Regression Test Selection Techniques: A Survey

Swarnendu Biswas and Rajib Mall
Dept. of Computer Science and Engineering
IIT Kharagpur, India - 721302
E-mail: {swarnendu, rajib}@cse.iitkgp.ernet.in

Manoranjan Satpathy and Srihari Sukumaran
GM India Science Lab, Bangalore, India
E-mail: {manoranjan.satpathy, srihari.sukumaran}@gm.com

Overview paper

Keywords: software maintenance, regression testing, regression test selection, model-based testing, UML, software components, embedded programs

Received: April 12, 2010

Regression testing is an important and expensive activity that is undertaken every time a program is modified to ensure that the modifications do not introduce new bugs into previously validated code. An important research problem, in this context, is the selection of a relevant subset of test cases from the initial test suite that would minimize both the regression testing time and effort without sacrificing the thoroughness of regression testing. Researchers have proposed a number of regression test selection techniques for different programming paradigms such as procedural, object-oriented, component-based, database, aspect, and web applications. In this paper, we review the important regression test selection techniques proposed for various categories of programs and identify the emerging trends.

Povzetek: Podan je pregled tehnik izbora testov za regresijsko testiranje programov.

1 Introduction

Software maintenance activities, on an average, account for as much as two-thirds of the overall software life cycle costs [75]. Maintenance of a software product is frequently necessitated to fix defects, to add, enhance or adapt existing functionalities, or to port it to different environments. Whenever an application program is modified for carrying out any maintenance activity, *resolution* test cases are designed and executed to check that the modified parts of the code work properly. *Regression* testing (also referred to as *program revalidation*) is carried out to ensure that no new errors (called regression errors) have been introduced into previously validated code (i.e., the unmodified parts of the program) [55]. Although regression testing is usually associated with system testing after a code change, regression testing can be carried out at either unit, integration or system testing levels. The sequence of activities that take place during the maintenance phase after the release of a software is shown in Figure 1. The figure shows that after a software is released, the failure reports and the change requests for the software are compiled, and the software is modified to make necessary changes. Resolution tests are carried out to verify the directly modified parts of the code, while regression test cases are carried out to test the unchanged parts of the code that may be affected by the code change. After the testing is complete, the new version of the software is released, which then undergoes a similar cycle.

Regression testing is acknowledged to be an expensive activity. It consumes large amounts of time as well as effort, and often accounts for almost half of the software maintenance costs [55, 49]. The extents to which time and effort are being spent on regression testing are exemplified by a study [22] that reports that it took 1000 machine-hours to execute approximately 30,000 functional test cases for a software product. It is also important to note that hundreds of man-hours are spent by test engineers to oversee the regression testing process; that is to set up test runs, monitor test execution, analyze results, and maintain testing resources, etc [22]. Minimization of regression test effort is, therefore, an issue of considerable practical importance, and has the potential to substantially reduce software maintenance costs.

Regression test selection (RTS) techniques select a subset of valid test cases from an initial test suite (T) to test that the affected but unmodified parts of a program continue to work correctly. Use of an effective *regression test selection* technique can help to reduce the testing costs in environments in which a program undergoes frequent modifications. Regression test selection essentially consists of two major activities:

– Identification of the affected parts - This involves

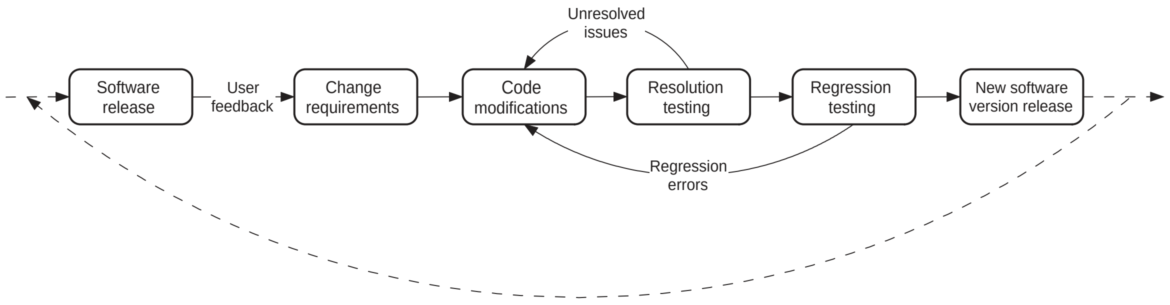


Figure 1: Activities that take place during software maintenance and regression testing.

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 [78] 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'.*

Leung and White [57] have observed that the use of an RTS technique can reduce the cost of regression testing compared to the *retest-all* approach, which involves running the entire test suite T to revalidate a modified program P', only if the cost of selecting a reduced subset of test cases to be run on P' is less than the cost of running the tests that the RTS technique omits. The *retest-all* approach is considered impractical on account of cost, resource and delivery schedule constraints that projects are frequently subjected to. Another approach is to *randomly* select test cases from T to carry out regression testing. However, random selection of test cases may fail to expose many regression errors. RTS techniques aim to overcome the drawbacks associated with the retest-all approach and in random selection of test cases by precisely selecting only those test cases that test the unmodified but affected parts of the program.

Though substantial research results on RTS have been reported in the literature, several studies [35, 36] show that very few software industries deploy systematic test selection strategies or automation support during regression testing. The approaches that are most often used in the industry for identification of relevant regression test cases are either based on expert judgment, or based on some form of manual program analysis. However, selection of test cases based on expert judgment tends to become ineffective and unreliable for large software products. Even for moderately complex systems, it is usually extremely difficult to manually identify test cases that are relevant to a change. This approach often leads to a large number of test cases being selected and rerun even for small changes to the original program, 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 be overlooked during manual selection. Another problem that surfaces during regression testing stems from the fact that testers (either from the same organization or from third-party companies) are usually supplied with only the functional description of the software, and therefore lack adequate knowledge about the code to precisely select only those test cases that are relevant to a modification [74].

A large number of RTS techniques have been reported for procedural [5, 7, 10, 37, 43, 44, 54, 56, 58, 80] and object-oriented programs [4, 14, 41, 73, 82], each aimed at leveraging certain optimization options. These 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 component-based [31, 66, 67, 72, 115, 116, 117], aspect programs [114, 109], web applications [86, 93, 61, 110, 85], etc.

RTS techniques have been reviewed by several authors [79, 6, 8, 34, 25, 24, 112]. In [79], Rothermel and Harrold have proposed a set of metrics to evaluate the effectiveness of different RTS techniques. Baradhi and Mansour [6], Bible et al. [8], and Graves et al. [34] 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 [25]. This lead Engström et al. to perform a qualitative study [25, 24] of the nature of the empirical data considered. The studies reported in [25, 24] are based on the similarities of the different RTS techniques and the quality of the empirical data used. Engström et al. [25] observe that it is very difficult to come up with an RTS technique which is generic enough (i.e., can be applied to different classes of applications) and is superior to all other techniques. The survey carried out by Engström et al. considers techniques which have been published before 2006. Therefore, their survey does not include many RTS techniques proposed after 2006 [114, 109, 18, 61, 86, 93, 85, 65, 31], and also does not include a few RTS techniques which were proposed before 2006 [10, 110, 107]. Moreover, their study does not include a detailed discussion about the merits and demerits of each technique.

In this paper, we present a detailed review of the RTS techniques proposed for different programming paradigms such as procedural, object-oriented, component-based, database, aspect and web software. Since a large number of RTS techniques have been proposed in the literature, we have limited our study to only the more prominent classes of RTS techniques. The techniques we have reviewed have been chosen based on their prominence determined by the number of citations and their frequency of referrals in other related studies. Our sources of information are existing reviews on RTS techniques [79, 6, 8, 25, 24, 34, 112], the citation index of the papers that we studied, and the online digital libraries, such as IEEE Xplore, ACM Digital Library, ScienceDirect, etc. The keywords that we used for our search on the online digital libraries include *regression testing*, *regression test selection*, *test selection*, etc. As an aid to understanding, and to keep the size of the review manageable, we have classified different RTS techniques together into relevant classes based on the motivation and similarity of the proposed approaches. We present a brief discussion on the working of each class of techniques, and discuss the merits and demerits of each. We also discuss issues that arise while designing RTS techniques for embedded programs, and identify the emerging trends in regression testing.

This paper is organized as follows: Section 2 presents basic concepts related to regression testing and which have been used in the rest of this paper. In Section 3, we discuss and compare various RTS approaches proposed for procedural programs. Subsequently, we discuss RTS techniques for object-oriented, component-based, database, web and AspectJ programs in Sections 4, 5, 6, 7, and 8 respectively. We discuss techniques for RTS of embedded software in Section 9. We discuss RTS techniques proposed for .Net and BPEL programs in Section 10. We discuss future research directions in regression testing and finally conclude the paper in Section 11.

2 Basic Concepts

In this section, we first discuss a few basic concepts that are extensively used in the context of regression testing. We then discuss some popular intermediate representations which are used for program model-based RTS.

For notational convenience, in the rest of the paper we denote the original and the modified programs by P and P' respectively. The initial regression test suite is denoted by T, and a test case in T is denoted by t.

2.1 Concepts Related to Regression Testing

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

Obsolete, Retestable and Redundant Test Cases: According to Leung and White [55], test cases in the initial test suite can be classified as obsolete, retestable and redundant (or reusable) test cases. Obsolete test cases are no more valid for the modified program. Retestable test cases are those test cases that execute the modified and the affected parts of the program and need to be rerun during regression testing. Redundant test cases execute only the unaffected parts of the program. Hence, although these are valid test cases (i.e., not obsolete), they can be omitted from the regression test suite without compromising the quality of testing.

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 [80]. The execution trace information for P can be generated by appropriately instrumenting the source code.

Fault-revealing Test Cases: A test case $t \in T$ is said to be *fault-revealing* for a program P, iff it can potentially cause P to fail by producing incorrect outputs for P [79].

Modification-revealing Test Cases: A test case $t \in T$ is considered to be *modification-revealing* for P and P', iff it produces different outputs for P and P' [79].

Modification-traversing Test Cases: A test case $t \in T$ is *modification-traversing* for P and P', iff the execution traces of t on P and P' are different [79]. 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 super-set of the set of the modification-revealing test cases.

Inclusive, Precise and Safe Regression Test Cases: Inclusiveness measures the extent to which an RTS technique selects modification-revealing tests from the initial regression test suite T [79]. 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 with respect to P, P' and T is expressed as $(m/n) * 100$ [79].

A *safe* RTS technique selects all those test cases from the initial test suite that are modification-revealing [79]. Therefore, an RTS technique is said to be safe, iff it is 100% inclusive. Regression test cases that are relevant to a change but are not selected by an RTS technique are instances of *false negatives*. Therefore, an RTS technique is safe if the test suite selected by it has no false negatives [18].

Precision measures the extent to which an RTS algorithm ignores test cases that are non-modification-revealing [79]. Test cases that are selected by a technique but are not relevant are false positives. An RTS technique is, therefore, precise iff it there are no false positives among the selected test cases [18].

2.2 Regression Test Suite Minimization and Prioritization

Regression test suite minimization (TSM) techniques [40, 62, 64] aim to reduce the size of the regression test suite by eliminating redundant test cases such that the coverage achieved by the minimized test suite is same as the initial test suite. Different studies published in the literature [83, 106, 62] report conflicting results on the impact of TSM techniques on the fault-detection capabilities of the reduced test suites. Lin et al. have observed [62] that the TSM problem is *NP-complete*, since the minimum set-covering problem [20] can be reduced to the TSM problem in polynomial time.

Regression test case prioritization (TCP) techniques [23, 99, 84] order test cases such that 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 [23, 84]. 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 [84]. 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 case of unpredicted interruptions to testing activities on account of delivery, resource or budget constraints.

Several TSM and TCP approaches have been proposed in recent years, and have emerged as active areas of research by themselves. However, our current work focuses only on RTS techniques. More detailed information about TSM and TCP approaches can be found in [22, 23, 112].

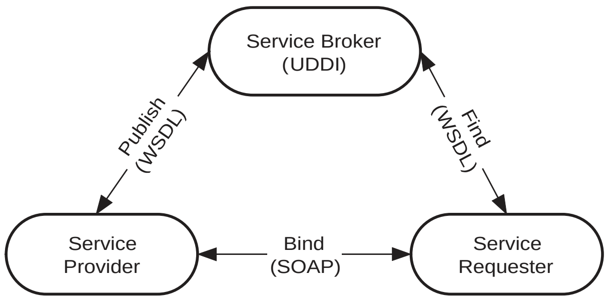


Figure 2: Web service architecture.

2.3 Few Other Relevant Concepts

In the following, we briefly discuss few other concepts that are relevant to this survey.

Program Slicing: Program slicing is a program analysis technique which was first introduced by Weiser [103] to aid in program debugging. 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 every possible input to the program.

Since the publication of Weiser's seminal work, the concept of slicing has been extended and many slicing algorithms have been proposed in the literature for other areas of program analysis such as program understanding, compiler optimization, reverse engineering, etc. More detailed information regarding program slicing can be found in [108, 95].

Web Services: Web services are now being extensively used in application development across distributed and remote platforms, and are examples of service-oriented architecture (SOA) based development. SOA-based development has received a big boost with the advent of standardized web services. A *web service* can be defined as a software component which implements a logic and is designed to be inter-operable over a network providing platform-independence.

Figure 2 shows the typical architecture and the specifications of a web service [93, 13]. A service provider publishes services to a service broker. Service requesters find required services using a service broker and then bind to them. Platform independence is achieved through use of the following web specifications:

– Simple Object Access Protocol (SOAP) - SOAP is an XML-based protocol for information exchange over the network between web service and the users. The XML messages can be transferred using any application layer protocol such as Hypertext Transfer Protocol (HTTP). An advantage of SOAP messages is that

```
     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 );
```

Figure 3: A sample program.

they can be exchanged between applications regardless of the development platform and the programming language being used.

- Web Service Description Language (WSDL) - WSDL is an XML-based language that is designed to provide an interface between a web service and its users.
- Universal Description Discovery and Integration (UDDI) - UDDI is an XML-based information registry where servers can publish their services. It allows users to locate any specific web services they might be interested in.

2.4 Graph Models for Procedural Programs

Graph models of programs have extensively been used in many applications such as program slicing [60, 89], impact analysis [52], reverse engineering [19], computation of program metrics [100], regression test selection [80, 73, 41, 7], etc. Analysis of graph models of programs is more efficient compared to textual analysis, and various types of relationships among program elements are also not explicit in the code. This has led to several representations such as Control Flow Graph (*CFG*) [3], Program Dependence Graph (*PDG*) [29] and System Dependence Graphs (*SDG*) [46] being proposed for procedural programs. In the following, we briefly discuss the important graph models proposed for procedural programs.

2.4.1 Flow Graph

A flow graph for a program P is a directed graph (N, E) where the program statements correspond to the set of nodes N in the flow graph, and the set of edges E represent the relationships among the program statements. However, the nodes in a flow graph can also correspond to basic blocks in a program. Typically it is assumed that there are two distinguished nodes called *start* with in-degree zero and *stop* with out-degree zero. There exists a path from $start$ to every other node in a flow graph, and similarly, there exists a path from every other node in the graph to $stop$.

2.4.2 Control Flow Graph

A control flow graph (*CFG*) [3] is a flow graph that represents the sequence in which the different statements in a program get executed. That is, it represents the flow of execution of control in the program. In fact, a *CFG* captures all the possible flows of execution of a program.

The *CFG* of the program P is the flow graph $G = (N, E)$ where an edge $(m, n) \in E$ indicates possible flow of control from node m to node n. Figure 4 represents the *CFG* of the program shown in Figure 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.

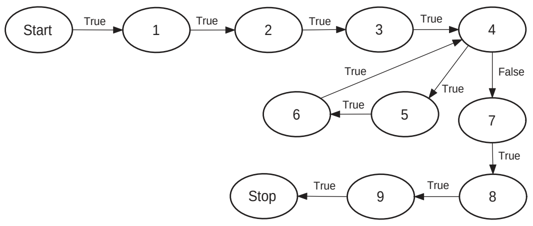


Figure 4: *CFG* for the example program shown in Figure 3.

2.4.3 Data Dependence Graph

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

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 3 and its *CFG* shown in Figure 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 of the 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 $(x, y) \in E$ iff y is data dependent on x.

2.4.4 Control Dependence Graph

The concept of control dependence [3] captures the dependency existing between two program elements when the execution of the second element is dependent on the outcome of the first.

Dominance: If x and y are two nodes in a flow graph, then x dominates y iff every path from $start$ to y passes through x. Similarly, y post-dominates x iff 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 iff 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) iff x is the immediate post-dominator of y.

Control Dependence: Let G be the *CFG* of a program P. Let x and y be two arbitrary nodes in G. A 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 3 and its *CFG* in Figure 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.

Control Dependence Graph: 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 $(x, y) \in E$, iff y is control dependent on x.

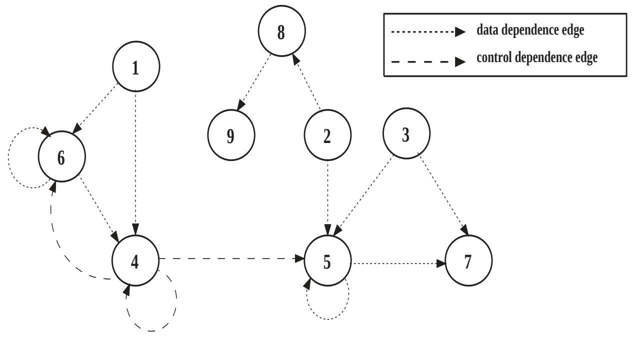


Figure 5: *PDG* of the program in Figure 3.

2.4.5 Program Dependence Graph

The program dependence graph (*PDG*) [29] for a program P explicitly represents both control and data dependencies in a single intermediate representation of P. The *PDG* of a program P is a directed graph $G_{PDG} = (N, E)$, where each node $n \in N$ represents a statement of the program P. A PDG contains both control dependence and data dependence edges. A control (or data) dependence edge (m, n) indicates that n is control (or data) dependent on m. 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 3 is shown in Figure 5.

2.4.6 System Dependence Graph

A major limitation of a *PDG* is that it can model only a single procedure and cannot handle inter-procedural calls. Horwitz et al. [46] enhanced the *PDG* representation to handle procedure calls and introduced the system dependence graph (*SDG*) representation which models the main program together with all the non-nested procedures.

```
     int x = 0;

CE1  class A {
E2     void mA() {
S3       int a = 0;
S4       B *bptr = new B();
S5       cin>>a;
S6       try {
C7         bptr->mB(a); }
S8       catch(E1 &e1) {
S9         cout<<"Error E1"<<endl; }
S10      catch(...) {
S11        cout<<"Error"<<endl; }
S12      cout<<x<<endl; }
     };

CE13 class B {
E14    float mB( int y ) {
S15      try {
S16        if ( y < 0 )
S17          throw new E2();
S18        x = sqrt(y); }
S19      catch(E2 &e2) {
S20        cout<<"Error E2"<<endl;
S21        throw; }
S22      cout<<x<<endl; }
     };

E23  main(int argc, char *argv[]) {
S24    A *aptr = new A();
C25    aptr->mA();
     }
```

Figure 6: An example program.

An *SDG* is very similar to a *PDG*. In fact, the *PDG* of the main program is a subgraph of the *SDG*. In other words, for a program without procedure calls, the *PDG* and the *SDG* are identical. The technique for constructing an *SDG* consists of first constructing a *PDG* for every procedure, including the main procedure, and then adding auxiliary dependence edges which link together the various subgraphs while maintaining call-return discipline.

2.5 Graph Models for Object-Oriented Programs

The object-oriented paradigm is based on several important concepts such as encapsulation, inheritance, polymorphism, dynamic binding, etc. These concepts usually lead to complex relationships among program elements, and render the graph models proposed for procedural programs inadequate for representing object-oriented programs [60]. Therefore, various models for object-oriented programs such as the Class Call Graph (*CCG*) [42], Inter-procedural Program Dependence Graph (*IPDG*) [77], Class Dependence Graph (*ClDG*) [77, 78], and Java Interclass Graph (*JIG*) [41] have been proposed. In the following, we briefly discuss the *ClDG* and *JIG* models.

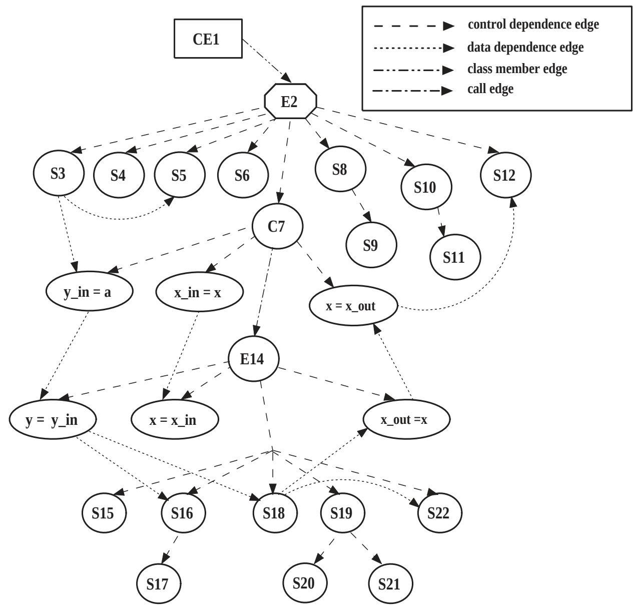


Figure 7: *ClDG* for class A of the program shown in Figure 6.

2.5.1 Class Dependence Graph

A *ClDG* [77, 78] is an extensively used model for intermediate representation of object-oriented programs. Each method in a *ClDG* is represented by its corresponding *PDG*. A class in a *ClDG* is denoted by a *class entry* node and the entry point for each method is represented by a *method entry* node. The *class entry* node is connected to each *method entry* node by a *class member* edge. A representative driver node (RDN) is added to the *ClDG* which summarizes the set of test driver routines used for class testing [77]. This RDN acts as the root of the *ClDG* for the whole program. Each entry node of a public method of the class is directly connected to the RDN by means of *driver edges*, thus implying that the driver routines can invoke the public methods of the class under test. For example, Figure 7 shows the *ClDG* constructed for class A of the program shown in Figure 6. The node labels in the *ClDG* correspond to the statement numbers in the program of Figure 6. The rectangular node labeled $CE1$ in Figure 7 represents the *class entry* vertex for class A. Node $E2$ represents a *method entry* node corresponding to the method `void A::mA()`. The edge $CE1 \rightarrow E2$ in Figure 7 is a *class member* edge. Figure 8 shows the *ClDG* for class B defined in Figure 6.

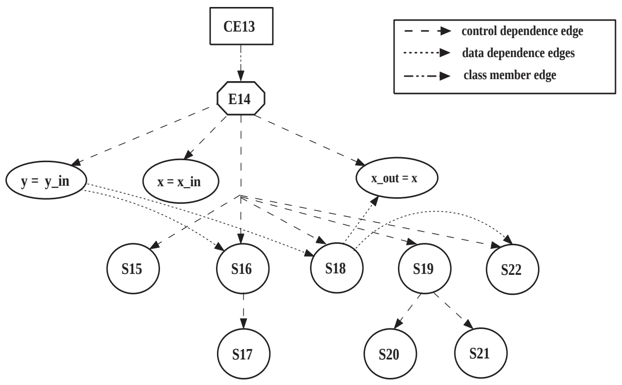
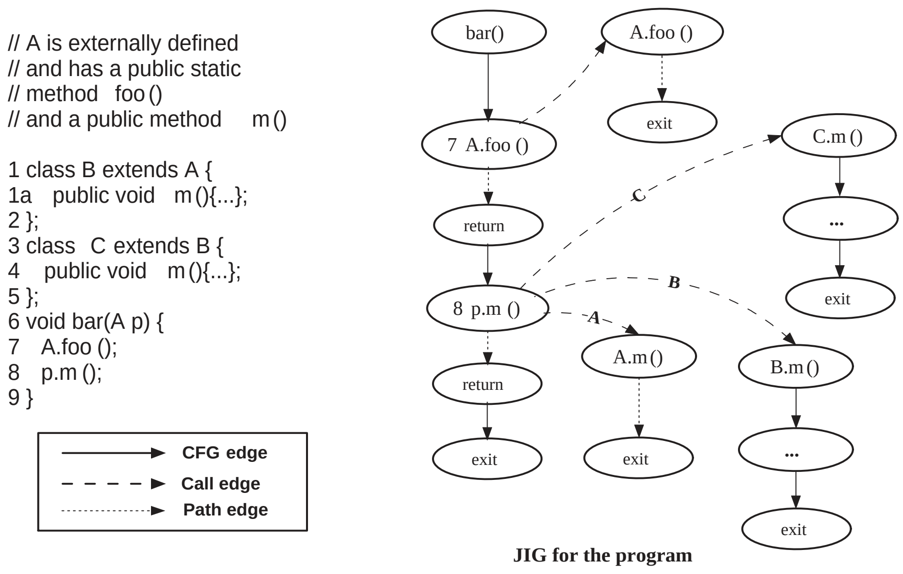


Figure 8: ClDG for class B of the program in Figure 6.

2.5.2 Java Interclass Graph

Intermediate representations such as *IPDG* and *ClDG* have been proposed in the context of C++ programs and do not satisfactorily model Java programs. Harrold et al. proposed an extended control flow model for Java programs called Java Interclass Graph (*JIG*) [41] that extends a *CFG* to capture the following features of a Java program:

- Variable and object type information - The variable or object type information is stored in a *JIG* node. The names of classes are represented using the full inheritance hierarchy which helps to easily detect any change to the inheritance tree for the class in the modified program.
- Internal and external methods of a class - Internal methods are represented in a *JIG* with an extended *CFG*. The extensions are: each call site is broken into a *call* and a *return* node. The *call* and *return* nodes are inter-connected with a *path* edge that represents the execution path through the called method.
  Since the source code is usually not available for externally-defined methods, these are represented in a *JIG* using collapsed *CFG*s.
- Calls to internal or external methods from internal methods - In a *JIG*, the *call* node is connected to the entry node of the called method with a *call* edge. There can only be one *call* edge if the method call is not polymorphic. For a polymorphic method call, the *call* node is connected to the entry node of each method that can be bound to the call. The class hierarchy analysis technique [21] can be used to identify all possible virtual call bindings.
  We illustrate the representation of method calls from internal methods in a *JIG* with the help of an example reported in [41]. Figure 9 shows a code snippet with calls to methods `foo()` and `m()`. In the program, class B extends class A (external to the program) and overrides the method `m()`. Class C extends class B and also overrides method `m()`. For polymorphic calls in a *JIG*, there exists an edge to all the methods for possible bindings. In the program, call to `A.foo()` from within function `bar()` represents a



```
// A is externally defined
// and has a public static
// method   foo()
// and a public method    m()

1 class B extends A {
1a   public void   m(){...};
2 };
3 class  C extends B {
4   public void   m(){...};
5 };
6 void bar(A p) {
7   A.foo();
8   p.m();
9 }
```

Figure 9: An example of method calls from internal methods in a *JIG*.

static binding. Therefore, in Figure 9, there exists only a single *call* edge between the nodes `7 A.foo()` and `A.foo()`. The method call `p.m()` is polymorphic, and there can be three possible bindings, one each for class A, B, and C. This is represented in the *JIG* by the three out edges from the node `8 p.m()`. Each outgoing edge connects to a possible method call to which it can bind during run-time.

– Calls to internal methods from external methods - There can be calls from externally-defined methods to internally-defined methods in Java due to inheritance and polymorphism. The external code is represented in a *JIG* as a node labeled *ECN* where *ECN* stands for *external code node*. For each internal class that is accessed from an external class, there is an outgoing edge from the *ECN* node to the *class entry* node of that internal class.
– Exception handling - A *JIG* represents a `try` statement with a *try* node. The code within the `try` block is represented as a *CFG*, and is connected with a control flow edge with the *try* node. Each `catch` statement is represented using a *catch* node, and the corresponding `catch` block is modeled using a *CFG*. The *catch* node is connected with the *CFG* using a control flow edge. The *try* node is connected to the *catch* node of the first `catch` block using a path edge labeled *exception*. A *finally* node and a *CFG* are used to represent the `finally` block of the `try` statement. Uncaught exceptions are modeled as *exceptional exit* nodes.

3 RTS Techniques for Procedural Programs

RTS techniques were first studied in the context of procedural programs [55, 56]. RTS for procedural programs is, therefore, an extensively researched topic, and many techniques have been proposed over the years [17, 56, 58, 37, 43, 92, 54, 76, 5, 80, 97, 98, 7, 10]. These techniques select relevant regression 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. Dataflow analysis-based techniques [43, 92, 44, 37]
2. Slicing-based techniques [7, 10, 2]
3. Firewall-based techniques [56, 58]
4. Differencing-based approaches [97, 98, 17]
5. Control flow analysis-based techniques [54, 80, 5]

In the following, we briefly discuss these different categories of RTS techniques and compare their effectiveness. We base our comparisons on the set of metrics introduced by Rothermel and Harrold [79]: safety, precision, efficiency, and generality. Rothermel and Harrold have presented a comprehensive survey of procedural RTS techniques in [79]. For the sake of completeness and continuity of the paper, we have included brief discussions on these techniques. We also discuss a few techniques [97, 98, 5] which were published after their work.

3.1 Dataflow Analysis-Based Techniques

In this subsection, we review RTS techniques [43, 92, 44, 37] based on dataflow analysis.

Dataflow analysis-based RTS techniques 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.

Harrold and Soffa [43] have proposed a dataflow coverage-based RTS technique that can be applied to analyze changes across multiple procedures. Their approach involves processing the dataflow information incrementally, i.e., process a single change, select test cases for that change, and update the dataflow information and test coverage information. The same process is repeated for all the changes one by one. In their approach, P is represented by a *CFG*, in which the nodes represent basic blocks. This reduces the size of the flow graph and makes graph analysis more efficient as compared to representing the individual program statements as nodes. Additional nodes are introduced in the flow graph to model global variables, function parameters, and return values of functions. Modifications to P usually result in changes to the basic blocks or the control flow structure of the program. The information in each node of the flow graph is extended to include the associated dataflow information for variables present in the node. For each variable definition in a node n, the node numbers of all the c-uses of the variable in the flow graph are stored in node n. The block numbers for all the p-uses of the variable are also stored in n. The information about the paths traversed when P is executed on each $t \in T$ is used to select test cases which exercise the modified def-use pairs for any variable, and are selected for retesting P'.

Dataflow-based RTS techniques reported in [43, 92, 44] usually carry out analysis either by processing the changes one by one and then incrementally updating the dataflow information for P', or compute the full dataflow information for P and P' and compare the differences between def-use pairs. Both these approaches require saving the dataflow information across testing sessions, or recompute them at the beginning of each testing session. The program slicing-based RTS technique proposed by Gupta et al. [37] 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 to be regression tested is sliced to select test cases that execute the affected def-use pairs.

3.1.1 Critical Evaluation

The techniques reported in [37, 43, 44] are based on computing dataflows in a program and are not able to determine the effect of program modifications that do not cause changes to the dataflow information [112]. The techniques also do not consider control dependencies among program elements for selecting regression test cases. As a result, these techniques are unsafe. Dataflow techniques are also imprecise because the presence of an 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 [79]. Examples illustrating the unsafe and imprecise nature of dataflow-based techniques are available in [79].

3.1.2 Slicing-Based Techniques

Agrawal et al. [2] have proposed a set of program slicing-based RTS techniques. The aim of these techniques is to select those test cases which can produce different outputs when executed with the modified program version P'. The authors define a slice with respect to a test case t as the set of program statements which are executed when P is executed with t. The authors have proposed four slicing techniques [2]: *execution* slice, *dynamic* slice, *relevant* slice, and *approximate relevant* slice. The RTS techniques proposed in [2] select a test case t for regression testing only if the slice of t computed using any one of the four approaches contains a statement modified in P'.

A PDG-based slicing approach for procedural programs was proposed by Bates and Horwitz [7]. However, the *PDG*-based slicing technique did not support inter-procedural regression testing. In [10], 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, iff they are executed the same number of times on any given input [10]. The concept of *common* execution patterns [10] has been introduced as an inter-procedural extension of the *equivalent* execution patterns proposed in [7]. Code elements are said to have a common execution pattern if they have the same equivalent execution pattern during some call to procedures. Common execution patterns capture the semantic differences among code elements [10]. 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.

3.1.3 Critical Evaluation

The program slicing-based RTS techniques proposed by Agrawal et al. [2] are unsafe [112]. The techniques are however precise [6] because they omit test cases that do not produce a different output. This eliminates the possibility of selecting non-modification-revealing test cases.

According to the studies reported by Rothermel and Harrold [79], the *PDG* [7] and *SDG*-based [10] slicing techniques are not safe when the changes to the modified program involve deletion of statements. The techniques are also imprecise. However, the *SDG* slicing-based RTS technique can be applied to select test cases for both intra- and inter-procedural modifications.

3.2 Module Level Firewall-Based Techniques

The firewall-based approach, first proposed by Leung and White [56, 58], 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. A 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 [56]. 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 included 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 from T which exercise the affected modules included in the firewall.

3.2.1 Critical Evaluation

The firewall technique is not safe as it does not select those test cases from outside the firewall that may execute the affected modules within the firewall [79]. 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 their relationships with other modules in the firewall, and hence limit the total amount of the source code that need to be analyzed. The firewall techniques handle RTS for inter-procedural program modifications but are not applicable for intra-procedural modifications [79].

3.3 Differencing-Based Techniques

In this subsection, we discuss RTS techniques [17, 97] that are based on analysis of the differences between the original and the modified programs.

3.3.1 Modified Code Entity-Based Technique

A modified code entity-based RTS technique was proposed by Chen et al. [17] 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 a global variable or a 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 P'.

3.3.2 Technique Based on Textual Differencing

Vokolos and Frankl [97, 98, 30] 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 [96, 97] 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 and then executed to generate the test coverage information. The test coverage information identifies 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.

3.3.3 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 [8, 79]. The technique proposed in [97] 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 [79], 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.

3.4 Control Flow Analysis-Based Techniques

A few RTS techniques [54, 80, 5] 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.

3.4.1 Cluster Identification Technique

The main concept used in the cluster identification technique proposed by Laski and Szermer [54] 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 *CFG*s (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 *CFG*s, 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.

3.4.2 Graph Walk-Based Technique

Rothermel and Harrold have proposed an RTS technique based on traversal of *CFG*s of the original and the modified programs [80]. This technique [80] is more efficient as compared to the graph walk-based RTS approaches based on dependence graph models [76, 78] proposed by the same authors. The approach proposed in [80] involves constructing *CFG*s 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 [80], a simultaneous depth-first traversal of the two *CFG*s G and G' is performed corresponding to each modified procedure in P and P'. The traversal is performed according to the 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.

3.4.3 DFA Model-Based Approach

Ball [5] has proposed a more precise RTS technique compared to [80] by modeling *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 states v_1 and v_2 of M. The two states are connected by a transition $v_1 \rightarrow_{BB(v)} v_2$, where $BB(v)$ is the basic block associated with node v in G.
2. The set of edges in G are modeled as state transitions. Therefore, an edge $m \rightarrow n$ in G represents a state transition $m_2 \rightarrow n_1$ in M.

These two conditions ensure that the DFA M accepts the set of all possible complete paths in G.

Ball introduced an intersection graph model for a pair of *CFG*s 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 technique uses edge coverage criterion as the basis for RTS analysis.

3.4.4 Critical Evaluation

The RTS techniques proposed in [80, 5, 54] are safe. Among the three techniques, the cluster identification technique is comparatively more imprecise because the test cases are selected based on whether they execute a cluster rather than the actually affected statements. The time complexity of the cluster identification technique [54] is bounded by the time required to compute the control scope of decision statements and is dependent on the input program size [79]. The techniques proposed in [80, 5] are the two most precise procedural RTS techniques. However, Ball's DFA-based approach is computationally more expensive than [80].

Ball has proposed another RTS technique [5] which uses path coverage criterion and is still more precise than the edge-coverage criterion proposed in [5]. 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

Class of RTS Techniques	References	Key Features	Merits	Demerits
Dataflow analysis-based techniques	[37, 43, 44, 92]	Based on dataflow and structural coverage criteria	Can analyze both intra- and inter-procedural modifications provided the modifications alter some def-use relations	Low on safety, imprecise
Slicing-based techniques	[7, 10, 2]	Based on slicing of programs or dependence graph models	Can analyze both intra- and inter-procedural modifications	Low on safety, imprecise, computationally more expensive than dataflow techniques
Module level firewall-based techniques	[56, 58]	Based on analyzing dependencies among modules	Comparatively more efficient as analysis of source code is limited to only modified modules	Low on safety, and highly imprecise
Modified code entity-based technique	[17]	Level of granularity can be adapted	Safe, and most efficient procedural RTS technique	Highly imprecise
Textual differencing-based technique	[97, 98, 30]	Based on textual differencing of C programs	Safe, and comparatively easy to implement a prototype	Imprecise, and difficult to adapt to other languages, maybe inefficient for large programs
Graph walk-based technique	[80]	Based on analysis of control flow models	Safe and most precise procedural RTS technique	Less efficient than [17, 56, 58]

Table 1: A comparison of RTS techniques for procedural programs.

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 affected 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 graph models of the program to identify the modifications [76, 80].

Table 1 summarizes the merits and demerits of the procedural RTS techniques discussed in Section 3. In column 3, we highlight the key features of each class of techniques, and summarize the merits and demerits in columns 4 and 5.

4 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 among various program elements, and make dependency analysis more difficult [104]. Moreover, in object-oriented development, reuse of existing libraries, class definitions, program executables (blackbox 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 [9, 68].

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

1. Firewall-based techniques [53, 47, 1, 48]
   (a) Class firewall technique [53]
   (b) Method level firewall technique [48]
2. Program model-based techniques [77, 82, 41, 73]
3. Design model-based techniques [4, 27, 69, 33, 14]

In the following, we briefly review the different classes of RTS techniques that have been proposed for object-oriented programs.

4.1 Firewall-Based Techniques

Firewall-based RTS techniques for object-oriented programs have been proposed by Kung et al. [53], Hsia et al. [47], Abdullah and White [1] and Jang et al. [48]. These techniques are based on the concept of a firewall defined originally by Leung and White [58] for procedural programs. The firewall techniques aim to identify the affected classes for the modified version of the software. A firewall can be defined as the set of all the affected classes that need to be retested. These techniques select all test cases which exercise at least one class from within the firewall.

4.1.1 Kung's Class Firewall Technique

Kung et al. [53] have proposed a firewall-based RTS technique for C++ programs. They have proposed the following three models to represent dependencies between various elements of a C++ program: Object Relation Diagram (*ORD*), Block Branch Diagram (*BBD*), and Object State Diagram (*OSD*). An *ORD* is a digraph that represents inheritance, aggregation and association relations, and captures the static dependencies between classes. An edge in an *ORD* is annotated with the type of relationship (inheritance, association, aggregation) that exists between the end nodes associated with that edge. A *BBD* represents the interface and the control structure of a method of a class, and the relationship of a class with the other classes in the program. An *OSD* is designed to capture the dynamic behavior of a class.

Changes to data items, methods and class definitions of the original program (if any) are identified by analyzing the three models corresponding to P and P'. The technique of Kung et al. [53] instruments P to collect information about which classes are exercised by which test cases. A class is potentially affected by a change to another class if it is either directly or indirectly related to the changed class through inheritance, aggregation or association relationships. The firewall for a class C is computed as the set of classes that are directly or transitively dependent on C (by virtue of relations such as inheritance, aggregation or association) as described in an *ORD*. When a class C is modified, the technique selects all the test cases that exercise one or more classes within the firewall for C.

Figure 10 shows an example *ORD* (along with test cases) adapted from [90]. In the figure, a solid arrow from one class to another indicates that the two classes are related by inheritance, aggregation or association relationships. The boundary (denoted by the dashed line) around the classes A, B, C and D depicts the firewall computed for class D. Whenever the class D is modified, classes A, B and C also need to be regression tested for they belong to the set of classes which constitute the firewall for class D. In Figure 10, a solid line from a test case to a class indicates that the test case is used to test the class (e.g., test case $TC1$ is used to validate classes D and F). Thus according to the firewall technique, only test cases $TC1$ and $TC2$ should be executed again after class D is modified since they exercise those classes within the firewall for D.

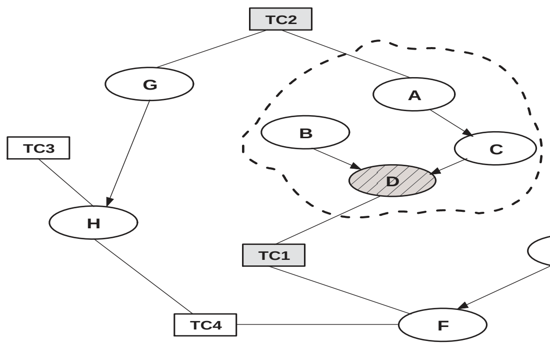


Figure 10: An example ORD and the firewall for class D.

4.1.2 Method-level Firewall Technique

Jang et al. [48] have proposed a change impact analysis approach to select regression test cases for C++ programs. While a class and a statement are considered as the units of testing in [53] and [77], the technique reported in [48] considers a method as the unit of retesting and aims to identify all affected methods. The authors have identified certain common types of modifications that are possible for a C++ program, and a method-level firewall is constructed for each modification to identify the impact of the changes.

4.1.3 Critical Evaluation

Firewall techniques are not safe because these techniques do not select test cases which may execute the affected modules from outside the firewall. These techniques are also imprecise since all test cases that execute a class in the firewall do not necessarily execute the affected parts of the code. For example, suppose that a class C is modified. Let another class D contain two methods `D::foo()` and `D::bar()`, of which the method `D::foo()` invokes the services provided by class C. Then, by the approach described in [53], class D is included in the firewall computed for C. Therefore, any test case which exercises D is included in the regression test suite. However, there might be test cases which exercise only the method `D::bar()` and hence could have been omitted from the regression test suite. The firewall-based approaches are however computationally more efficient and are preferred for RTS analysis of large programs. Moreover, the technique proposed in [48] is more efficient than [53] since this method aims to achieve a balance between the efficiency of class firewall-based technique [53], and the precision of more fine-grained approaches like [77].

4.2 Program Model-Based Techniques

In the following, we discuss different RTS techniques [77, 73, 82, 41, 65] that have been proposed for object-oriented programs and are based on an analysis of program models for selecting regression test cases.

4.2.1 Technique Based on Class Dependence Graphs

Rothermel and Harrold were one of the earliest to propose an RTS technique for object-oriented programs [77]. They have divided the problem of RTS for object-oriented programs into two parts: RTS of the application program, and RTS of the modified or derived classes. For RTS of the application program, the technique models the original program P and the modified program P' using *IPDG* models. However, it is difficult to use an *IPDG* for RTS of modified and derived classes because an *IPDG* models programs having a single entry point whereas a class can have multiple entry points. This problem can be overcome by treating the test routines as application programs and then applying the approach for RTS of application programs. However, this approach incurs a large overhead because it may be necessary to construct and traverse a *PDG* for each method of a class several times. Therefore, the original and the modified programs are modeled as *ClDG*s for RTS of modified and derived classes. The test coverage information is used to associate the predicate and statement nodes of the

ClDG models with each test case. Then, a technique similar to [80] is used to select regression test cases.

4.2.2 Technique Based on Extended Control Flow for C++

Rothermel et al. [82] have proposed an approach for RTS of C++ programs based on an analysis of the control flow representations of the original and the modified programs by extending the technique proposed in [80]. Since a *CFG* represents the control flow information of only a single method, the concepts of Inter-procedural Control Flow Graph (*ICFG*) and Class Control Flow Graph (*CCFG*) have been introduced to represent control flow of multi-function programs and object-oriented programs respectively. An $ICFG$ for program P is composed of *CFG*s for each method in P. Each call site in P is represented by a pair of nodes called *call* and *return* nodes [82]. Each *call* node is connected to the entry node of the called method by a call edge, and each exit node is connected to the *return* node of the calling method by a return edge. An *ICFG* is used to model programs having a single entry point, whereas a class can have multiple entry points [82]. A *CCFG* is used to model classes, and consists of individual *CFG*s for all methods of a class. Given the graph models for the original and the modified programs, the RTS algorithm [82] extends the graph walk-based approach [80] to traverse the models and select relevant regression test cases.

4.2.3 Technique Based on Extended Control Flow for Java

Harrold et al. were the first to develop a safe RTS technique [41] for Java programs based on control flow analysis. Their technique is an adaptation of the graph walk techniques proposed in [80, 82], and can handle various object-oriented features such as inheritance, polymorphism, dynamic binding and exception handling. Their method consists of three steps: constructing intermediate representations for the source programs, analyzing the graphs and determining the set of dangerous edges, and test case selection. Harrold et al. use a *JIG* representation for modeling Java programs.

The two *JIG*s constructed for the original and the modified programs are simultaneously traversed (depth-first) to identify dangerous edges. Finally, based on the test coverage information obtained through code instrumentation, the technique selects test cases that exercise the dangerous edges identified during graph traversal.

4.2.4 Partition-Based Techniques

Partition-based techniques are motivated by the need to combine the effectiveness of precise but expensive RTS techniques with techniques that work at a higher-level of abstraction and are relatively imprecise.

Partition-Based RTS Technique for Java Programs: Orso et al. have presented a novel two-phase partitioning approach for RTS of large Java programs [73]. Their technique works in two phases, called *partitioning* and *selection*. In the partitioning phase, the original and the modified programs are modeled as Interclass Relation Graphs (*IRG*) [73]. The two *IRG*s are analyzed to identify hierarchical, aggregation, and use relationships among classes and interfaces. Then, the set of classes and interfaces that have been changed are identified. The partition phase analyzes syntactical changes at the *statement* level and the *declaration* level. A change at the statement level consists of addition, deletion or modification of program statements. A declaration level change means modifications in the declaration of the type of a variable, addition or deletion of a method, change in the modifier list of an existing method, etc. The class dependency information along with the changes is used to identify at an abstract level the affected parts of the code. The set of affected classes and interfaces identified from the first phase constitutes a *partition* of the program.

In the selection phase, a more detailed analysis of the partitions are carried out. The selection phase builds *JIG* models representing the modified regions of code from the partitions. The *JIG* models are then analyzed using the RTS technique proposed by Harrold and Rothermel in [80]. An edge-level test selection criterion is used to select test cases which execute the affected parts of code.

Partition-Based RTS Technique for C# Programs: Mansour and Statieh [65] have proposed a two phase RTS technique targeted for the C# programs. Their RTS technique first constructs an Affected Class Diagram (*ACD*) based on the changes made to the modified program. An *ACD* represents modifications made at the level of a class, an interface, web or window services, and COM+ components. Their technique then uses a test coverage criterion based on the *ACD* to select a subset of test cases. An *ACD* models a program at a high level of abstraction. A more detailed analysis is then carried out by modeling the programs using C# Interclass Graphs (*CIG*). A C# Interclass Graph (*CIG*) is a control flow graph that captures all the affected methods in an *ACD*. The technique constructs *CIG* models for the original and the modified programs, and regression test cases are selected based on the graph walk techniques [41, 82].

4.2.5 Critical Evaluation

The program model-based object-oriented RTS techniques [77, 82] are safe and are more precise as compared to the firewall-based techniques [53, 47, 1, 48], but are less efficient. This is because of the high overhead incurred in inter-procedural dependence analysis for large software. The *ClDG* model-based technique is also less efficient than the technique proposed in [82] since it is based on an analysis of dependence graphs while the analysis in [82] is based on control flow relationships. Both the techniques reported in [82, 77] do not consider several other common features of object-oriented programs such as exception handling. These techniques [82, 77] can also be imprecise when testing affected polymorphic calls [41].

The techniques proposed in [41, 73] are safe RTS techniques for Java software, and are able to handle important object-oriented features of Java such as polymorphism, dynamic binding and exceptions. These techniques use a different method for capturing polymorphism than [82] which leads to a more precise selection of regression test cases. The two phase partition-based RTS technique proposed for Java programs [73] is comparatively more efficient than [41] because the analysis of the affected parts of the program is divided across two phases - a coarse-grained first phase and a more fine-grained second phase. This two-phased approach helps to limit the extent of code for which a minute low-level analysis is required.

Although the class firewall based technique [53] does not consider certain object-oriented features such as exceptions, the technique can still be extended for selecting regression test cases for a subset of the Java program features. The advantages of the firewall techniques are that they are comparatively more efficient than the program model-based techniques [41, 73], and can be applied for RTS of large programs. However as already discussed, the firewall techniques are unsafe and are relatively less precise than the techniques reported in [41, 73].

Mansour and Statieh's RTS technique [65] is a safe RTS technique tailored for C# programs. But the technique can be computationally expensive for large programs since the complexity of the algorithm is quadratic in the number of the *CIG* nodes.

4.3 Design Model-Based Techniques

Model-based testing of software has become very popular with the advent of the model-driven development paradigm. In the model-driven development (MDD) paradigm, a design model is usually refined to obtain the code. The widespread use of CASE tools for object-oriented system development ensures a close correspondence between a design model and its code. Hence, design models can effectively be used for RTS analysis of object-oriented programs [15].

Unified Modeling Language (UML) [12] is an ISO standard for representing analysis and design models of object-oriented programs. The following are some important advantages of UML-based regression testing [26]:

- *Traceability* - It is easier to maintain traceability between the design artifacts and the test cases than maintaining traceability between code and the test cases. It is also easier to identify changes between across different versions of design artifacts as compared to analyzing changes across code versions [15, 14].
- *Scalability* - Code-based regression testing becomes very expensive when applied to large programs. A model being a simplified representation of a code, model-based testing is comparatively more efficient.
- *Language independence* - Different parts of a software may be developed using different programming languages. It, therefore, is difficult to design and implement an RTS technique which can take into account parts developed using different programming languages during test case selection. A UML model-based RTS technique helps to overcome this shortcoming since it is independent of the implementation [14].

We now briefly discuss few UML model-based RTS techniques [4, 27, 33, 14, 69] that have been proposed in the literature.

4.3.1 RTS Based on Class and Sequence Models

Ali et al. have proposed an RTS technique based on analysis of UML class and sequence diagrams [4]. Their technique analyzes class and sequence diagrams at the level of class attributes and operations. Concurrency in sequence diagrams is captured by the use of asynchronous messages and parallel instructions which cannot be adequately represented by traditional *CFG* models [32]. Therefore, an extended control flow model called Concurrent Control Flow Graph (*CCFG*) has been introduced in [32]. Ali et al. extends the model-based control flow analysis proposed by Garousi et al. in [32] for regression testing based on UML design models by also including information available from class diagrams. A *CCFG* model is constructed for each sequence diagram. To model a sequence diagram invoking other sequence diagrams, the corresponding *CCFG*s are connected using control flow edges. The sequence and the corresponding class diagrams are analyzed and an extended concurrent control flow graph (*ECCFG*) is constructed to model the program. The information about which attributes of a class receive messages in a sequence diagram are derived from the corresponding class diagrams, and are represented in the *ECCFG*. The pre- and post-conditions of a method are also represented in an *ECCFG* by introducing new nodes. The *ECCFG* models for the original and the modified version of the application are then analyzed to find out the changes between program versions. This information is used to select regression test cases.

4.3.2 RTS Based on Class and State Machine Diagrams

Farooq et al. [27, 28] have presented a model-based RTS technique that uses information from UML 2.1 behavioral state machine and the structural class diagrams for test selection analysis. During software development, UML documents such as state machine and class diagrams describing the design and working of the software often undergo several modifications. The modifications made to one document may also affect other parts of the software.

Their proposed approach uses information from the modified class and state diagrams to find out the directly and indirectly affected elements of the model. For example, a state transition is considered to be affected if it uses any changed attribute or method of the corresponding class in its events, guard conditions, or actions. Those test cases that cover the modified transitions during execution are classified as retestable test cases.

4.3.3 RTS Based on Control Flow Analysis of Sequence Diagrams

Naslavsky and Richardson have proposed an RTS approach [69] based on control flow analysis of UML sequence diagrams for a MDD environment. The technique involves a model-based transformation from a sequence diagram to a *CFG*. The traceability between test cases and the sequence diagrams is used to determine which *CFG* elements are executed by each test case. The two *CFG*s corresponding to P and P' are then analyzed to find out the affected model elements, and the traceability information is used to select relevant regression test cases.

4.3.4 RTS Based on Use Case Diagrams

Gorthi et al. have proposed an RTS technique [33] based on UML use case diagrams. Their approach uses the concept of *behavioral* slicing which decomposes use cases into user actions followed by some computations and the output. Behavioral slicing helps in identifying changes made to the activity diagrams. Each node in an activity diagram is also assigned a criticality value to help increase the effectiveness of the selected test cases. Whenever the requirements are modified, the activity diagrams are also modified to reflect the changes to the system. The models for the original and the modified specifications are then analyzed to find out the *affected* paths in the diagram. The paths in the diagram that have one or more modified nodes are considered to be affected and the test cases which execute the affected paths are selected for regression testing.

4.3.5 RTS Based on UML Architectural and Design Models

Briand et al. have proposed an RTS approach based on analysis of UML design models [14]. Their approach assumes full traceability between the design model(s), the code and the test cases. The traceability between the design and test cases helps in associating the changes in the design models to the test cases which need to be executed to exercise the affected parts in design. Their approach involves analysis of use case, class and sequence diagrams. Their technique also assumes that there is a unique sequence diagram specifying possible object interactions along with each use case. The approach assumes that any pre- or post-conditions among classes are specified using the Object Constraint Language (OCL). Their analysis classifies test cases as obsolete, retestable and reusable test cases.

4.3.6 Critical Evaluation

In the following, we present a comparative evaluation of the UML-based RTS techniques that we discussed in subsection 4.3. The RTS evaluation framework proposed by Rothermel and Harrold [79] were originally for code-based techniques, and hence cannot be used in a straightforward manner to evaluate UML-based RTS techniques. For example, in the context of a UML-based RTS technique, a test case is modification-traversing iff it triggers a changed UML model element (e.g., messages for UML sequence diagrams).

The techniques proposed in [27, 28, 14] are safe with respect to the changes possible to the UML artifacts. An advantage of RTS based on analysis at a higher level of abstraction is improved efficiency as compared to code-based techniques. However, RTS based on UML design models are not as precise when compared to detailed code analysis-based techniques [26].

UML model-based RTS techniques require a close correspondence between the requirement artifacts, design models, code and the test cases, which may not always be possible in practice. Therefore, the applicability of these techniques is limited to a MDD environment.

4.4 Specification-Based RTS Techniques

In the industry, a practical difficulty in RTS is that the testers may not have access to the design models or the actual source code. In such scenarios, model-based or code-based analysis is not possible. This is especially true for COTS applications. Also, it is difficult to apply program analysis techniques and tools for many legacy software which have been developed using older programming languages (e.g., COBOL) for which there is a dearth of effective program analysis techniques [18]. Code-based techniques may also suffer from problems of scalability [16]. These limitations have motivated researchers to develop RTS techniques [18, 16] based on specifications which are usually available to the testers. In this context, it should be noted that although we have classified these techniques as a subtype of object-oriented RTS techniques, these techniques can be extended to a wider variety of programming paradigms such as component-based software.

4.4.1 Activity Diagram-Based Selection

Chen et al. have proposed a specification-based RTS technique [16] which uses UML activity diagrams for modeling the potentially affected requirements and system behavior. They have also classified the regression test cases that are to be selected into *target* and *safety* test cases. Target test cases are those that exercise the affected requirements, while safety test cases help achieve a pre-defined coverage target. The steps involved in selecting target test cases are as follows: A traceability matrix is created to capture the association between requirements and the test cases, i.e., which test cases exercise a particular requirement. The

Class of RTS Techniques	References	Key Features	Merits	Demerits
Firewall-based techniques	[53, 1, 48, 47]	Analyzes dependencies among modules	Computationally efficient	Unsafe and imprecise, need to be extended to handle certain object-oriented features such as exceptions
Program model-based techniques	[77]	Analysis is based on dependencies among class elements	Safe, and more precise than firewall-based techniques, is applicable for RTS of both modified classes, and classes derived from the modified classes, and application programs	Computationally more expensive than the firewall techniques
	[82]	Based on analysis of control flow models	Safe RTS technique for C++ programs, more efficient than [77]	Does not consider some common object-oriented constructs like exception handling, can be imprecise
	[41, 73]	Based on analysis of control flow models, two-phased technique [73]	Safe and precise RTS techniques for Java programs, two-phased technique is more computationally more efficient than [41]	Expensive for large programs with small changes because of fine-grained analysis
Design model-based techniques	[69, 4, 14, 27, 33]	Based on analysis of different UML design models (e.g, sequence, activity, use case diagrams), assumes that a traceability exists between the design models, the source code, and the test cases, suited to model-driven development environments	More efficient than program model-based approaches, suited for RTS of large programs, analysis is at a higher level of abstraction, and is independent of the implementation	Not safe, comparatively less precise than program model-based RTS techniques
Specification-based techniques	[18, 16]	Based on analysis of requirement models, assumes complete traceability from the specifications to test cases	More efficient than program model-based approaches, can be applied to systems with large test suites, techniques are platform-independent can be easily extended to a wide class of programs	Not safe, comparatively less precise than program model-based RTS techniques

Table 2: A comparison of RTS techniques for object-oriented programs.

modifications that are made to the original program P can result in a change of specification, or can be changes which are limited only to the code. In case when the changes are limited only to the code, the elements (nodes and edges) which are affected in the relevant activity diagrams are identified. Chen et al. have extended the RTS technique proposed in [82] to handle those modifications which lead to changes in the specifications also. Safety test cases are selected with an aim to mitigate risks. The idea is to more thoroughly test those parts of the code for which the probability of a fault being present and its cost (i.e., consequence of impact) is high [16].

4.4.2 Requirement Coverage Matrix-based Approach

Chittimalli and Harrold [18] have proposed a specification-based RTS approach. Their technique is essentially based on tracking which specifications are being tested by which test case from T. This information is represented as a *requirement coverage* matrix between the set of requirements and the test cases. The technique proposed in [73] has been used to identify the affected parts of the code, and subsequently the set of requirements that are affected due to changes are also identified. These are termed as *affected* requirements. The information from the requirement coverage matrix is used to select the test cases which exercise the affected requirements.

4.4.3 Critical Evaluation

The specification-based approaches are efficient as they do not depend on any static analysis of the source code. For the technique proposed in [18], the safety and precision of the approach is largely dependent on the quality and accuracy of the requirement coverage matrix. However, the safety of the approach is compromised by fact that dependence relationships existing among program elements cannot be completely and accurately captured by the requirement coverage matrix. Moreover, often in practical situations, code changes may be too trivial to affect the requirements, and the requirement coverage matrix may also be out of date.

We summarize the merits and demerits of the different RTS techniques applicable for object-oriented programs in Table 2. In column 3, we highlight the key features of each class of techniques, and summarize the merits and demerits in columns 4 and 5.

5 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 [31, 72]:

– 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, we classify the RTS techniques [72, 66, 67, 87, 31, 74, 115, 117, 116, 107] proposed for component-based software into the following classes:

1. Metacontent-based RTS approaches
   (a) Code coverage-based approach [72]
   (b) Enhanced change information-based approaches [66, 67]
2. Model-based techniques
   (a) UML model-based techniques [87, 107]
   (b) Component model-based technique [31]
   (c) Dynamic behavior and impact analysis using models [74]
3. Analysis of executable code [115, 116, 117]

In the following, we review a few prominent RTS techniques reported for component-based software.

5.1 Metacontent-Based RTS Approaches

The difficulty of inadequate information exchange between the component user (c-user) and the component developer (c-developer) during component-based software development can be overcome by sharing relevant component information required for RTS analysis. Orso et al. [71] have proposed the concept of content change information, called *component metacontent*, as a means of sharing information about the changes that a component undergoes across different versions. Different RTS techniques may define their own sets of required metacontents that need to be shared by the c-developers. Some examples of the type of information that are shared as metacontents range from the component version to more detailed like the coverage information of a particular test suite on the concerned component.

In the following, we discuss the different metacontent-based RTS techniques [72, 66, 67] reported in the literature.

5.1.1 Code Coverage-Based Approach

The code coverage-based RTS technique for component-based software was proposed by Orso et al. [72] and is based on existing procedural RTS techniques [17, 80, 81].

In case the c-users are unaware of the components that have undergone a change, then during RTS for the application code, any test case in which a method of the modified components is called is selected for regression testing. This can lead to selection of test cases unrelated to the specific change. A more precise selection of test cases can be made with information about the modifications made to the components. To enable a more precise selection of regression test cases, the technique [72] assumes the availability of the following metacontent information:

– Coverage information of the initial test suite on the component.
– Component version.
– Set of control flow edges affected due to the modifications to the component.

The c-developer supplies this information in the form of metadata and metamethods during the release of the modified component. The coverage information is based on the *CFG* edges traversed during execution of a test case. Based on the coverage information, test cases which execute the affected edges of the *CFG* are selected for regression testing according to the graph walk technique proposed by Rothermel and Harrold [80] .

5.1.2 Enhanced Change Information-Based Approaches

Mao et al. have observed [66] that the applicability of the technique suggested by Orso et al. [71, 72] is restricted due the fact that it requires a very detailed metacontent information to be provided by the c-developer. They have proposed RTS approaches [66, 67] which emphasize the availability of specific data from the c-developers to the c-users.

Change Information-Based Approach: A component provides services when invoked through its published APIs. Information is exchanged between components and the application program by means of the parameters of the published APIs, and the component variables which can directly be accessed from the application program (called published variables or PVs). Keeping this in view, the approach suggested by Mao and Lu [66] aims to identify changes at the method level for the modified components (to be performed by the c-developers). Invocation of methods within each component is modeled by constructing a Labeled Method Call Graph (*LMCG*) for each component. An *LMCG* for a component C is defined as $LMCG(C) = (V,\ E)$, where V represents the set of methods in the component (both published APIs and internal methods), and E represents the call relations among different methods of the component along with the pre-conditions required for a successful invocation. The enhanced change information (ECI) consists of the set of published APIs and the pre-conditions for invoking each published API. The ECI for the modified program statements in a component can be computed as follows: Suppose a certain method A invokes a method B in some component C. Then, the pre-condition for the method A invoking method B can be found out by analyzing $LMCG(C)$.

The ECI for the modified components are supplied to the c-users as files in a standard format (such as XML) along with the executable of the updated component. The c-users need to instrument the application source code to find out the values of the PVs and the parameters that are passed to each published API that is present in the ECI. For each test case, if the recorded values of the input parameters and the PVs satisfy the pre-condition for that published API, then the test case is selected for retesting the application integrated with the modified component.

Built-in Test Script-Based Approach: An RTS technique for component-based software using another level of information interchange between c-users and c-developers has been reported in [67]. This approach is motivated by the fact that it is only the c-developers who have detailed knowledge of the working of a component and the modifications effected to each of its versions. This technique proposes that the c-developers place test scripts in the component source code during modifications. The purpose of these test scripts is to gather information about the execution pattern of the component during execution of the test cases. This information helps to identify the test cases which cover the modified statements of the component.

A Method Call Graph (*MCG*) for a component C is defined as $MCG(C) = (V, E)$, where V represents the set of methods in the component (both published APIs and internal methods), and E represents call relations among the different methods in the component. The component APIs affected due to modifications to a component C are identified by the c-developers by analyzing the relationships between component methods using *MCG(C)*. Test functions for the affected methods are designed by the c-developers and are also published so as to facilitate selection of test cases by the c-users. The execution information gathered on invoking the test methods are used by the c-users to select test cases which execute the affected component methods.

5.1.3 Critical Evaluation

The metacontents-based approach [72] selects regression test cases by performing control flow analysis at the statement-level, and hence can be expensive for large programs. This problem can be overcome by using a coarser granularity during RTS analysis (method or class level) [66, 67]. The metacontent information in this case should be provided at the method or class levels.

5.2 Model-Based RTS Techniques

Model-based RTS techniques proposed for component-based software products are essentially refinements to model-based RTS techniques proposed for procedural and object-oriented programs. In the following, we briefly discuss a few model-based RTS techniques [87, 107, 31, 74] proposed for component-based software.

5.2.1 UML Model-Based RTS Techniques

Sajeev and Wibowo have proposed an RTS technique [87] for component-based software using UML and OCL models. Their technique assumes that the functionalities provided by the modified component is a superset of the functionalities provided by the original component, i.e., the new component version may include bug fixes and optimizations of the existing functionalities along with new functionalities that have been introduced. While UML is used to model the function call relations across components, OCL is used to represent the change information across component versions. The sequence of methods that are invoked by each test case is also tracked. All those test cases which either execute a directly modified method or a method which in turn invokes a directly or indirectly modified method are selected for regression testing.

Wu and Offutt have proposed another UML model-based RTS technique [107] for component-based software. In this technique, collaboration and sequence diagrams are used to analyze the control flow behavior of a component and how objects interact with each other through message description. The changes made to a modified component version will be reflected in a collaboration diagram as a change to a class method, or a change in the interaction sequences. The statechart diagrams are used to analyze the internal behavior of objects of a component. The class diagrams are used to identify the affected classes when the definition of one class is modified. For each modification in the collaboration diagram, the affected parts of the component are identified using control and data dependency analysis on the collaboration and the corresponding statechart diagrams. The test cases executing the affected parts are selected for regression testing.

5.2.2 Component Model-Based Technique

Gao et al. [31] have introduced several new models such as the Component Function Access Graph (*CFAG*), the Dy-

Class of RTS Techniques	References	Key Features	Merits	Demerits
Metacontent-based approaches	[72, 66, 67]	Assumes availability of metacontent information for RTS analysis	Metacontent information can be easily prepared, exchange of change information is simpler in [66, 67] than [72]	Emphasizes mutual collaboration between c-users and c-developers, control flow-based analysis in [72] may be expensive for large programs
Model-based	[87, 31, 74, 107]	Based on analysis of component models, models are passed as metadata	Computationally more efficient than the metacontent-based approaches	Proposed techniques are not safe and are less precise than metacontent-based approaches
Executable analysis-based	[115, 117, 116]	Novel approach based on reverse engineering the component binaries	Minimum dependence on the c-developers	Can be imprecise as selection analysis is at the function level, difficult to precisely identify changes among binaries

Table 3: A comparison of RTS techniques for component-based software.

namic CFAG (*DCFAG*), the Function Dependency Graph (*FDG*) and the Data-and-Function Dependency Graph (*DFDG*) to represent component API-based information at the system level. A *CFAG* models the static function call relationship of the component APIs, i.e., function calls from the application code to the component APIs. Each node in a *CFAG* represents a component API. An edge $e_i = (f_i, f_j)$ between two nodes (i.e., methods) f_i and f_j denotes that the second method is invoked after the first method. A *DCFAG* model provides a dynamic view of the function call sequences during the execution of a particular test case. Therefore, there can be many *DCFAG*s possible for a component C and a component API A_i. An *FDG* model is used to represent invocation dependencies between two functions in a component. A *DFDG* model is used to represent the define and use relationships among functions and variables. The technique [31] assumes that these models are supplied as metadata with new component releases.

For RTS analysis, the c-users require information about the modifications made to the component APIs. Changes to a component API are possible due to many reasons, such as modification to a function prototype, addition/deletion of parameters to a function, etc. These changes can be identified by comparing the revised component API specifications with the older version. However, there may be other dependency relations (control and data) which may indirectly affect APIs which are not themselves modified. These indirectly affected APIs are identified by analyzing the *FDG*s (for functions) and the *DFDG*s (for data variables).

Gao et al. have extended the firewall approach [53, 58] to identify the impact of component modifications on the other elements of the component (functions and variables). New types of firewalls are introduced to compute the set of affected functions due to changes in other APIs, functions or data variables of the component. For each modified element of a component, the firewall approach helps to identify the set of directly or indirectly affected component APIs. Regression test cases are then selected based on whether a test case executes the affected component APIs or not.

5.2.3 Dynamic Behavior and Impact Analysis Using Models

In [74], Pasala et al. have proposed an RTS technique for component-based software that analyzes the dynamic behavior (e.g., interaction of methods at runtime) of components to select test cases. This technique [74] is able to select regression test cases for components developed in .NET and Java. The information about the dynamic behavior is captured by executing the initial test suite and tracking the sequence of method invocations. These interactions are modeled as Functional Interaction Graphs (*FIG*). This step needs to be run once for every software application that is to be regression tested. To identify the affected methods in the newer component versions, the component binaries are reverse engineered to generate an intermediate code. The syntactical changes between different components are identified to track the directly affected methods, and the changes are then semantically analyzed to determine the set of indirectly affected methods. Once the complete set of affected methods are determined, the *FIG*s are then analyzed to select test cases relevant for regression testing.

5.2.4 Critical Evaluation

The RTS techniques proposed by Sajeev and Wibowo [87] and Wu and Offutt [107] are imprecise because these techniques perform RTS analysis at a high level of abstraction such as classes and methods. These techniques are also less safe than [31, 74] because they do not involve detailed dependency analysis at the statement level. However, the technique proposed by Gao et al. [31] is also not safe as it does not consider the effect of component modifications on the software as a whole and limits impact analysis to only the component level.

5.3 Analysis of Executable Code

Zheng et al. have proposed a family of RTS techniques [115, 116, 117] based on analysis of the executable code (binaries such as .dll, .lib) of the modified components. Their techniques are known as Integrated - Black-box Approach for Component Change Identification (*I-BACCI*), along with a version number to specify the exact approach. The *I-BACCI* technique uses the firewall approach for analysis of the glue code (application code which integrates the COTS components [116]). This technique utilizes the following information for RTS analysis:

– Binary files for the original and the modified versions of all the changed components.
– Glue code.
– Initial test suite developed for the glue code.

This technique involves reverse engineering the executables to identify the function definitions, code sections, etc. The extracted source code of the two versions of a component are then analyzed to find out the functions which have been modified between the two versions. Then, function call graphs (*FCG*) are constructed based on the identified function call relationships for the modified components. The *FCG*s are analyzed to find out the functions in the glue code which call published component functions. The glue code functions which directly or indirectly invoke the modified component functions are considered to be affected. The test cases that execute the affected functions in the glue code are selected for regression testing.

5.3.1 Critical Evaluation

The *I-BACCI* technique has the minimum dependency on information required from c-developers. However, an important limitation of the approaches proposed in [115, 117, 116] is precise identification of the changes between the component versions by reverse engineering the executables. For example, during RTS analysis, the technique may fail to identify and ignore all trivial differences that are introduced in the component executables due to build configurations, build and target platforms, etc. These techniques are also not precise since they do not perform a statement-level analysis, and affected code elements are identified at the level of functions. Therefore, there might be glue code functions which invoke modified published functions of the component but do not actually execute the modified program statements in the published function. Test cases which exercise such glue code functions can in fact be safely ignored during regression testing.

We summarize the important characteristics of RTS techniques proposed for component-based software in Table 3. The key features, merits and demerits of each RTS technique as compared to similar techniques proposed for component-based software have been presented in columns 3, 4 and 5 respectively.

6 RTS Techniques for Database Applications

A large number of database applications are currently in use. These applications are usually composed of several components contributing to an increase in their sophistication [38]. Database applications also need to be frequently modified due to different requirements, e.g., change in components, growing number of users and data, etc. In this context, regression testing of database applications is an important activity.

The requirements and challenges in regression test selection of database applications are different from the classes of programs that we have discussed so far. Regression test selection of database applications need to take into account the following features:

– RTS techniques for other classes of programs implicitly assume that the test cases are independent of each other and can be executed in any order. This assumption is not valid for database applications as the output of a test case may change the *database state*, in the process affecting the execution of other test cases. Therefore, in addition to the global program state, the states of the database need to considered during RTS for database applications.
– The state of the database may have to be *reset*, i.e., restore the initial database configuration, many times during regression testing. Resetting of a database is acknowledged to be an expensive activity both in terms of cost and time [38].
– Database languages support features such as structured queries, integrity constraints, exception handling and table triggers, which complicate impact analysis of the modified parts of the program. For example, firing of triggers can create implicit intermodular control dependencies [39].

The traditional notions of safety and dependencies cannot be applied in regression testing of database applications because those techniques were developed for *stateless* applications. In this context, a few RTS techniques have been proposed for database applications [105, 39]. We briefly review these techniques in this section.

6.1 Two Phase RTS Technique for SQL-Based Systems

Haraty et al. [39] have proposed a two-phase technique for RTS of structured query language (SQL) based database applications. Apart from traditional control and data dependencies among elements in a database application, Haraty et al. have identified the following aspects that need to be considered:

– Dataflow dependencies - Dataflow dependencies can arise among database modules due to usage of tables across modules.
– Component dependencies - These arise among different database modules due to firing of table triggers, modifications to tables or views, or due to modifications to SQL statements. Component dependencies are transitive in nature.

– Exception handling - Raising of exceptions can affect control flow relationships, which need to be taken into account during RTS analysis.

Haraty et al. have proposed a control flow model of SQL statements where a node in the *CFG* represents an SQL statement. Their modeling technique also represents possible changes in control flow that arise due to exceptions. They have identified two types of changes that are possible in a database application:

1. Code changes - These are possible additions, deletions, and modifications to SQL statements within a database module.
2. Database component changes - These include changes to the database component definition itself, e.g., changes in the interface.

Their technique determines modifications between the two versions of the program and identifies potential areas of the code where the changes can impact. To identify the set of affected components due to modifications, Haraty et al. have used the concept of a *component firewall*. A database module is considered to be affected and is, therefore, included in the component firewall if any one of the following conditions holds:

– The definition of the module is modified.
– The module is deleted.
– The module is data or control dependent on another modified or deleted module.
– The module becomes dependent on some other module due to modifications.

The first step in constructing the component firewall is to identify the directly changed modules. Then, the transitive closure of the directly changed modules is computed to find the set of all potentially affected database modules. In the second phase, relevant test cases are selected based on any one of the two algorithms: one is based on traversal of *CFG*s and the other is based on firewalls. The firewall-based algorithm is based on analyzing the module-level dependencies among database components.

6.1.1 Critical Evaluation

Experimental studies show that the firewall-based RTS may ignore omitting potential modification-revealing test cases, and is therefore unsafe [105]. The firewall-based technique is also imprecise for reasons similar to the firewall approaches proposed for traditional programs.

6.2 CFG-Based Safe RTS Technique

Willmor and Embury [105] have extended the safe control flow analysis-based RTS algorithm proposed by Rothermel and Harrold [80] for procedural programs to database applications. RTS based on only definition-use relationships is not safe for database applications. This is because it is possible for an instruction to write some data to the database that will later be read by a program statement that precedes the earlier instruction in some execution path [105]. Therefore, the authors have introduced the concept of *database dependencies* to capture the additional dependencies that arise among elements in a database program. A statement is called *database dependent* if the statement can update the database, and has been modified in P' such it can affect the database state. Statements which are dependent on database dependent statements are considered affected, and the test cases that execute these statements are called database-dependent test cases. Based on these additional dependencies, Willmor and Embury's technique selects modification-revealing test cases with respect to the program state, and database-dependent test cases with respect to the database state.

6.2.1 Critical Evaluation

The technique proposed in [105] is the first safe RTS technique for database applications. However, the approach can be imprecise. Consider the case in which a statement that adds new tuples to a table is modified so that it is capable of adding only a subset of the tuples that could be added by the original statement. In such circumstances, no new faults can be introduced in the modified code due to the change which cannot already be detected in the original program. Therefore, those test cases which test code that can potentially be affected by this change need not be selected.

7 RTS Techniques for Web Applications and Services

Web applications and services are dynamic in nature and constantly evolve. They are frequently updated and, therefore, need to be regression tested to verify the correctness of the unmodified functionalities. In the following, we discuss the main features of web applications and services that need to be taken into account during RTS:

– Web applications are composed of server side services, user applications, and middleware. A safe RTS technique for web applications should consider all types of dependencies that can arise in the different layers of the web application under test.
– Web applications and services are inherently distributed in nature and are loosely-coupled.
– Web services are usually composed of and make use of other services. Therefore, the dependencies arising due to a modification to another service also need to be considered during RTS.

A difference in the nature of composition of component-based and web applications is that a component-based software physically integrates a component. Therefore, it is up to the component user to upgrade to newer releases of the component. However, a web service can be updated as deemed fit by the concerned developer and is not owned and neither controlled by the application developers, thereby further complicating RTS of web applications.

Recently, many RTS techniques have been proposed for web applications [86, 93, 61, 110, 85]. We briefly review these techniques in this section.

7.1 RTS for Web Applications Based on Slicing

Xu et al. have proposed an RTS technique for web applications based on slicing [110]. They assume that web applications consist of multiple static HTML pages and programs running on the server side. The types of changes that an HTML page can undergo can be divided into the following basic classes: insertion of a page element (e.g., anchor, hyperlink, etc.), deletion of a page element, insertion of a page and deletion of a page. More complex changes are decomposed into a combination of these basic modifications. In this context, it needs to be noted that certain kinds of changes, such as formatting related changes, cannot affect other web pages. The different HTML pages in an web application can be data or *hyperlink* dependent on each other. The dependencies that can arise are further divided into direct and indirect dependencies. For example, if a hyperlink is inserted, then it needs to be checked that the link is working if the target page is part of the website. Indirect dependencies arising due to definition-usage relationships of variables are analyzed using traditional techniques. Then, the slice is computed on the indirect data dependencies on an extended *SDG* model of the web application which is to be regression tested. Test cases that execute the potentially affected web elements are selected for regression testing.

7.1.1 Critical Evaluation

The details of the slicing technique used for identifying potentially affected web elements have not been provided in [110]. However, it can be inferred that the technique will suffer from drawbacks similar to slicing *SDG*s for procedural programs. The technique is, however, precise in selecting relevant test cases for the types of changes that have been considered.

7.2 RTS Based on System Models

Tarhini et al. have proposed a safe RTS technique for web services-based applications [93]. The technique defines web services as self-contained component-based applications residing at separate locations and communicating using XML-encoded messages using SOAP interfaces. The communication using message exchange may also be time-constrained. The services provided by a web service are shared using WSDL specifications.

The authors have modeled a web application in two hierarchical levels to avoid state explosion. In the first level, the interaction of the components with the main application is modeled using a Timed Labeled Transition System (*TLTS*). Each node in a *TLTS* represents a component, and an edge joining two nodes represents a transition between the two components. The internal behavior of each component is modeled in the second level. Each node in the second-level *TLTS* represents a state of the component that is being modeled. The authors have proposed an RTS technique which selects all relevant test cases that test the side-effects of adding, removing or fixing an operation or a timing constraint in an existing component based on an analysis of the constructed two-level *TLTS* models. The approach for selecting relevant regression test cases is as follows: Construct the *TLTS* for the modified web service; generate test cases for testing the *TLTS* model corresponding to the modified web service; find the difference between the initial test suite and the generated test suite. The differential set of test cases are selected for regression testing.

7.2.1 Critical Evaluation

The technique in [93] is safe because it selects every test case that produces a different behavior in the modified system. However, this technique cannot strictly be considered as a pure RTS technique because the analysis involves generation of test cases as an intermediate step.

7.3 Control Flow-Based RTS Techniques

Ruth et al. [86, 85] and Lin et al. [61] have proposed safe RTS techniques for web services based on analysis of control flow models. We discuss these techniques in the following.

The RTS technique proposed by Ruth et al. [86, 85] is a gray-box technique since it is difficult to carry out white-box regression testing for web services because often the source code for the components may not be available with the web service developer. It is a gray-box technique because it does not require the source code of the web services. Instead, their approach assumes that the component web service providers would provide the following information as metadata along with a service release: WSDL specification, a set of test cases, *CFG*s for the web services, and test coverage information.

Their technique requires that each procedure in a web service is modeled as a *CFG* at the service developer side. Their technique also assumes that the method calls to other services are decided statically. The *CFG*s for all the individual procedures are then combined to form a global *CFG*. When a web service is modified, then a global *CFG* is also constructed for the modified web service. Each node in a *CFG* stores a hash code of the corresponding statement. The authors have extended the graph traversal algorithm proposed in [80] to simultaneously traverse the global *CFG*s for the original and the modified programs and identify the nodes of the graph which are changed. All control flow edges which can be reached from the modified nodes are marked as *dangerous*. The changes made to the modified web service can be identified from a difference in the hash values without requiring analysis of the source code. The test cases which execute the dangerous edges are selected for regression testing.

Lin et al. have proposed a safe RTS technique [61] for Java web services based on code transformation. Their technique models Java code at the client side and the service side as a single *combined* program. The services and the interfaces provided by the web service are available from the WSDL specifications. The technique simulates message passing between the client application and the web service through local proxy objects. The merged program is then modeled as a *JIG* which has the same structure as the original application. Once modeling of the original and the modified web service is complete, the algorithm proposed in [41] is used to select relevant regression test cases.

7.3.1 Critical Evaluation

The control flow-based techniques proposed in [61, 85, 86] have advantages and disadvantages that are comparable to the control flow-based RTS techniques proposed for procedural programs. These techniques are safe, and comparatively more precise and at the same time less efficient than the technique proposed in [93]. The technique proposed by Ruth et al. [86, 85] is also similar to component-based RTS techniques [72, 66, 67] because it relies on metacontent information supplied by the web services developers.

8 RTS Techniques for Aspect-Oriented Programs

Aspect-oriented software development paradigm is an emerging methodology that aims to modularize software development by isolating low priority and auxiliary functionalities from the application's main business logic. In the traditional programming model, it is up to the programmer to manage and interleave other auxiliary issues (called as *concerns*) into the main application code. Concerns which are spread across multiple modules are called *crosscutting* concerns. For example, for a programmer who is developing a module for a banking software, related issues such as logging, performance, security, authentication, exception handling, etc. are examples of crosscutting concerns. Aspect-oriented programming (AOP) allows programmers to relegate these secondary crosscutting concerns to stand-alone modules called *aspects*. AOP has also introduced new terminologies such as *advice*, *pointcut*, *introduction*, *join points*, *shadow*.

AOP has been adopted for many object-oriented programming languages and AOP languages such as AspectJ has gained considerable popularity among the Java developer community. Introduction of aspects usually change the behavior of the original Java program. Therefore, AspectJ programs also need to be thoroughly regression tested after some modifications. In this section we discuss the proposed RTS techniques for AspectJ programs [114, 109] since AspectJ is the most widely used aspect-oriented language [50].

8.1 RTS of AspectJ Programs using Control Flow Models

Zhao et al. [114] proposed an RTS technique for AspectJ programs by extending the work of Harrold et al. [41]. They have proposed a System Control Flow Graph (*SCFG*) and an Aspect Control Flow Graph (*ACFG*) to model an AspectJ program. The authors have introduced additional nodes and edges, such as *join point* vertex, in an *SCFG* to model AspectJ constructs. Each individual aspect in a program is represented using an *ACFG*. An *ACFG* is composed of individual *CFG*s which represent static control flow relationships that exist among advice, inter-type members, and methods of an aspect. An *aspect entry* vertex is used to represent entry to the aspect. An *aspect membership* edge is used to connect the *aspect entry* vertex to different possible aspect members such as advice, inter-type members, pointcuts, or methods. Their model is also able to represent interactions between aspects and classes.

Once the *SCFG* graphs have been constructed for the original and modified pair of AspectJ programs, the depth-first search technique proposed in [41] is used to identify the dangerous edges in the graph. The test cases that execute the dangerous edges are selected for regression testing.

8.2 RTS for AspectJ Programs Based on Extended JIG

Xu and Rountev [109] have proposed a safe intermediate graph representation-based RTS technique for AspectJ programs. They have first proposed a control flow-based graph model for AspectJ programs named AspectJ Inter-module Graph (*AJIG*) which is an extension of a *JIG*. An *AJIG* consists of *CFG*s that model control flow relationships within Java classes similar to *JIG*s, within aspects, and across boundaries between aspects and classes through non-advice method calls. An *AJIG* also consists of interaction graphs that model interactions between methods and advices at certain join points. The following types of interactions are modeled by an *AJIG*:

- A method call from a Java class method to another class method or an aspect method.
- A method call from an advice to a class method or an aspect method.
- A method call from an aspect method to a class or an aspect method.

Relevant regression test cases are selected by comparing the *AJIG* graphs for P and P'. The authors have extended the graph traversal algorithm proposed in [41]. Their two-phase algorithm also handles situations where the destination nodes for a pair of edges that are compared are statement shadows. In the first phase, the invocation order of the advices are compared using the two interaction graphs corresponding to P and P'. The output of the first phase is a set of *dangerous* edges in P that are changed in P' and a set of advices whose invocation order remains the same and whose bodies need to be further inspected in the second phase. In the second phase, the *CFG*s for each advice identified in the first phase are traversed to identify dangerous edges. The test cases executing the set of dangerous edges are selected for regression testing.

8.3 Critical Evaluation

The technique proposed by Zhao et al. [114] ignores situations where multiple advices apply at a shadow, or where there can be dynamic advices. Their proposed graph model cannot represent these suitably, and hence, may miss out on selecting potentially fault-revealing test cases.

The RTS technique proposed by Xu and Rountev is a safe RTS technique for AspectJ programs and overcomes the drawbacks inherent in the RTS technique proposed in [114].

9 RTS Techniques for Embedded Programs

During the last decade, there has been a rapid surge in the usage and reach of embedded applications. A variety of embedded applications have infiltrated almost every facet of our daily lives. Over the years, embedded applications are becoming more and more sophisticated and are being extensively used in real-time and safety critical applications. The domains where embedded applications are being heavily used at present include entertainment, automobiles, life-saving medical instruments, nuclear power stations, and defense warfare. As a result, extremely reliable operation of these applications has become an essential necessity.

Regression testing is a challenging task in the life cycle of an embedded software [88, 70, 91]. Embedded applications are often composed of concurrent, co-operating tasks, many of which may be real-time in nature. Issues such as concurrent execution and deadlines of tasks add new dimensions to the complexity of testing embedded programs. For example, concurrent tasks in an embedded program may get scheduled differently even when the same set of events occur with minor alterations to their timing of occurrence. This can cause unrepeatable test results. Furthermore, an error which is not manifested in one test case may be exposed by another test case having the same inputs, same start state and executing the same functions but having a different timing behavior. Embedded systems usually accept inputs (in the form of events) from the environment concurrently and asynchronously. Since it is not always possible to predict the exact input pattern, therefore, the behavior of an embedded system needs to be tested for all possible input combinations. This may necessitate testing of embedded software using a large number of test cases. Moreover, the high cost of execution of test cases for embedded programs makes minimization of the costs incurred in regression testing highly desirable [36]. Selection of a set of safe test cases for embedded applications has, therefore, been acknowledged as an important research problem [118].

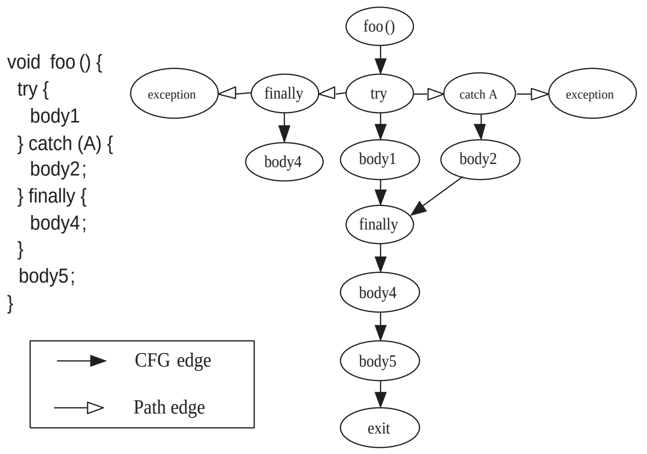
The RTS techniques for traditional programs cannot satisfactorily be used to select regression test cases for embedded programs, since embedded programs have many features that are radically different from traditional programs. A few examples of these features are the following:

- A real-time task is usually associated with a deadline by which it needs to produce the required results. Thus, test cases validating the timing aspects of a modified feature need to included. Therefore, an RTS approach based solely on analysis of data and control dependency aspects alone would be unsafe. In this context, analysis of control flow information for checking the timing properties has been advocated by many researchers [102, 45].
- Embedded programs are concurrent and event-driven. The dependencies arising due to these features can result in subtle bugs in the programs, and need to be specifically regression tested.
- Embedded programs often use explicit exception handling mechanisms. This is especially true for safety-critical applications where error situations need to be properly handled. Throwing an exception alters the normal flow of control in a program. Hence, all affected control flow paths in the program need to be regression tested.

In the literature, we could find only one study by Biswas et al. [11] related to RTS of embedded applications. In [11], the authors have proposed an *EClDG* model for representing embedded programs. An *EClDG* model is an extension of a *ClDG* and represents both control and data dependencies that exist among program elements. An *EClDG* also contains control flow edges to represent tasks in an embedded program which are essentially a sequential execution of program statements. An *entry* node in an *EClDG* model associated with each task in the corresponding embedded program also stores the priority and the criticality information related to the task. Some of the additional features that are represented in an *EClDG* are exception handling, and information available from UML design models, such as, object states and state transitions. Regression test cases are selected by slicing the *EClDG* model. Each point of change between the original (P) and the modified (P') program acts as a slicing criterion. Identification of which model element is executed by each test case is determined by instrumenting the source code. The test cases that execute the potentially affected model elements are selected for regression testing.

Class of RTS Techniques	References	Key Features	Merits	Demerits
Database	[105, 39]	RTS techniques need to consider database states	Willmor and Embury's technique [105] is safe	Proposed techniques are imprecise
Web applications	[86, 93, 61, 110, 85]	Analysis cannot rely on the availability of the source code of web services	Techniques proposed in [93, 86, 85, 61] are safe, system model-based approach [93] is more efficient than [86, 85, 61]	Techniques can be imprecise, and depend on metacontent information
AspectJ	[114, 109]	Needs to take into account the dependencies that arise due to *pointcut*, *join* points, etc.	Technique reported in [109] is safe	Control flow-based techniques may be computationally expensive, cannot be directly adapted for higher-level analysis

Table 4: A comparison of RTS techniques proposed for database, web, and aspectj programs.



```
void foo () {
  try {
    body1
  } catch (A) {
    body2;
  } finally {
    body4;
  }
  body5;
}
```

Figure 11: Modeling exceptions in the RTS technique proposed in [51].

10 Other RTS Techniques

In this section, we discuss a few RTS techniques proposed for BPEL programs [59, 63, 101] and programs developed in .Net framework [51]

10.1 RTS Technique for .Net Programs

In recent times, many virtual machine environments have been proposed such as Java and Microsoft .Net framework. In a virtual machine (VM) environment, the program is compiled into a platform-independent intermediate code. The advantage of such virtual machine environment is that it introduces a layer of abstraction and hides the low-level intricacies of the target architecture. The VM environment can also introduce check points to enhance performance, security, etc. of the application code. In the following, we discuss a safe RTS technique proposed for programs developed in Microsoft .Net framework.

Koju [51] have presented a safe RTS technique for programs developed in .Net framework. Since Microsoft .Net framework supports many programming languages such as Visual Basic, C++, C#, an RTS technique based on source-code analysis would require to take into account the features of all the .Net framework supported programming languages. The authors have avoided this problem by selecting regression test cases based on an analysis of the intermediate code, which is in Microsoft Intermediate Language (MSIL). Their technique is based on the graph walk-based RTS technique proposed by Harrold [41] for Java programs. However, the *JIG* model proposed in [41] cannot model .Net specific features such as *delegates*. The authors have also proposed a more efficient and precise way of analyzing the dependencies introduced due to class hierarchies and exceptions compared to [41]. The improved analysis of class hierarchies is applied to model method calls from code internal and external to the application under test. The modeling of exceptions is improved by representing the `catch` and `finally` block on the opposite sides of a `try` block. An example of the exception modeling technique proposed by Koju et al. [51] is shown in Figure 11. The figure shows a partial *JIG* modeling the exception handling code shown in Figure 11.

The important steps in the RTS technique proposed in [51] are as follows:

- Construct the extended *JIG* models corresponding to the MSIL code for the original and the modified programs.
- Instrument the original source program and execute the instrumented program with the initial test suite to generate the test coverage information.
- Traverse the two extended *JIG* models to identify dangerous edges. The test cases executing the dangerous edges are selected for regression testing.

This technique is safe for RTS based on MSIL languages, and is more precise than [41] because of the improved representation for exception handling and the dependencies arising due to class hierarchies. In spite of our best efforts, this is the only technique that we could find in the literature for RTS in a virtual machine framework.

10.2 RTS Techniques for BPEL Programs

We have pointed out in subsection 2.3 that SOA-based development is increasingly being adopted in different services industries. Business process execution language

(BPEL) is a part of the SOA standards, and is popularly being used to develop business process and composite services. Composite services in BPEL are composed of a process, an interface described in WSDL, and component services that interact with the process. The component services can be elementary services or composed of other elementary services. Modifications to a BPEL composite service can take place due to different reasons such as modifications to the process or the interface, replacement of a service with another service, etc. Whenever a BPEL composite service is modified, it becomes necessary to select regression test cases to test the unmodified parts of the program. In the following, we briefly discuss the control flow analysis-based RTS techniques proposed by Li et al. [59, 63, 101].

A BPEL flow graph [113] can only capture the control flow relations in a BPEL process. Therefore, it cannot be used to model BPEL composite services. In view of this, Li et al. [59] have proposed an Xtended BPEL Flow Graph (*XBFG*) to model BPEL composite services. Along with the business process, an *XBFG* is also able to model composite services and the message interactions between the process and the composite services. The technique constructs *XBFG* models for both the original and the modified BPEL composite services. The types of changes possible between two BPEL composite services are assumed to be: process change, binding change, change in the path conditions, and interface change. The technique then compares the test paths between the two *XBFG*s to find out the model elements influenced by the process and the binding changes. The paths in the *XBFG* models which are affected due to the changes are identified, and relevant regression test cases are selected to test the affected paths.

11 Conclusion and Future Research Directions

It is acknowledged that RTS techniques which analyze modifications at a finer level of granularity (e.g., program statements) are more precise than techniques which perform analysis at a comparatively higher level of abstraction (e.g., design models). Rothermel and Harrold have shown that the problem of designing precise RTS techniques is PSPACE-hard [80]. Moreover, the extensive computations for a fine-grained analysis (e.g., graph walk-based techniques for procedural/object-oriented programs) make these techniques more expensive, less efficient, and less scalable compared to the coarse-grained approaches. This is an important trade-off that needs to be considered while selecting a suitable RTS technique. After all, selection of an RTS technique makes sense only if the cost of test selection is less than the difference in cost between running the entire test suite and the selected test suite [57].

Modern commercial software products are becoming increasingly large and complex, and are usually tested using thousands of test cases. Therefore, to obtain further savings in regression testing effort, researchers need to consider the following issues:

- With the trend of increasing application size, an RTS technique should scale to very large programs having code sizes of the order of millions of KLOC. For modern large software systems, scalability is an important issue. Therefore, an interesting direction of research would be to investigate compositional and summary-based approaches to RTS.
- The RTS technique should take into account all possible relationships depending on the targeted class of programs while selecting test cases, i.e., it should be a safe technique for that class of programs.

Model-based regression testing: In view of the fact that static analysis of large software systems is computationally expensive, model-based RTS techniques appear to be a promising approach that not only scales well, but is more efficient [112]. Furthermore, of late MDD has been receiving a lot of attention. In MDD, there exists a close relationship between the design model(s) and code in the sense that any change to the model gets reflected in the code and vice versa. Therefore, instead of performing RTS on code, test selection could be automatically performed based on design models. Model-based RTS can also help to take into consideration several aspects of program behavior (like state transitions, message paths, task criticality, etc.) that are not easily identified from static code analysis.

Improved RTS tool support: In future, the reported work on RTS should gradually shift from theoretical research to tool implementations. It has been pointed out in several studies [35, 36, 112] that the current tool support for automated RTS is rather poor. Therefore, concerted effort should be directed towards developing integrated RTS tools using capture-and-replay mechanisms.

Synthesized regression testing techniques: Most of the RTS techniques reported in the literature are either code-based or model-based. Since both these approaches have their own unique advantages, these approaches can possibly be meaningfully synthesized and this issue deserves further investigation. For example, the analysis performed in a code-based RTS technique can be made more effective by using the information available from the UML design models, SRS documents, etc.

Yoo and Harman [112] have pointed out that real-world regression testing needs to select test cases keeping in mind multiple objectives such as, the number of test cases executed, cost involved in testing, code coverage achieved, time available for testing, etc. However, most of the reported work on multi-objective regression testing is in the fields of test suite minimization and prioritization [111, 99]. An interesting avenue of research could be to merge regression test selection techniques with either minimization or prioritization approaches. In such a synthesized approach, the regression test suite first selected by a structural RTS technique can then be further minimized/prioritized. Regression testing using such a synthesized approach can help take into account multiple objectives during testing, and can potentially help achieve further savings in regression test effort without compromising the thoroughness of testing.

RTS techniques for other domains: Increased usage of real-time embedded products in safety-critical applications has resulted in greater emphasis being placed on the quality of the code. The high costs and complexities involved in carrying out regression testing of these products act as an added incentive for developing improved RTS techniques for these programs. However, not much research work has so far been reported on investigations into effective RTS for embedded, real-time and safety-critical software, though it appears to be a promising avenue for research.

For discrete control applications, industry practitioner's usually use UML models whereas for hybrid control applications, MATLAB Simulink/Stateflow models [94] are popular. In this context, suitable RTS techniques are needed for hybrid control applications and reactive software.

Moreover, as pointed out by Yoo and Harman [112], more detailed investigation is required to study the effectiveness of RTS techniques for testing non-functional requirements.

References

[1] 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.

[2] H. Agrawal, J. Horgan, E. Krauser, and S. London. Incremental regression testing. In *IEEE International Conference on Software Maintenance*, pages 348–357, 1993.

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

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

[11] 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.

[12] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley, 2nd edition, 2005.

[13] Mustafa Bozkurt, Mark Harman, and Youssef Hassoun. Testing web services: A survey. Technical Report TR-10-01, Kings College London, 2010.

[14] 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.

[15] L. Briand, Y. Labiche, and G. Soccar. Automating impact analysis and regression test selection based on UML designs. In *Proceedings of the International Conference on Software Maintenance (ICSM'02)*, pages 252–261, 2002.

[16] Y. Chen, R. Probert, and D. Sims. Specification-based regression test selection with risk analysis. In *CASCON '02: Proceedings of the 2002 conference of the Centre for Advanced Studies on Collaborative research*, page 1, 2002.

[17] 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.

[18] 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.

[19] 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.

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

[21] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Lecture Notes in Computer Science*, volume 952, pages 77–101. Springer-Verlag, 1995.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] M. Fahad and A. Nadeem. A survey of uml based regression testing. In Zhongzhi Shi, E. Mercier-Laurent, and D. Leake, editors, *Intelligent Information Processing IV*, volume 288 of *IFIP Advances in Information and Communication Technology*, pages 200–210. Springer Boston, 2008.

[27] 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*, A-MOST '07, pages 44–52. ACM, 2007.

[28] Q. Farooq, M. Iqbal, Z. Malik, and M. Riebisch. A model-based regression testing approach for evolving software systems with flexible tool support. In *17th IEEE International Conference on Engineering of Computer-Based Systems (ECBS)*, pages 41–49. IEEE Computer Society, March 2010.

[29] 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.

[30] P. Frankl, G. Rothermel, K. Sayre, and F. Vokolos. An empirical comparison of two safe regression test selection techniques. In *ISESE '03 Proceedings of the 2003 International Symposium on Empirical Software Engineering*, pages 195–204. IEEE Computer Society, 2003.

[31] 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.

[32] V. Garousi, L. Briand, and Y. Labiche. *Model Driven Architecture - Foundations and Applications*, volume 3748 of *Lecture Notes in Computer Science*, chapter Control Flow Analysis of UML 2.0 Sequence Diagrams, pages 160–174. Springer Berlin / Heidelberg, October 2005.

[33] 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.

[34] 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.

[35] M. Grindal, J. Offutt, and J. Mellin. On the testing maturity of software producing organizations. In *TAIC-PART '06: Proceedings of the Testing: Academic & Industrial Conference on Practice And Research Techniques*, pages 171–180, 2006.

[36] 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.

[37] 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.

[38] F. Haftman, D. Kossmann, and E. Lo. A framework for efficient regression tests on database applications. *The VLDB Journal*, 16(1):145–164, January 2007.

[39] R. Haraty, N. Mansour, and B. Daou. *Advanced Topics in Database Research*, volume 3, chapter Regression test selection for database applications, pages 141–165. Idea Group, 2004.

[40] 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.

[41] 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.

[42] M. Harrold and G. Rothermel. Performing data flow testing on classes. In *Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering*, pages 154–163, 1994.

[43] 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.

[44] 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.

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

[46] 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.

[47] 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.

[48] 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.

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

[50] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 327–353. Springer-Verlag, 2001.

[51] T. Koju, S. Takada, and N. Doi. Regression test selection based on intermediate code for virtual machines. In *Proceedings of the International Conference on Software Maintenance*, ICSM '03, page 420. IEEE Computer Society, September 2003.

[52] 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.

[53] 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.

[54] 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.

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

[56] 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.

[57] H. Leung and L. White. A cost model to compare regression test strategies. In *Proceedings of the Conference on Software Maintenance*, pages 201–208, 1991.

[58] 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.

[59] B. Li, D. Qiu, S. Ji, and D. Wang. Automatic test case selection and generation for regression testing of composite service based on extensible BPEL flow graph. In *26th IEEE International Conference on Software Maintenance, ICSM 2010*, pages 1–10. IEEE Computer Society, 2010.

[60] 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.

[61] Feng Lin, Michael Ruth, and Shengru Tu. Applying safe regression test selection techniques to Java web services. In *International Conference on Next Generation Web Services Practices, 2006. NWeSP 2006.*, pages 133–142, Los Alamitos, CA, USA, September 2006. IEEE Computer Society.

[62] 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.

[63] H. Liu, Z. Li, J. Zhu, and H. Tan. Business process regression testing. In *Proceedings of the 5th international conference on Service-Oriented Computing*, ICSOC '07, pages 157–168. Springer-Verlag, 2007.

[64] 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.

[65] N. Mansour and W. Statieh. Regression test selection for C# programs. *Advances in Software Engineering*, 2009:1:1–1:16, January 2009.

[66] 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. IEEE Computer Society, December 2005.

[67] 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.

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

[69] 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*, ASE '07, pages 567–570. ACM, November 2007.

[70] 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.

[71] A. Orso, M. Harrold, and D. Rosenblum. Component metadata for software engineering tasks. In *Revised Papers from the Second International Workshop on Engineering Distributed Objects*, EDO '00, pages 129–144. Springer-Verlag, November 2000.

[72] 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.

[73] 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.

[74] 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.

[75] R. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, New York, 2002.

[76] 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.

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

[78] 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.

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

[80] 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.

[81] G. Rothermel and M. Harrold. Empirical studies of a safe regression test selection technique. *IEEE Transactions on Software Engineering*, 24(6):401–419, June 1998.

[82] G. Rothermel, M. Harrold, and J. Dedhia. Regression test selection for C++ software. *Software Testing, Verification and Reliability*, 10(2):77–109, June 2000.

[83] 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.

[84] 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.

[85] 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.

[86] 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.

[87] 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*, APSEC '03, pages 78–. IEEE Computer Society, 2003.

[88] A. Sangiovanni-Vincentelli and M. Di Natale. Embedded system design for automotive applications. *Computer*, 40(10):42–51, October 2007.

[89] 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.

[90] M. Skoglund and P. Runeson. A case study of the class firewall regression test selection technique on a large scale distributed software system. In *International Symposium on Empirical Software Engineering*, pages 74–83, November 2005.

[91] 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.

[92] 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.

[93] 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.

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

[95] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, September 1995.

[96] F. Vokolos. *A regression test selection technique based on textual differencing*. PhD thesis, Polytechnic University, 1998. UMI Order No. GAX98-10583.

[97] 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.

[98] 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.

[99] 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.

[100] 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.

[101] D. Wang, B. Li, and J. Cai. Regression testing of composite service: An XBFG-based approach. In *Proceedings of the 2008 IEEE Congress on Services Part II*, pages 112–119. IEEE Computer Society, 2008.

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

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

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

[105] D. Willmor and S. Embury. A safe regression test selection technique for database-driven applications. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 421–430. IEEE Computer Society, 2005.

[106] 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.

[107] 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.

[108] 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.

[109] 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.

[110] Lei Xu, Baowen Xu, Zhenqiang Chen, Jixiang Jiang, and Huowang Chen. Regression testing for web applications based on slicing. In *Proceedings of the 27th Annual International Computer Software and Applications Conference, 2003. COMPSAC 2003.*, pages 652–656, Los Alamitos, CA, USA, November 2003. IEEE Computer Society.

[111] S. Yoo and M. Harman. Pareto efficient multi-objective test case selection. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, pages 140–150, 2007.

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

[113] Y. Yuan, Z. Li, and W. Sun. A graph-search based approach to BPEL4WS test generation. In *Proceedings of the International Conference on Software Engineering Advances (ICSEA' 06)*, pages 14–. IEEE Computer Society, October 2006.

[114] 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.

[115] 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.

[116] 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.

[117] 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.

[118] F. Zhu, S. Rayadurgam, and W. Tsai. Automating regression testing for real-time software in a distributed environment. In *ISORC '98: Proceedings of the The 1st IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, page 373, 1998.