compromised. Therefore, the following constraints need to be satisfied by the optimized test suite computed by GA-TSO:

- $\mathbf{C_1}$: The time taken in executing the optimized test suite should meet the given deadline for regression testing.
- $\mathbf{C_2}$: The optimized test suite should not miss out testing any critical functionalities.
- $\mathbf{C_3}$: The optimized test suite should not miss out testing any task whose timing behavior may get be affected.

7.2.2.1. Minimization of CR Test Cases

Our methodology for identifying CR test cases is based on the similarity-based test case selection technique reported by Cartaxo et al. [24]. To identify CR test cases, we first construct an SDGC model M for P. Based on the test coverage information, we mark the edges in M that are covered by each test case $t_i \in R$. While marking M for a test case t_i, we also mark the edges in M connecting those program elements on which the elements executed by t_i are either data or control dependent. This technique helps to take into account all those program elements which are *indirectly* tested by the test case t_i. In the following, we explain how the number of CR test cases in a test suite can be computed.

Let the size of the optimized test suite S be r. We compute a degree of *similarity* for every pair of test cases t_i and t_j as follows: Let $common(t_i, t_j)$ denote the number of edges of M common between the execution traces of t_i and t_j, and $num(t_i)$ denote the number of edges of M that are covered by t_i. We then prepare a similarity matrix SIM of size $r \times r$ where the value of each element SIM_{ij} is computed as:

$$SIM_{ij} = \frac{common(t_i, t_j)}{num(t_i)}, \qquad i \neq j \tag{7.2}$$

$$= 0, \qquad i = j \tag{7.3}$$

After computing $common(t_i, t_j)$, the edges already covered in the execution trace

of t_i are removed before computing $common(t_i, t_{j+1})$. It should be noted that the similarity matrix SIM is symmetric in nature, i.e., $SIM_{ij} = SIM_{ji}, \quad i, j = 1, \ldots, r$, and that the actual values of $SIM_{ij}, \quad i = j$ along the principal diagonal are unimportant. A high value of $SIM_{ij}, i \neq j$, indicates that test cases t_i and t_j execute a large number of common statements. Based on the similarity matrix SIM, a test case t_i is considered to be a CR test case if the following condition is satisfied:

$$\sum_{j} SIM_{ij} = 1, \qquad i \neq j \tag{7.4}$$

Let $numcr(S)$ denote the total number of CR test cases in S. The number of CR test cases appearing in an optimized test suite S should ideally be as low as possible. Therefore, GA-TSO aims to minimize the value of $numcr(S)$.

7.2.2.2. Minimization of the Cost of Regression Testing

The total cost incurred in executing a test suite S, denoted by $cost(S)$, is the sum of the costs of executing the individual test cases of S. Therefore,

$$cost(S) = \sum_{i=1}^{r} cost(t_i) \tag{7.5}$$

$$= \sum_{i=1}^{r} cost_{setup}(t_i) + \sum_{i=1}^{r} cost_{ex}(t_i) \tag{7.6}$$

GA-TSO aims to minimize $cost(S)$, i.e., the cost incurred in regression testing the PUT with the optimized test suite S.

7.2.2.3. Maximization of the Observable Reliability of Non-Critical Functionalities

Let P' consist of m functions. Let $pv(f_j)$ denote the priority value assigned to a function f_j based on the probability of it getting executed during a typical usage of the PUT. For a higher observable reliability of the product, a function f_j having a higher value of $pv(f_j)$ should be tested more thoroughly than a function which is executed less frequently. To capture this information, the fitness function can be defined as follows: Let Q be the total number of times all the functions get tested by S. If a test case t_i tests $numf(t_i)$ functions, then

$$Q = \sum_{t_i \in S} numf(t_i) = \sum_{i=1}^{r} numf(t_i) \tag{7.7}$$

Let the sum of the priority values of all the functions in the program P' be denoted by $PV(P')$. It can be calculated as

$$PV(P') = \sum_{f_j \in P'} pv(f_j) = \sum_{j=1}^{m} pv(f_j) \tag{7.8}$$

The number of test cases in S that should test a function f_j, denoted by $numtest(f_j)$, should be proportional to the value of $pv(f_j)$. Ideally,

$$numtest(f_j) = \frac{pv(f_j) * Q}{PV(P')} \tag{7.9}$$

Let the difference between the actual and the ideal number of test cases executing a function f_j in P' be $\sigma(f_j)$. To ensure that the functions are tested in proportion to $pv(f_j)$, $\sigma(f_j)$ should be as near to zero as possible for an optimized test suite S. The difference between the actual and the expected number of test cases in the optimized test suite S, denoted by $\sigma(S)$, executing the functions is the sum of $\sigma(f_j)$ for each function f_j. Therefore,

$$\sigma(S) = \sum_{j=1}^{m} \sigma(f_j) \tag{7.10}$$

$$= \sum_{j=1}^{m} pv(f_j) * (|numtest(f_j) - count(f_j)|) \tag{7.11}$$

where, $count(f_j)$ is the actual number of test cases in S that test function f_j.

7.2.3. Overview of GA-TSO

The activity diagram in Figure 7.2 shows the important steps in the working of GA-TSO. We now briefly describe the different processing activities that are performed in each step as shown in Figure 7.2.

- *Read input artifacts* - In this step, the original program P, the modified program

P', the regression test suite R, the test coverage information of R, and the operation profile of the PUT are input to GA-TSO.

- *Construct SDGC model* - In this step, the SDGC model M for the program P is constructed. The steps involved in SDGC model construction have been discussed in Section 5.2.
- *Mark SDGC model* - In this step, the test coverage information is marked on M. Marking a SDGC model means adding information about which test cases execute a particular node. This involves storing the identifiers for the test cases which execute a particular statement $s \in P$ in the node $n \in M$ corresponding to s.
- *Identify affected tasks* - In this step, the tasks which are affected due to control, data or execution dependencies that exist among the elements of an embedded program are identified. The steps involved have been discussed in detail in Section 6.5. The test cases which execute the set of potentially affected tasks can easily be identified from the test coverage information. It is important that these test cases are not omitted during regression test suite optimization.
- *Identify critical functionalities* - The functionalities that are critical in the operation of the embedded program to be regression tested can be identified from the SRS and the design documents. It is important that the test cases that execute these critical functionalities are not omitted during regression test suite optimization.
- *Apply optimization algorithm* - In this step, a GA-based optimization algorithm is applied to compute the pareto optimal set of optimal test suites with respect to the objectives and the constraints listed in Section 7.2.2. A random solution is picked from the pareto optimal set as the optimized regression test suite. The optimized set of regression test cases is denoted in Figure 7.2 by the datastore *Optimized Test Suite*.

7.3. Experimental Studies

Our optimization technique simultaneously optimizes multiple objectives which can be conflicting in nature. For example, a test case t_i which executes more critical code components may take more time and can be more expensive to execute than another test case t_j. We have chosen genetic algorithms (GA) [38] to implement the multi-objective RTSO problem, since evolutionary algorithms like GA are

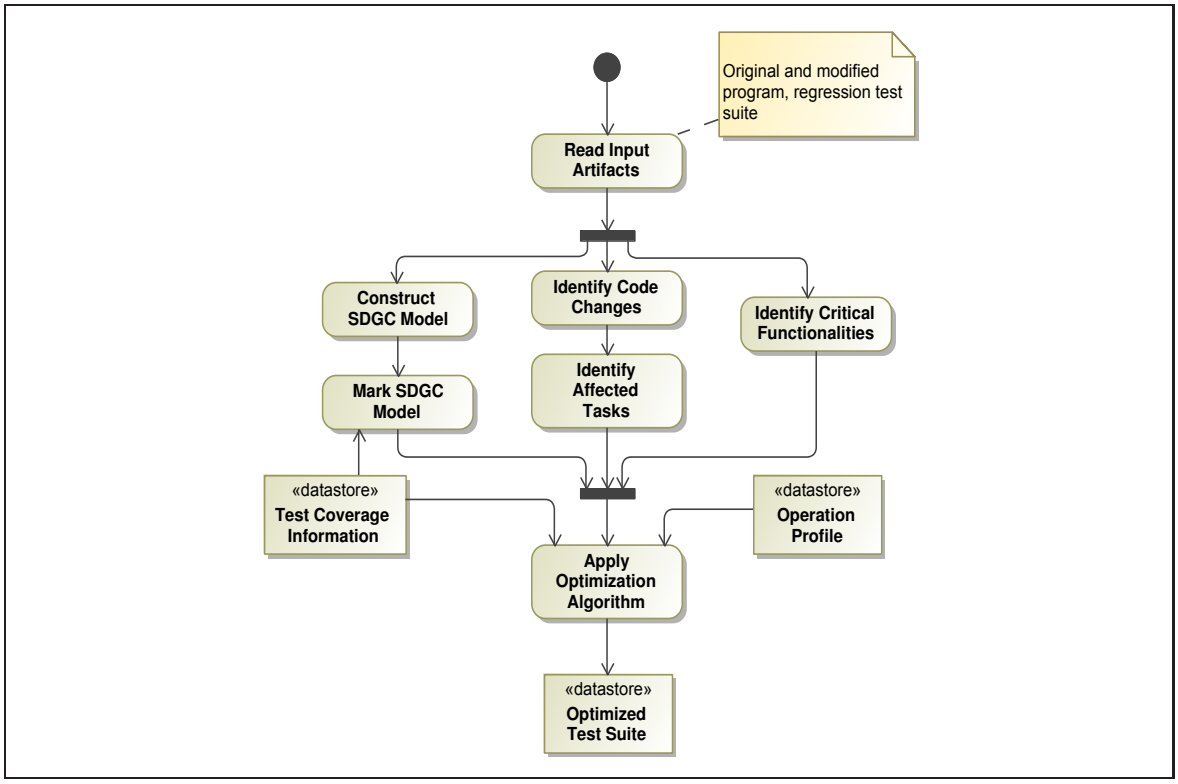


Fig. 7.2: Activity model of GA-TSO.

widely used as the preferred choice for solving multi-objective optimization (MOO) problems as compared to deterministic techniques like linear programming and gradient methods [58, 105].

To study the effectiveness of our approach, we have implemented GA-TSO to realize a prototype tool. We have named our prototype tool `TSOTool` which stands for *T*est *S*uite *O*ptimization *Tool*. In the following, we first discuss our prototype implementation `TSOTool`, and then discuss the results obtained from the experimental studies performed with `TSOTool`.

7.3.1. TSOTool: A Prototype Implementation of GA-TSO

`TSOTool` has been developed as a Windows console application using C programming language. The pseudocode of our GA-based implementation is shown in Algorithm 4. `TSOTool` takes as input the regression test suite R that is to be optimized, the test coverage information, the set of objective functions ($\mathcal{F}$), the set of constraints (C), population size (q), chromosome size (cr), maximum number of iterations (MAX_RUNS), crossover probability (p_c), and the mutation probability (p_m). In our implementation of `TSOTool`, we have assumed that the information

regarding critical functionalities and the potentially affected test cases of the embedded program to be regression tested are available as inputs. `TSOTool` first uses the test coverage information to identify test cases (denoted by TC) that execute the critical functionalities and the affected tasks. Our algorithm then creates an initial population denoted by P_{count}, with *count* set to zero, of size q where the size of each chromosome is cr. The initial population is created such that all the test cases in TC are encoded in the population. Then, the value of the fitness functions are computed for each chromosome in the population. The population for the next generation is created by applying the selection, crossover and the mutation operators on the individuals. The process is repeated for *MAX_RUNS* iterations where the value of *MAX_RUNS* is fixed heuristically. `TSOTool` ensures that the information regarding the test cases in TC are carried over across generations so as to satisfy the set of constraints C.

At the end of *MAX_RUNS* iterations, the population will contain the best possible chromosomes. `TSOTool` chooses a random individual S from the pareto optimal solution as the output.

In the following, we discuss some important issues in the implementation of `TSOTool`.

Encoding Scheme: A chromosome is encoded as $t_1 t_2 t_3 \ldots t_{cr}$ where each constituent test case $t_i,\ i = 1, \ldots cr$, belongs to the regression test suite R.

Selection: We use an elitist selection model [38] so that the best values from every generation gets carried over to the next. In this approach, the probability of an individual getting selected is determined using the rank-based selection technique [36]. Determination of the other members for the next generation is done using a roulette wheel selection technique [38].

Crossover: Two chromosomes are selected at random from a generation for participating in crossover. In our technique, p_c is assumed to be one. We consider a single-point crossover and a random point of crossover is chosen. Application of crossover may cause the same test case to appear twice in a particular chromosome. However, executing a test case multiple times usually does not expose additional bugs. Hence, we replace any one of the duplicate test cases (if any) with another randomly selected test case which is not already present in the concerned chromosome.

Algorithm 4 TSOTool Pseudocode

```
1: procedure TSOTool(q, r, R, F, C, p_c, p_m, S)
      ▷ q = Population Size  ▷ r = Chromosome Size  ▷ R = Regression Test Suite  ▷ F = Set of Objective Functions
   ▷ C = Set of Constraints ▷ MAX_RUNS = Maximum Iterations ▷ p_c = Crossover Probability ▷ p_m = Mutation Probability
      ▷ S = Output is the Optimized Test Suite  ▷ Information about critical functionalities and affected tasks is available
2:    Identify the set of test cases TC which execute the critical functionalities and affected tasks
3:    count ← 0                                              ▷ Keep track of the number of iterations
4:    Create initial population P_count of size q randomly from R where each chromosome is of size r
                                                             ▷ P_i is population at the i^th iteration
5:    while count <= MAX_RUNS do
6:       for chromosome in P_count do
7:          computeFitness() for all functions in F          ▷ Compute the fitness of a chromosome
8:       end for
9:       Copy the best individual to P_count+1
                                  ▷ Ensure that test cases in TC are carried across generations
10:      Apply selection to individuals from P_count
11:      Apply crossover to individuals from P_count
12:      Apply mutation to individuals fromP_count
13:      Generate P_count+1
14:      count ← count + 1
15:   end while
16:   Identify the pareto optimal set S from the last population
17:   Choose a random individual from the pareto optimal set S
18: end procedure
```

Figure 7.3 shows the application of the crossover operator on two sample chromosomes. In this case, the crossover point is taken to be two, and the test cases after the crossover point are swapped to produce a new pair of chromosomes.

Mutation: The mutation operator mutates parts of a chromosome in order to increase the exploration of the solution space. It is generally recommended that p_m should be kept low so that the good solutions are not disturbed. We have chosen p_m to be 0.05. When a particular test case t_j in a chromosome is chosen for mutation, another test case t_j not already included in the chromosome is randomly selected to replace t_i.

Stopping Criteria: We use a heuristic-based stopping criterion. Our heuristic depends on the improvement in the fitness values achieved across a number of generations. Any non-dominated individual [36,38] in the population after termination of the algorithm is a potential solution to our optimization problem.

7.3.2. Results

In the following, we discuss the results obtained from the experimental studies performed using `TSOTool`. For our experimental studies on RTSO, we have used five programs out of the eight with which we carried out our studies on RTSEM. We did not include the programs Power Window Controller, Quasilinear, and

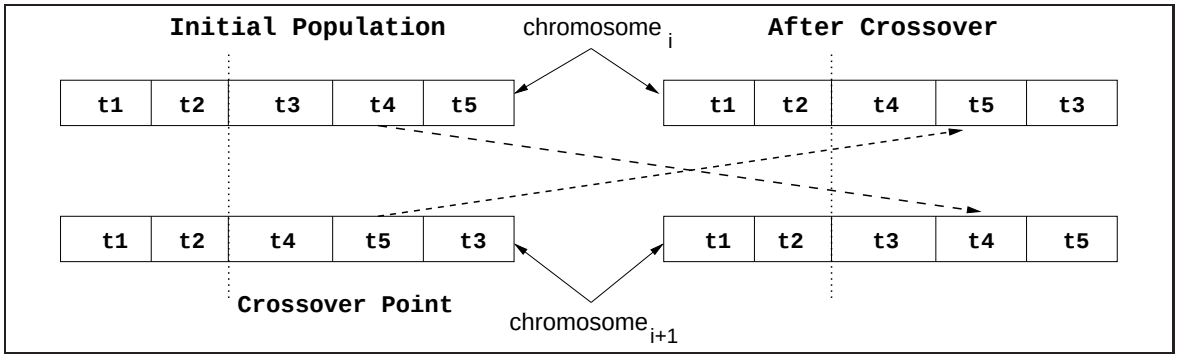


Fig. 7.3: Crossover operation.

Vector Calculator the since the size of the selected regression test suites were too small to be meaningfully optimized. The prototype tool was used to optimize the regression test suite for each PUT. The results of our experimental studies have been summarized in Table 7.1. Column 1 in Table 7.1 shows the names of the PUTs and column 2 stands for the size of the regression test suite for each PUT. Each PUT was then tested with the optimized regression test suite computed by `TSOTool`. Column 3 shows the number of *fault-revealing* test cases from the initial test suite that were included in the test suite optimized using `TSOTool`. Column 4 shows the cost that is incurred by executing the set of optimized regression test cases on each PUT. The next two columns 5 and 6 show the corresponding values when the regression test suites were optimized only with respect to the cost of regression testing. Column 7 in Table 7.1 shows the percent improvement in selection of fault-revealing test cases, while column 8 shows the percent increase in regression testing cost that is incurred while executing the test suites optimized with `TSOTool`.

From Table 7.1, we can observe that a greater number (i.e., percentage) of fault-revealing test cases are included during optimization with `TSOTool`. It should be noted that a higher number of fault-revealing test cases in the optimized test suite is preferred since it ensures that the fault detection effectiveness of the test suite is not compromised. However, the overall cost of regression testing using the test suite optimized using `TSOTool` is comparatively more than the test suite optimized based on test case cost. This can be explained by the fact that usually the test cases that test the more critical functionalities are more expensive to execute. What is more important to note is that there is a significant increase in the number of fault-revealing test cases that are included by `TSOTool` for regression testing. This is because `TSOTool` does not omit any test cases that execute the affected tasks and the critical parts of the code which is consistent with the objective of GA-TSO.

PUT	Size of Regression Test Suite	Size of Optimized Test Suite	Optimized by GA-TSO		Optimized w.r.t Cost		% Change in Fault-revealing Test Cases	% Change in Cost
			# Fault-revealing Test Cases	Cost	# Fault-revealing Test Cases	Cost		
Power Window Controller (with obstacle detection)	30	20	14	170	9	114	55.56	32.74
Cruise Controller	26	20	13	157	9	119	44.44	24.20
ATC Disc Copier	23	15	11	123	7	80	57.14	34.96
Climate Controller	19	15	10	111	8	94	25.00	15.32
If Patern	18	15	9	114	7	97	28.57	14.91

Tab. 7.1: Summary of experimental results carried out with a prototype implementation of GA-TSO.

The limitation in testing a program with a test suite optimized with only a single objective can easily be observed from Table 7.1. For example, the test suite optimized with respect to only cost achieves greater savings in terms of the cost, but omits many fault-revealing test cases. For the PUT ATC Disc Copier, this difference was as high as 57.14%. So although there is a savings in terms of the cost of regression testing that is incurred, it is offset by the fact that there is a lesser chance of faults being exposed using the test suite optimized only with respect to the cost of regression testing. This is unacceptable especially for safety-critical embedded applications, where occurrence of a fault can lead to irreversible damage.

The pareto optimal solutions obtained at the end of the last population for the five PUTs of Table 7.1 are shown in Figures 7.4 and 7.5. The x- and y-axis in the figures represent the cost of regression testing $cost(S)$ and the value of $\sigma(S)$ respectively. Note that the results plotted in Figures 7.4 and 7.5 are with respect to the objectives $\mathbf{F_2}$ and $\mathbf{F_3}$.

7.4. Comparison with Related Work

Most RTSO techniques reported in the literature have been proposed in the context of traditional programs. In spite of our best efforts, we could not find any study

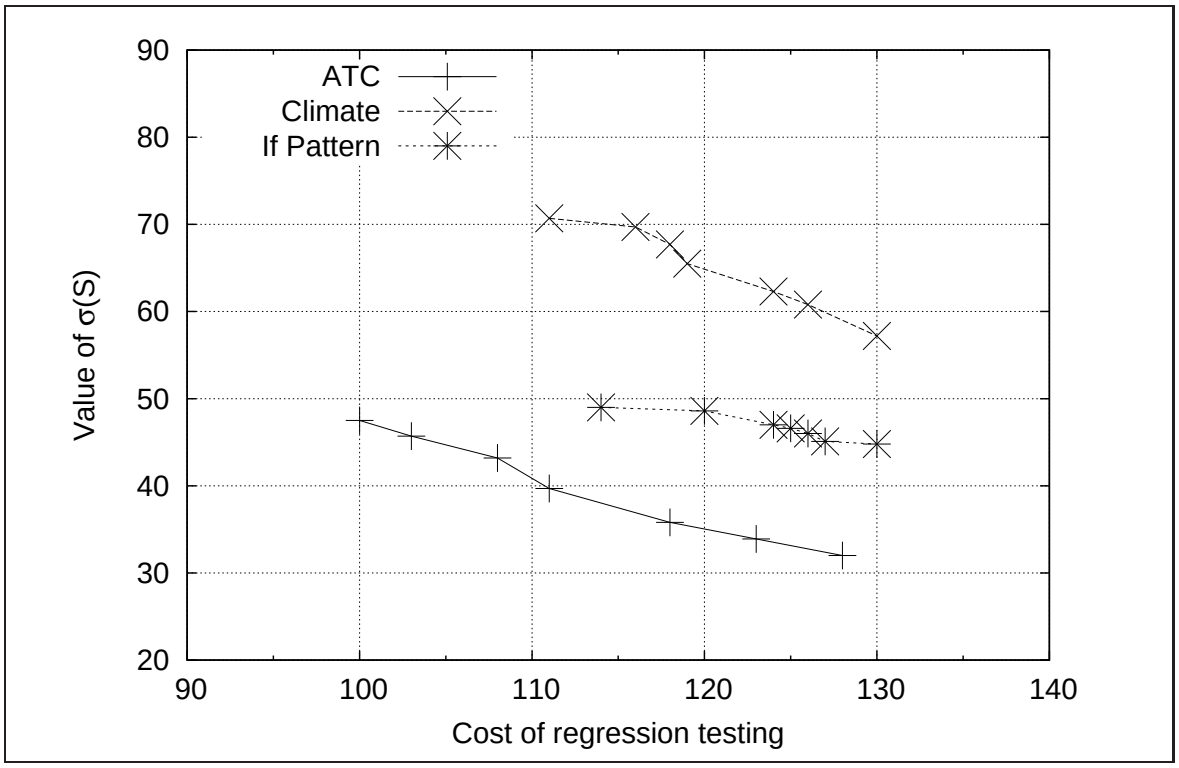


Fig. 7.4: Pareto frontier obtained for the PUTs.

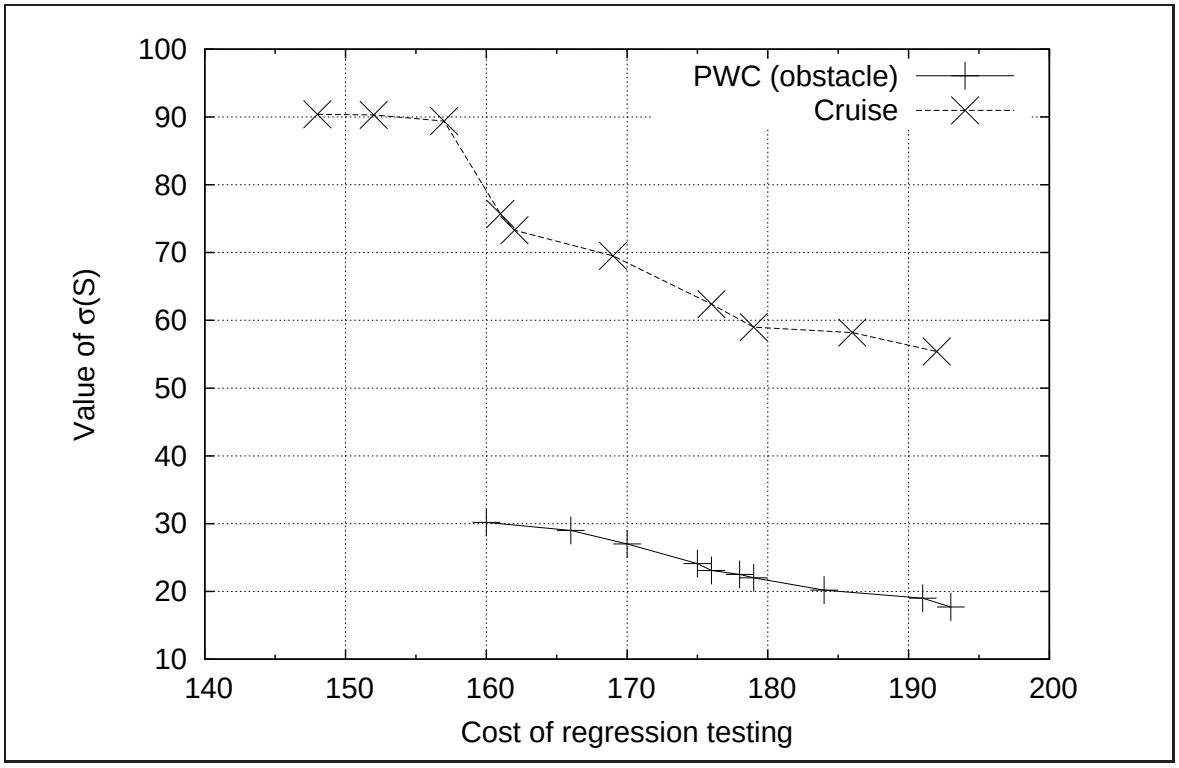


Fig. 7.5: Pareto frontier obtained for the PUTs.

which specifically addresses the problem of optimizing regression test suites of embedded applications. In the absence of any directly comparable work, we compare our RTSO technique with the approaches reported in [34, 135].

Farooq and Lam [34] have proposed a non-pareto min-max based TSO technique which removes redundant test cases and maximizes branch coverage. Zhang et al. [135] have proposed a resource-aware RTSO technique. Their technique [135] first selects test cases which meet the resource constraints, and then the selected test cases are prioritized. However, both these techniques ignore important embedded program features such as tasks, task deadlines, criticality, etc. The event-driven nature of embedded programs and features such as task deadlines, criticality give rise to additional dependencies among the program elements which need to be regression tested. Therefore, these techniques may omit test cases that could potentially expose faults in the PUT thereby compromising the thoroughness of regression testing embedded applications.

7.5. Conclusion

Traditional RTSO techniques cannot be applied to satisfactorily optimize regression test suites for embedded programs without sacrificing the thoroughness of testing since these techniques do not consider testing of critical functionalities and the affected tasks in an embedded program. In this chapter, we have presented a GA-based multi-objective regression test suite optimization technique for embedded programs. Our RTSO technique aims to maximize the observable reliability of the frequently-executed non-critical functionalities, minimize the cost of regression testing, and minimize the number of CR test cases in the regression test suite. Our experimental studies show that the thoroughness of regression testing is not compromised since all the critical and frequently-used functionalities and the potentially affected tasks are thoroughly tested within the regression testing deadline.

Chapter 8

Conclusions and Future Work

The main goal of this thesis was to present improved regression testing approaches for embedded programs. In this thesis, we have presented a novel regression test selection and optimization approach for embedded programs. We have implemented prototype tools to realize our proposed approaches, and have presented experimental evaluations of our approaches.

This chapter is organized as follows: In Section 8.1, we highlight the main contributions of this thesis. In Section 8.2, we discuss possible extensions to our work.

8.1. Summary of Contributions

In the following, we elaborate the three main contributions of this thesis.

8.1.1. SDGC Model

The models proposed in the literature for use in RTS ignore many important features of embedded programs such as tasks, task precedence orders, timing constraints, inter-task communication using message queues and semaphores, interrupts and exception handling though these are aspects that need to be considered during regression test selection. To overcome this shortcoming, we have proposed a novel graph model for embedded programs which we have named *System Dependence Graph with Control flow* (SDGC) which is an original contribution. An SDGC model is an extension of the standard CFG and SDG models and is able to capture the following important information of an embedded program: tasks, task precedences, task priorities, inter-task communication using message passing or shared

resources, timers, and exception handling.

Since an SDGC is an extension of an SDG model, therefore all the node and edge types defined for an SDG model are also present in an SDGC model. In addition, we have introduced additional node and edge types to capture important features of an embedded programs. We have developed an algorithm ConstructSDGC to construct SDGC models from a given embedded C program. As part of our work, we have analyzed and reported the time and space complexity requirements of the algorithm ConstructSDGC. We have also developed a prototype tool which implements our SDGC construction algorithm. We have tested and manually verified the correctness of our SDGC construction algorithm with several case studies adapted from the automotive control domain.

8.1.2. RTSEM: A Model-Based RTS Technique for Embedded Programs

Existing RTS techniques for procedural programs are either based on analysis of data and control dependencies or based on analysis of only control flow information. However, modifications to a task in an embedded application can affect the timing behavior (i.e., completion times) of other tasks. Therefore, selecting regression test cases for embedded programs using existing techniques can be unsafe. In this context, we have proposed a regression test selection technique which we have named *R*egression *T*est *S*election for *EM*bedded programs (RTSEM). RTSEM selects regression test cases by analyzing the execution dependencies that exist among tasks in addition to the usual control and data dependency analysis. Our technique determines the execution dependencies among tasks that arise due to various issues such as task precedence orders, task priorities, inter-task communication using message queues and semaphores, exception handling, and execution of interrupt handlers.

To study the effectiveness of our approach, we have implemented RTSEM to realize a prototype tool. We have named our prototype implementation `MTest` which stands for *Model-based Test case selector*. For our experimental studies, we used eight industry C programs from the automotive control domain. To compare the performance and effectiveness of our approach with a popular approach for RTS of procedural programs, we also selected regression test cases using the SDG-based RTS approach proposed by Binkley [19]. From our experimental studies, we observe a 28.33% increase on the average in the number of selected regression test

cases over Binkley's approach. What is more important is that there was a 36.36% increase in the number of *fault-revealing* test cases that were selected for regression testing. Thus, our experimental studies carried out using `MTest` show that it is important to consider the special semantics of embedded program features such as tasks, inter-task communication, etc., and the task execution dependencies that arise due to these features for RTS.

8.1.3. GA-TSO: A RTSO Technique for Embedded Programs

Our third major contribution is a multi-objective RTSO technique for embedded applications which we have named GA-TSO. Our multi-objective RTSO technique selects a pareto optimal set of test cases that minimizes the cost of regression testing, maximizes the reliability of the frequently-executed non-critical functionalities of the PUT and can be executed within the time allotted for regression testing. Since many embedded applications are being extensively used in safety-critical and real-time application domains, therefore it is also important that the test cases that test critical functionalities and affected real-time tasks are not omitted during RTSO. Unless these constraints are met, the thoroughness of regression testing an embedded application using an optimized test suite may be severely compromised.

We have developed our multi-objective RTSO technique for embedded programs using genetic algorithms. We have also developed a prototype tool implementing GA-TSO as a console application in C programming language. We have carried out our experimental studies using a set of five embedded C programs used during our RTS work. Experimental studies carried out by us show that the test suites optimized by our method include all the fault-revealing test cases from the initial regression test suite and at the same time achieve substantial savings in regression testing effort. However, the thoroughness of regression testing is not compromised since all the relevant test cases that test the critical and frequently-used functionalities in an embedded program are also executed during regression testing.

8.2. Directions for Future Research

During the course of this work, many new ideas emerged related to regression test selection and optimization for embedded programs. It would be interesting to explore these ideas and check whether they can feasibly be used to improve our

current approaches. We discuss some of these possible extensions of our work in the following.

Aperiodic and sporadic tasks and asynchronous message passing: At present, our work is based on a few simplifying assumptions. For example, we consider the task execution dependencies introduced only due to the synchronous message passing model. We have also assumed that the tasks in an embedded program are only periodic in nature. Sophisticated embedded applications often include aperiodic and sporadic tasks, and may communicate using the asynchronous message passing model also. Our RTS technique at present does not consider the dependencies that may arise due to presence of these features in an embedded program. Therefore, a suitable RTS technique needs to be proposed which would analyze and select test cases based on the additional dependencies that arise due to these features.

Embedded program development using C++/UML: Embedded software developed for many applications such as those from the infotainment and telematics domain are larger and more complex. These applications usually require improved visual interfaces, and are event-based. Therefore, infotainment and telematics applications are often developed using C++/UML to take advantage of object-oriented development practices. For safe RTS of such embedded applications developed using C++/UML, it is important that the dependencies introduced due to use of object-oriented features are also taken into account.

Model Driven Development: Of late, model-driven development (MDD) has been receiving a lot of attention. The model-driven paradigm is being extensively used for development of embedded controllers in various application domains such as avionics, automotive, industrial control, etc. In MDD, there exists a close correspondence between the design model(s) and code. Therefore, instead of performing RTS solely based on code analysis, test selection could also be performed based on an analysis of the design models.

In the industry, usually UML models are used for discrete control application development, whereas for hybrid control applications, MATLAB Simulink/Stateflow (SL/SF) models [115] are more popular. For example, SL/SF model-based development is extensively used in the automotive domain for developing heating, ventilation and air conditioning (HVAC) systems, power window controllers, etc. For evolving embedded applications in an MDD environment, the modifica-

tions may be carried out in any of the following two ways: (a) the design model is modified, and (b) the code auto-generated from the model is modified. In the latter case, a minor change made to the code may not mandate a change to the original model. As an example, consider a variable assignment statement which is modified in the code. This change is not reflected in the design models, but it needs to be noted that the change can affect a guard condition or an action statement in the SL/SF model due to data dependencies. In this context, an RTS technique for control applications needs to select test cases based on both code analysis as well as model analysis.

Model-based RTS can help take into consideration several aspects of embedded programs that are not easily extracted from the code. Such program aspects include object states, message path information, exception handling, timing characteristics etc. Some of these information can easily be extracted from the design models and the SRS document, and can be incorporated in the graph model representing the embedded program under test. An analysis of such a model, that has been augmented with the information extracted from the design and analysis models, can help to accurately identify all the relevant regression test cases.

Disseminations out of this Work

1. S. Biswas, R. Mall, M. Satpathy, S. Sukumaran. A Model-Based Regression Test Selection Approach for Embedded Applications. *ACM SIGSOFT Software Engineering Notes*, 34(4):1-9, July 2009.

2. S. Biswas, R. Mall, M. Satpathy, S. Sukumaran. Regression Test Selection Techniques: A Survey. *Submitted* to *Informatica*.

3. S. Biswas, R. Mall, M. Satpathy. Task Dependency Analysis for Regression Test Selection of Embedded Programs. *Submitted* to *IEEE Embedded Systems Letters*.

4. S. Biswas and R. Mall. Model-Based Optimization of Regression Test Suites for Embedded Applications. *Submitted* to *TENCON 2011*.

5. S. Biswas, R. Mall, M. Satpathy. A Regression Test Selection Technique for Embedded Applications. *To be submitted* to *ACM Transactions in Embedded Computing Systems*.

Bibliography

[1] ANTLR Parser Generator. Website. `http://www.antlr.org/`.

[2] ANTLR plugin for Eclipse. Website. `http://antlreclipse.sourceforge.net/`.

[3] Eclipse. Website. `http://www.eclipse.org/`.

[4] Gcov - Using the GNU Compiler Collection (GCC). Website. `http://gcc.gnu.org/onlinedocs/gcc/Gcov.html`.

[5] Graphviz. Website. `http://www.graphviz.org/`.

[6] Minimalist GNU for Windows. Website. `http://www.mingw.org/`.

[7] MISRA-C: 2004 - Guidelines for the use of the C language in critical systems. Website, October 2004.

[8] IEC 61508. Functional safety of electrical/electronic/programmable electronic safety-related systems - part 1: General requirements, April 2010.

[9] K. Abdullah and L.White. A firewall approach for the regression testing of object-oriented software. In *Proceedings of 10th Annual Software Quality Week*, page 27, May 1997.

[10] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Dorling Kindersley (India) Pvt Ltd, 2nd edition, 2008.

[11] A. Ali, A. Nadeem, Z. Iqbal, and M. Usman. Regression testing based on UML design models. In *Proceedings of the 13th Pacific Rim International Symposium on Dependable Computing*, pages 85–88, 2007.

[12] M. Allen and S. Horwitz. Slicing java programs that throw and catch exceptions. In *PEPM '03: Proceedings of the 2003 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*, pages 44–54, 2003.

[13] T. Ball. On the limit of control flow analysis for regression test selection. In *ISSTA '98: Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*, pages 134–142, 1998.

[14] G. Baradhi and N. Mansour. A comparative study of five regression testing algorithms. In *Proceedings of Australian Software Engineering Conference, Sydney*, pages 174–182, 1997.

[15] S. Bates and S. Horwitz. Incremental program testing using program dependence graphs. In *Conference Record of 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 384–396, January 1993.

[16] J. Bible, G. Rothermel, and D. Rosenblum. A comparative study of coarse- and fine-grained safe regression test-selection techniques. *ACM Transactions on Software Engineering and Methodology*, 10(2):149–183, April 2001.

[17] R. Binder. *Testing Object-Oriented Systems:Models, Patterns, and Tools*. Addison-Wesley, 1999.

[18] D. Binkley. Reducing the cost of regression testing by semantics guided test case selection. In *Proceedings of the Conference on Software Maintenance*, pages 251–260. IEEE Computer Society Press, 1995.

[19] D. Binkley. Semantics guided regression test cost reduction. *IEEE Transactions on Software Engineering*, 23(8):498–516, August 1997.

[20] S. Biswas and R. Mall. Regression test selection techniques: A survey. Technical report, Indian Institute of Technology, Kharagpur, India, March 2011.

[21] S. Biswas, R. Mall, M. Satpathy, and S. Sukumaran. A model-based regression test selection approach for embedded applications. *ACM SIGSOFT Software Engineering Notes*, 34(4):1–9, July 2009.

[22] L. Briand, Y. Labiche, and S. He. Automating regression test selection based on UML designs. *Information and Software Technology*, 51(1):16–30, January 2009.

[23] E. Cartaxo, W. Andrade, F. Neto, and P. Machado. LTS-BT: A tool to generate and select functional test cases for embedded systems. In *SAC '08: Proceedings of the 2008 ACM Symposium on Applied Computing*, pages 1540–1544, 2008.

[24] E. Cartaxo, P. Machado, and F. Neto. On the use of a similarity function for test case selection in the context of model-based testing. *Software Testing, Verification And Reliability*, July 2009.

[25] Y. Chen, D. Rosenblum, and K. Vo. TestTube: A system for selective regression testing. In *Proceedings of the 16th International Conference on Software Engineering*, pages 211–222, May 1994.

[26] P. Chittimalli and M. Harrold. Regression test selection on system requirements. In *ISEC '08: Proceedings of the 1st conference on India software engineering conference*, pages 87–96, 2008.

[27] A. Cleve, J. Henrard, and J. Hainaut. Data reverse engineering using system dependency graphs. In *Proceedings of the 13th Working Conference on Reverse Engineering*, pages 157–166, 2006.

[28] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 2001.

[29] H. Do, S. Mirarab, L. Tahvildari, and G. Rothermel. The effects of time constraints on test case prioritization: A series of controlled experiments. *IEEE Transactions on Software Engineering*, 36(5):593–617, September 2010.

[30] S. Elbaum, A.Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE Transactions of Software Engineering*, 28(2):159–182, February 2002.

[31] E. Engström, P. Runeson, and M. Skoglund. A systematic review on regression test selection techniques. *Information and Software Technology*, 52(1):14–30, January 2010.

[32] E. Engström, M. Skoglund, and P. Runeson. Empirical evaluations of regression test selection techniques: a systematic review. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 22–31, 2008.

[33] Q. Farooq, M. Iqbal, Z. Malik, and A. Nadeem. An approach for selective state machine based regression testing. In *Proceedings of the 3rd international workshop on Advances in model-based testing*, pages 44–52, 2007.

[34] U. Farooq and C. Lam. A max-min multiobjective technique to optimize model based test suite. In *SNPD '09: Proceedings of the 2009 10th ACIS International Conference on Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing*, pages 569–574, 2009.

[35] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[36] C. Fonseca and P. Fleming. Genetic algorithms for multiobjective optimization: Formulation, discussion and generalization. In *Genetic Algorithms: Proceedings of the 5th International Conference*, pages 416–423. Morgan Kaufmann, July 1993.

[37] J. Gao, D. Gopinathan, Q. Mai, and J. He. A systematic regression testing method and tool for software components. In *Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, pages 455–466, 2006.

[38] D. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, 2000.

[39] R. Gorthi, A. Pasala, K. Chanduka, and B. Leong. Specification-based approach to select regression test suite to validate changed software. In *Proceedings of the 2008 15th Asia-Pacific Software Engineering Conference*, pages 153–160, 2008.

[40] T. Graves, M. Harrold, J. Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. *ACM Transactions on Software Engineering and Methodology*, 10(2):184–208, April 2001.

[41] J. Guan, J. Offutt, and P. Ammann. An industrial case study of structural testing applied to safety-critical embedded software. In *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, pages 272–277, 2006.

[42] R. Gupta, M. Harrold, and M. Soffa. Program slicing-based regression testing techniques. *Journal of Software Testing, Verification, and Reliability*, 6(2):83–112, June 1996.

[43] M. Harrold, R. Gupta, and M. Soffa. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology*, 2(3):270–285, July 1993.

[44] M. Harrold, J. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. Regression test selection for Java software. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 312–326, January 2001.

[45] M. Harrold and M. Soffa. An incremental approach to unit testing during maintenance. In *Proceedings of the International Conference on Software Maintenance*, pages 362–367, October 1988.

[46] M. Harrold and M. Soffa. Interprocedural data flow testing. In *Proceedings of the ACM SIGSOFT '89 third symposium on Software testing, analysis, and verification*, pages 158–167, December 1989.

[47] D. Hatley and I. Pirbhai. *Strategies for Real-Time System Specification*. Dorset House Publishing Company, 1987.

[48] K. Hla, Y. Choi, and J. Park. Applying particle swarm optimization to prioritizing test cases for embedded real time software retesting. In *Proceedings of the 2008 IEEE 8th International Conference on Computer and Information Technology Workshops - Volume 00*, pages 527–532, 2008.

[49] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–61, January 1990.

[50] P. Hsia, X. Li, D. Kung, C. Hsu, L. Li, Y. Toyoshima, and C. Chen. A technique for the selective revalidation of object-oriented software. *Journal of Software Maintenance: Research and Practice*, 9(4):217–233, 1997.

[51] IEEE Standards Association. IEEE posix certification authority. Website. `http://standards.ieee.org/regauth/posix/`.

[52] Y. Jang, M. Munro, and Y. Kwon. An improved method of selecting regression tests for C++ programs. *Journal of Software Maintenance: Research and Practice*, 13(5):331–350, September 2001.

[53] B. Jiang, Y. Mu, and Z. Zhang. Research of optimization algorithm for path-based regression testing suit. In *2010 Second International Workshop on Education Technology and Computer Science*, pages 303–306, 2010.

[54] S. Jiang, S. Zhou, Y. Shi, and Y. Jiang. Improving the preciseness of dependence analysis using exception analysis. In *Proceedings of the 15th International Conference on Computing IEEE*, pages 277–282, 2006.

[55] G. Kapfhammer. *The Computer Science Handbook*, chapter on Software testing. CRC Press, Boca Raton, FL, 2nd edition, 2004.

[56] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Proceedings of IEEE International Conference on Neural Networks, Piscataway, NJ.*, pages 1942–1948, 1995.

[57] D. Knuth. Structured programming with go to statements. *ACM Computing Surveys (CSUR)*, 6(4):261–301, December 1974.

[58] A. Konak, D. Coit, and A. Smith. Multi-objective optimization using genetic algorithms: A tutorial. *Reliability Engineering & System Safety*, 91(9):992–1007, September 2006.

[59] H. Kondoh and K. Futatsugi. To use or not to use the goto statement: programming styles viewed from Hoare logic. *Science of Computer Programming*, 60(1):82–116, March 2006.

[60] J. Korpi and J. Koskinen. *Advances and Innovations in Systems, Computing Sciences and Software Engineering*, chapter Supporting Impact Analysis by Program Dependence Graph Based Forward Slicing, pages 197–202. Springer Netherlands, 2007.

[61] D. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima, and C. Chen. On regression testing of object-oriented programs. *Journal of Systems and Software*, 32(1):21–40, January 1996.

[62] J. Laski and W. Szermer. Identification of program modifications and its applications in software maintenance. In *Proceedings of the Conference on Software Maintenance*, pages 282–290, November 1992.

[63] H. Leung and L. White. Insights into regression testing. In *Proceedings of the Conference on Software Maintenance*, pages 60–69, 1989.

[64] H. Leung and L. White. A study of integration testing and software regression at the integration level. In *Proceedings of the Conference on Software Maintenance*, pages 290–300, November 1990.

[65] H. Leung and L. White. A firewall concept for both control-flow and data-flow in regression integration testing. In *Proceedings of the Conference on Software Maintenance*, pages 262–270, 1992.

[66] D. Liang and M. Harrold. Slicing objects using system dependence graphs. In *Proceedings of the International Conference on Software Maintenance*, pages 358–367, November 1998.

[67] Feng Lin, Michael Ruth, and Shengru Tu. Applying safe regression test selection techniques to java web services. *Next Generation Web Services Practices, International Conference on*, 0:133–142, September 2006.

[68] J. Lin, C. Huang, and C. Lin. Test suite reduction analysis with enhanced tie-breaking techniques. In *4th IEEE International Conference on Management of Innovation and Technology, 2008. ICMIT 2008.*, pages 1228–1233, September 2008.

[69] R. Mall. *Real-Time Systems Theory and Practice*. Pearson Education, 1st edition, 2007.

[70] N. Mansour and K. El-Fakih. Simulated annealing and genetic algorithms for optimal regression testing. *Journal of Software Maintenance: Research and Practice*, 11(1):19–34, 1999.

[71] C. Mao and Y. Lu. Regression testing for component-based software systems by enhancing change information. In *APSEC '05: Proceedings of the 12th Asia-Pacific Software Engineering Conference*, pages 611–618, 2005.

[72] C. Mao, Y. Lu, and J. Zhang. Regression testing for component-based software via built-in test design. In *Proceedings of the 2007 ACM symposium on Applied computing*, pages 1416–1421, 2007.

[73] P. Marwedel. *Embedded System Design*. Springer, 2007.

[74] A. Mathur. *Foundations of Software Testing*. Pearson Education, 2008.

[75] R. Maxion and R. Olszewski. Improving software robustness with dependability cases. In *28th International Symposium on Fault Tolerant Computing*, pages 346–355, 1998.

[76] S. McConell. *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press, 2nd edition, June 2004.

[77] J. McGregor and D. Sykes. *A Practical Guide to Testing Object-Oriented Software*. Addison-Wesley, March 2001.

[78] J. Musa. Operational profiles in software-reliability engineering. *IEEE Software*, 10(2):14–32, March 1993.

[79] L. Naslavsky and D. Richardson. Using traceability to support model-based regression testing. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, November 2007.

[80] M. Netkow and D. Brylow. Xest: an automated framework for regression testing of embedded software. In *Proceedings of the 2010 Workshop on Embedded Systems Education*, WESE '10, pages 7:1–7:8. ACM, October 2010.

[81] A. Orso, M. Harrold, D. Rosenblum, G. Rothermel, M. Soffa, and H. Do. Using component metacontent to support the regression testing of component-based software. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, pages 716–725, 2001.

[82] A. Orso, N. Shi, and M. Harrold. Scaling regression testing to large software systems. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, pages 241–251, November 2004.

[83] OSEK. OSEK/VDX time-triggered operating system specification 1.0. Website, July 2001. `http://portal.osek-vdx.org`.

[84] S. Parsa and A. Khalilian. On the optimization approach towards test suite minimization. *International Journal of Software Engineering and Its Applications*, 4(1), January 2010.

[85] A. Pasala, Y Fung, F. Akladios, A. Raju, and R. Gorthi. Selection of regression test suite to validate software applications upon deployment of upgrades. In *19th Australian Conference on Software Engineering*, pages 130–138, March 2008.

[86] C. Poole and J. Huisman. Using extreme programming in a maintenance environment. *IEEE Software*, 18(6):42–50, November 2001.

[87] D. Ritchie and B. Kernigham. *The C Programming Language*. Prentice Hall of India, 2nd edition, 2007.

[88] A. Romanovsky, J. Xu, and B. Randell. Exception handling in object-oriented real-time distributed systems. In *ISORC '98 Proceedings of the The 1st IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, ISORC '98, pages 32 – 42. IEEE Computer Society, April 1998.

[89] G. Rothermel, S. Elbaum, A. Malishevsky, P. Kallakuri, and X. Qiu. On test suite composition and cost-effective regression testing. *ACM Transactions on Software Engineering and Methodology*, 13(3):277–331, 2003.

[90] G. Rothermel and M. Harrold. A safe, efficient algorithm for regression test selection. In *Proceedings of the Conference on Software Maintenance*, pages 358–367, 1993.

[91] G. Rothermel and M. Harrold. Selecting regression tests for object-oriented software. In *International Conference on Software Maintenance*, pages 14–25, March 1994.

[92] G. Rothermel and M. Harrold. Selecting tests and identifying test coverage requirements for modified software. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 169–184, August 1994.

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

[94] G. Rothermel and M. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210, April 1997.

[95] G. Rothermel, M. Harrold, and J. Dedhia. Regression test selection for C++ software. *Software Testing, Verification and Reliability*, 10:77–109, June 2000.

[96] G. Rothermel, M. Harrold, J. Ostrin, and C. Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *Proceedings of the International Conference on Software Maintenance*, pages 34–43, November 1998.

[97] G. Rothermel, R. Untch, C. Chu, and M. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, October 2001.

[98] RTCA/DO-178B. Software considerations in airborne systems and equipment certification, December 1992.

[99] M. Ruth, S. Oh, A. Loup, B. Horton, O. Gallet, M. Mata, and S. Tu. Towards automatic regression test selection for web services. In *Proceedings of the 31st Annual International Computer Software and Applications Conference - Volume 02*, COMPSAC '07, pages 729–736. IEEE Computer Society, 2007.

[100] M. Ruth and S. Tu. A safe regression test selection technique for web services. In *Proceedings of the Second International Conference on Internet and Web Applications and Services*, pages 47–. IEEE Computer Society, 2007.

[101] A. Sajeev and B. Wibowo. Regression test selection based on version changes of components. In *APSEC '03: Proceedings of the Tenth Asia-Pacific Software Engineering Conference Software Engineering Conference*, page 78, 2003.

[102] Jane Sales. *Symbian OS Internals: Real-Time Kernel Programming*. John Wiley & Sons, 2005.

[103] F. Salewski and A. Taylor. Fault handling in FPGAs and microcontrollers in safety-critical embedded applications: A comparative survey. In *ISESE '06: Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, pages 124–131, August 2007.

[104] A. Sangiovanni-Vincentelli and M. Di Natale. Embedded system design for automotive applications. *IEEE Computer Society*, 40:42–51, October 2007.

[105] I. Sbalzariniy, S. Müller, and P. Koumoutsakos. Multiobjective optimization using evolutionary algorithms. In *Center for Turbulence Research Proceedings of the Summer Program 2000*, July 2000.

[106] T. Schotland and P. Petersen. Exception handling in C without C++. Online, February 2011.

[107] J. Seo, Y. Ki, B. Choi, and K. La. Which spot should I test for effective embedded software testing? In *SSIRI '08: Proceedings of the 2008 Second International Conference on Secure System Integration and Reliability Improvement*, pages 135–142, 2008.

[108] A. Silberschatz, P. Galvin, and G. Gagne. *Operating System Concepts*. Wiley India Pvt Ltd, 8th edition, 2010.

[109] S. Sinha, M. Harrold, and G. Rothermel. System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow. In *Proceedings of the 21st International Conference on Software Engineering*, pages 432–441, 1999.

[110] S. Sinha and M. J. Harrold. Analysis of programs with exception-handling constructs. In *ICSM '98: Proceedings of the International Conference on Software Maintenance*, page 348. IEEE Computer Society, 1998.

[111] D. Sundmark, A. Pettersson, S. Eldh, M. Ekman, and H. Thane. Efficient system-level testing of embedded real-time software. In *Work in Progress Session of the 17th Eurmicro Conference on Real-Time System, Spain*, pages 53–56, December 2007.

[112] D. Sundmark, A. Pettersson, and H. Thane. Regression testing of multi-tasking real-time systems: A problem statement. *ACM SIGBED Review*, 2(2):31–34, April 2005.

[113] A. Taha, S. Thebaut, and S. Liu. An approach to software fault localization and revalidation based on incremental data flow analysis. In *Proceedings of the 13th Annual International Computer Software and Applications Conference*, pages 527–534, September 1989.

[114] A. Tarhini, H. Fouchal, and N. Mansour. Regression testing web services-based applications. In *AICCSA '06 Proceedings of the IEEE International Conference on Computer Systems and Applications*, pages 163–170. IEEE Computer Society, 2006.

[115] The Mathworks, Inc. MATLAB. Website, April 2011. `http://www.mathworks.com`.

[116] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3), September 1995.

[117] F. Vahid and T. Givargis. *Embedded System Design: A Unified Hardware/Software Introduction*. John Wiley & Sons, 1st edition, 2002.

[118] F. Vokolos. *A regression test selection technique based on textual differencing*. u, Polytechnic University, 1998.

[119] F. Vokolos and P. Frankl. Pythia: A regression test selection tool based on textual differencing. In *Proceedings of the 3rd International Conference on Reliability, Quality & Safety of Software-Intensive Systems (ENCRESS' 97)*, pages 3–21, May 1997.

[120] F. Vokolos and P. Frankl. Empirical evaluation of the textual differencing regression testing technique. In *ICSM '98: Proceedings of the International Conference on Software Maintenance*, pages 44–53, 1998.

[121] K. Walcott, M. Soffa, G. Kapfhammer, and R. Roos. Time aware test suite prioritization. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, pages 1–12, 2006.

[122] N. Walkinshaw, M. Roper, and M. Wood. The Java system dependence graph. In *Third IEEE International Workshop on Source Code Analysis and Manipulation*, pages 55–64, September 2003.

[123] P. Ward and S. Mellor. *Structured Development for Real-Time Systems*. Prentice Hall Professional Technical Reference, 1991.

[124] M. Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449, 1981.

[125] David Wigg. ANTLR C++ grammar. Website. `http://www.antlr.org/grammar/list`.

[126] N. Wilde and R. Huitt. Maintenance support for object-oriented programs. *IEEE Transactions on Software Engineering*, 18(12):1038–1044, December 1992.

[127] Wind River Systems. Wind River VxWorks: Embedded RTOS with support for POSIX and SMP. Website, August 2010. `http://www.windriver.com/products/vxworks/`.

[128] W. Wong, J. Horgan, S. London, and A. Mathur. A study of effective regression testing in practice. In *Proceedings of the Eighth International Symposium on Software Reliability Engineering*, pages 230–238, November 1997.

[129] Y. Wu and J. Offutt. Maintaining evolving component-based software with UML. In *Proceedings of 7th European Conference on Software Maintenance and Reengineering (CSMR '03)*, pages 133–142, March 2003.

[130] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen. A brief survey of program slicing. *ACM SIGSOFT Software Engineering Notes*, 30(2):1–36, March 2005.

[131] G. Xu and A. Rountev. Regression test selection for AspectJ software. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 65–74, 2007.

[132] Lei Xu, Baowen Xu, Zhenqiang Chen, Jixiang Jiang, and Huowang Chen. Regression testing for web applications based on slicing. *Computer Software and Applications Conference, Annual International*, 0:652, 2003.

[133] Z. Xu, K. Gao, and T. Khoshgoftaar. Application of fuzzy expert system in test case selection for system regression test. In *2005 IEEE International Conference on Information Reuse and Integration*, pages 120–125, August 2005.

[134] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 1(1):121–141, March 2010.

[135] X. Zhang, H. Shan, and J. Qian. Resource-aware test suite optimization. In *QSIC '09: Proceedings of the 2009 Ninth International Conference on Quality Software*, pages 341–346, 2009.

[136] J. Zhao, T. Xie, and N. Li. Towards regression test selection for AspectJ programs. In *Proceedings of the 2nd workshop on Testing aspect-oriented programs*, WTAOP '06, pages 21–26. ACM, 2006.

[137] J. Zheng, B. Robinson, L. Williams, and K. Smiley. An initial study of a lightweight process for change identification and regression test selection when source code is not available. In *Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering*, pages 225–234, November 2005.

[138] J. Zheng, B. Robinson, L. Williams, and K. Smiley. Applying regression test selection for COTS-based applications. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 512–522, May 2006.

[139] J. Zheng, B. Robinson, L. Williams, and K. Smiley. A lightweight process for change identification and regression test selection in using COTS components. In *ICCBSS '06: Proceedings of the Fifth International Conference on Commercial-off-the-Shelf (COTS)-Based Software Systems*, pages 137–143, February 2006.