A model-based regression test selection approach for embedded applications

Article *in* ACM SIGSOFT Software Engineering Notes · July 2009

DOI: 10.1145/1543405.1543413 · Source: DBLP

4 authors, including:

Rajib Mall
Indian Institute of Technology Kharagpur
172 PUBLICATIONS **2,352** CITATIONS

SEE PROFILE

Manoranjan Satpathy
Indian Institute of Technology Bhubaneswar
70 PUBLICATIONS **483** CITATIONS

SEE PROFILE

A Model-Based Regression Test Selection Approach for Embedded Applications

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

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

Abstract

Regression test selection techniques for embedded programs have scarcely been reported in the literature. In this paper, we propose a model-based regression test selection technique for embedded programs. Our proposed model, in addition to capturing the data and control dependence aspects, also represents several additional program features that are important for regression test case selection of embedded programs. These features include control flow, exception handling, message paths, task priorities, state information and object relations. We select a regression test suite based on slicing our proposed graph model. We also propose a genetic algorithm-based technique to select an optimal subset of test cases from the set of regression test cases selected after slicing our proposed model.

Keywords: Regression testing, Regression test selection, Embedded software, Regression test optimization, Real-time, Safety critical.

1 Introduction

Over the last decade, there has been a proliferation of embedded systems and a variety of embedded applications have infiltrated into almost every facet of our daily lives, for example entertainment, automobiles, medical devices etc. With every passing year, the embedded applications are becoming more and more sophisticated resulting in a rapid increase in their size and complexity. Procedural languages and techniques are usually suited for the development of simple embedded systems like device drivers. However, object-oriented technologies are being increasingly adopted for development of embedded systems having more sophisticated interfaces such as automobile infotainment etc. The popularity of object-oriented techniques is largely attributable to the advantages they offer to handle complexity during design, development and maintenance of large software products as compared to the traditional design approaches. [Mal08]). On the other hand, satisfactory testing of object-oriented programs has turned out to be a challenging research problem [Bin99, MS01]. The real-time and safety-critical nature of embedded programs adds another dimension of complexity in testing of embedded programs [SPE+07, SPT05, TFB90].

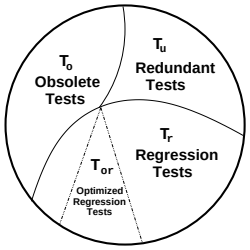

Figure 1: A partition of the initial test suite

Every software product typically undergoes frequent changes in its lifetime. These changes are necessitated on account of various reasons such as fixing defects, enhancing or modifying existing functionalities, or adapting to newer execution environments. Whenever a program is modified, it is necessary to carry out *regression* testing to ensure that no new errors (called regression errors) have been introduced. Regression testing is an expensive activity as it consumes large amounts of time and computing resources, and often accounts for more than half of the software maintenance costs [LW89].

The regression test suite of a product is a carefully chosen subset of the initial test suite. Figure 1 shows the classes into which test cases from the initial test suite can be partitioned after a modification has been made to a program [LW89, Mat08]. Obsolete test cases (T_o) are those test cases which are no longer valid for testing the modified program as they execute program elements that have been deleted in the modified version. Redundant test cases (T_u) execute only the unchanged parts of a program. Hence, although redundant test cases are valid (i.e., not obsolete), there is no need to re-execute them during regression testing. Regression test cases (T_r) are those test cases that execute the modified parts of a program, and hence, need to be run during regression testing. In Figure 1, only the test set $T_r \subseteq T$ should be used to revalidate the modified program. However, the set of test cases T_r selected for regression testing can be optimized by ignoring test cases which execute those modified parts that have already been covered by other test cases. In Figure 1, $T_{or} \subseteq T_r$ represents the set of optimized regression test cases.

http://doi.acm.org/10.1145/1543405.1543413 DOI: 10.1145/1543405.1543413

Regression test cases are usually selected from an initial test suite either based on expert judgment or based on manual analysis of the program. In either of these cases, even a very small change to the original program might require a large number of test cases to be rerun, leading to an unnecessarily high overhead in regression testing. What is probably more disconcerting is the fact that many test cases which could have potentially detected regression errors are overlooked during test selection [Bin99]. On the other hand, regression test selection techniques focus on *automatically* selecting *only* those test cases for retesting an application that are relevant to the modifications made to the original program.

Regression testing of real-time, safety-critical embedded applications is accepted as one of the most challenging tasks in the lifecycle of these applications [SPE+07, SPT05, TFB90]. Additional complexities are introduced during regression testing because of timing constraints and concurrent executions which are typical characteristics of embedded programs. Another dimension in regression testing of real-time and safety-critical embedded applications is that the execution of the test cases can be very expensive. The high cost of test setup for embedded applications, while being tested using hardware-in-loop (HIL) techniques or on fully operational systems (implemented on specialized hardware), contributes to the high expenses incurred during testing.

Automated selection of an optimal set of regression test cases from the initial test suite is a promising way of reducing the expenses incurred in regression testing. An important area of research, in this context, is the development of a technique for selection of an optimal suite of regression test cases for embedded applications such that the thoroughness of regression testing is not compromised. The problem of regression test selection (RTS) for procedural and object-oriented programs has been investigated by many researchers, and over the years many novel RTS techniques have been proposed [Bin97, GHS96, HJL+01, RH97, RHD00]. But these techniques cannot satisfactorily be applied to embedded programs, since embedded programs have many features that are very different from those of the traditional programs. Some of the important features that make it difficult to use RTS techniques designed for traditional programs for RTS of embedded programs are the following:

- Embedded programs are usually composed of a set of tasks. Each task is generally associated with specific priority and criticality information. For a safety-critical real-time system, any failure of the high priority and critical tasks are not tolerated, though occasional failures of low priority and less critical tasks may be acceptable. Hence, test cases should be selected such that the higher priority and critical tasks get more thoroughly tested compared to the low priority tasks.
- A real-time task is usually associated with a deadline by which it needs to complete its execution. Thus, all test cases testing the timing aspects of a modified task need to included.
- Embedded programs are concurrent and event-driven. These features can result in subtle bugs in the program that need to be specifically tested.
- Embedded programs use explicit exception handling mechanisms. This is especially true for safety-critical applications where error situations need to be properly handled. When an exception is thrown, the normal flow of control in a program usually gets altered. Hence, while testing a program, all possible control flows in the program need to be tested.

Several existing RTS techniques for traditional programs are based solely on static program analysis. However, a static analysis of programs has several shortcomings. It is computationally expensive, and various types of relations (like state transitions, message paths, task criticality etc.) among the program elements are not explicit in the code. Further, selection of regression test cases based on code analysis becomes more problematic if different parts of a program are written in different programming languages. The drawbacks of RTS techniques based on only code analysis is further accentuated for software products that are large, complex, and are frequently modified. Moreover, maintenance of softwares products evolving over many years adds to the complexities in the code. These drawbacks of code analysis-based RTS techniques have lead to an increased research focus on model-based regression testing techniques.

Model-based analysis is efficient and has several inherent advantages. A model is a compact representation of programs, and algorithms used for processing those models are more efficient than those for text analysis. Further, model-based regression testing can help take into consideration several aspects of embedded programs that are not easily extracted from the code. Such program aspects include object state, message path information, exception handling, timing characteristics etc. Some of these information can easily be extracted from design models, and the others from the requirements specification document (SRS) and incorporated in the model representing the program under test. An analysis of such a model, that has been augmented with the information extracted from the design and analysis models, can help to accurately identify all the relevant regression test cases. This model can also be used for prioritizing the regression test cases and selecting an optimal test suite.

In this paper, we propose a model-based RTS technique for embedded programs. We first propose a graph model that is constructed from program analysis and captures the different characteristics of embedded programs that are relevant for RTS. We subsequently enhance this model with information extracted from the SRS document, and the analysis and the design models. We then discuss our approach for RTS and regression test suite optimization based on the constructed model.

This paper is organized as follows: Section 2 contains an overview of our approach. In Section 3, we discuss our proposed intermediate representation of embedded programs and the construction of the same from analysis of the code,

http://doi.acm.org/10.1145/1543405.1543413 DOI: 10.1145/1543405.1543413

design models and the SRS document. We present our RTS and optimization strategies in Sections 4 and 5 respectively. In Section 6, we compare our approach with the related work available in the literature. Section 7 concludes the paper.

2 Overview of Our Approach

We have named our technique *M*odel-based Regression *Test* Selection for Embedded Software (MTest). Our technique is essentially based on first constructing suitable models for the original as well as the modified embedded program. The constructed models are then analyzed to identify the model elements that might be directly or indirectly affected due to the code modifications. Test cases executing the affected elements of the model are selected for revalidating the modified program. Considering that even for small code changes a large number of regression test cases can get selected, we propose a technique to select any desired number of test cases from the regression test suite to reduce the regression testing effort without unduly diluting the thoroughness of testing.

The important steps of our approach have schematically been shown in Figure 2. In Figure 2, rectangular blocks represent artifacts such as code, design, SRS etc. The ellipses in the figure represent processing activities such as instrumentation, slicing etc. We now briefly discuss the different steps involved in our approach.

(i) The *Intermediate Model Constructor* constructs the intermediate model for the original program. The constructed model also contains additional information from the analysis and the design models (e.g., SRS, UML diagrams).

(ii) The *Code Instrumenter* instruments the original program, and the instrumented code is executed on the initial test suite (T) by the *Program Execution* module. The instrumented code helps to determine the model elements which are covered by each test case. Each model element is marked with the test cases that execute it.

(iii) The *Model Differencer* analyzes the modified source code and identifies the model elements that are modified and tags those elements on the model.

(iv) The *Slicer* performs a forward slice [HRB90] on the modified marked model to identify the affected model elements that need to be retested. Each modified model element and the definition or use of all the variables at that point act as the slicing criterion [MM06]. Thus, all the directly or indirectly affected regions of code are identified through slicing. The set of test cases[1] which execute the affected elements are selected for regression testing of the embedded application.

(v) The *Optimizer* analyzes additional information about the program components gathered from the operational profile [Mus93], and prioritizes the test cases based on the criteria used in the operational profile module. A subset of test cases[2] from the prioritized test suite is chosen for regression testing of the software.

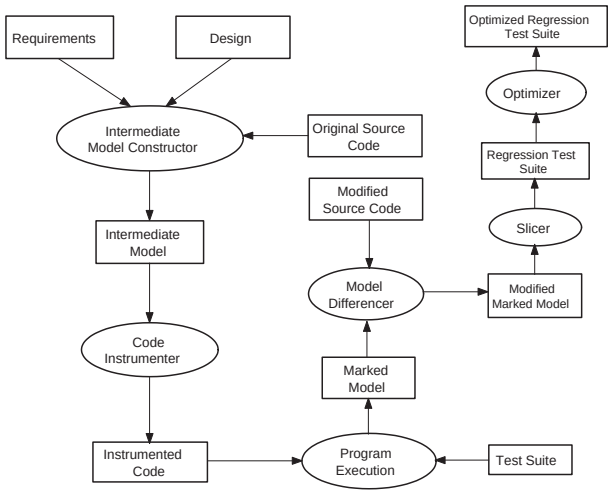

Figure 2: Schematic representation of our proposed RTS approach

3 EClDG: Our Proposed Model

In this Section, we first highlight the drawbacks of existing graphical models which render them inadequate to represent all the important characteristics of an embedded program. We extend an existing graph representation (i.e., ClDG [LM96, RH94a, RH94b]) to incorporate the necessary information that would be useful in RTS of embedded programs. We have named our proposed model EClDG (Extended-ClDG).

3.1 Existing Models

The System Dependence Graph (SDG) [HRB90] was proposed as an intermediate representation of procedural programs. SDGs have been used for a wide variety of applications, including program slicing [LH98, SHR99], impact analysis [KK07], reverse engineering [CHH06] etc. SDGs were extended to capture the special features of object-oriented programs, and the Class Dependence Graph (ClDG) [LM96, RH94a, RH94b] was proposed. ClDG is an extensively used model for intermediate representation of object-oriented programs. But embedded programs have several features that are not captured in a ClDG. Moreover, representing information available from the design model is important for embedded programs. This is because certain features such as object state, message paths, timing information, task priority and criticality of embedded programs are not easily identifiable from code analysis, but are explicitly available

[1] Corresponds to T_r in Figure 1.

[2] Corresponds to T_{or} in Figure 1.

http://doi.acm.org/10.1145/1543405.1543413 DOI: 10.1145/1543405.1543413

```
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 3: A sample program

from the design model. Hence, a ClDG representation is insufficient to model embedded programs. The following list summarizes the aspects of embedded programs that are not represented in a ClDG.

- In an embedded program, the criticality and timing information is usually associated with threads. ClDG does not represent the control flow of a program and hence cannot represent threads or their associated timing and criticality information.
- ClDG does not represent exception handling mechanism in embedded programs.
- SDG does not support any mechanism to represent information about object states, and state behavior.

3.2 EClDG Model

We have extended a ClDG model to incorporate additional features which capture the aspects specific to an embedded program. We augment a ClDG with the following additional information:

- Control flow.
- Exception handling.
- Information available from design models, such as:
  - Method sequences and message paths.
  - Timing and priority information.
  - Object state information.

In the following, we discuss how the above information is represented in our EClDG model.

3.2.1 Representation of Control Flow Information

Embedded programs usually consist of many co-operating tasks. Each task is associated with a deadline, as well as with certain priority and criticality information. To represent the priority and criticality information of tasks, we first need a way of representing the tasks in the EClDG. For this, we propose a mechanism to represent threads in the EClDG.

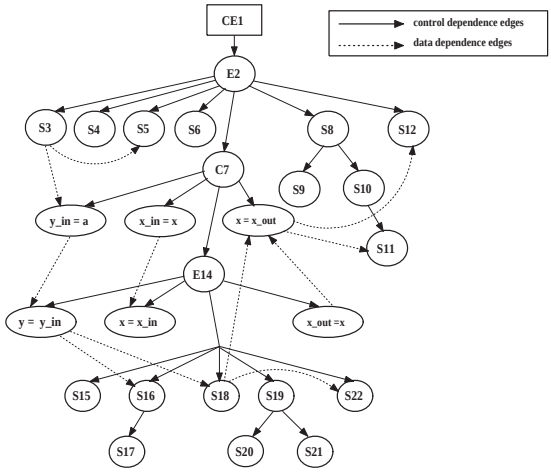

Figure 4: ClDG for class A for the program in Figure 3

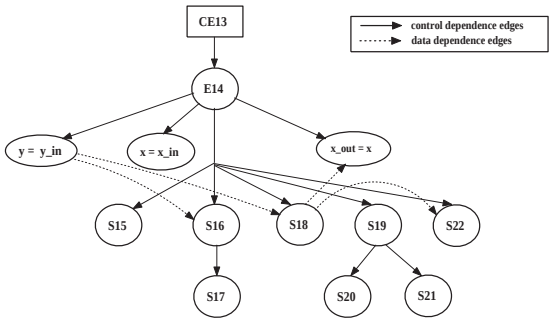

Figure 5: ClDG for class B for the program shown in Figure 3

A thread can be represented in terms of the sequence of statements constituting the thread. Therefore, it is necessary to incorporate control flow information in the EClDG to represent threads.

Computation of control flow information can be determined from the program by analyzing the sequence of program statements and the method calls. We have introduced control flow edges in the EClDG to represent the order in which the statements within a given method are executed.

Example 1 Figures 4 and 5 show the ClDG for the classes A and B of Figure 3. Figure 6 shows the ClDG for the entire program. The ClDG of Figure 6 augmented with the control flow information is shown in Figure 7. The control dependence and data dependence edges in the ClDG of Figure 7 is represented by continuos and dotted edges. The dashed edges in the figure represent the control flow information.

3.2.2 Representation of Exception Handling Information

Exception handling [Str04] is an important feature of object-oriented and embedded programs. Exceptions raised in a

http://doi.acm.org/10.1145/1543405.1543413 DOI: 10.1145/1543405.1543413

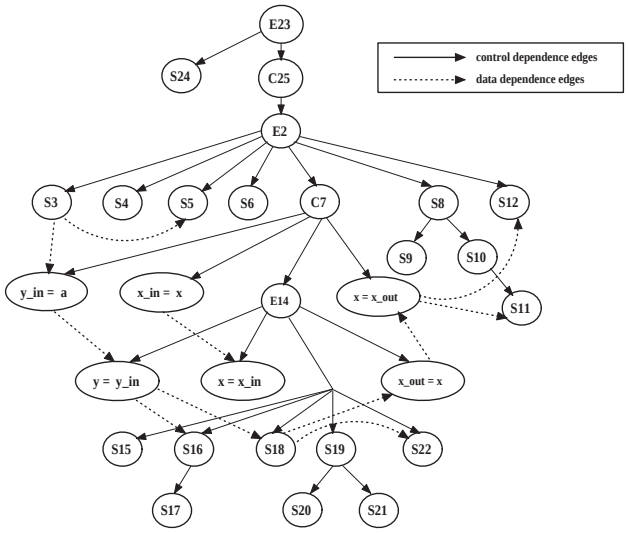

Figure 6: ClDG for the program given in Figure 3

program can alter the flow of control in that program, and may also change the dependence relationships for some variables [JZSJ06]. For example, execution of a throw statement in a C++ program would alter the normal control flow of the program. Therefore, it is necessary to represent all these possible alternate paths in the EClDG. In such cases, the data dependencies may also be affected as the exception handling mechanism may alter the definition-use sequences for some variables. For example in the sample program shown in Figure 3, in the absence of an exception in the method B::mB(), the value of the variable x in line S12 is dependent on the value of x in line S18, otherwise it is dependent on the global value of x [JZSJ06].

An exception can result in the following possible *intra-function* execution paths [JZSJ06]:

- An exception is raised in a try block [Str04]. The corresponding catch block handles the exception and the execution exits normally from the method.
- An exception is raised in a try block. The corresponding catch block handles the exception but rethrows or raises another exception. This exception is then handled by the enclosing try blocks (if nested) in a similar manner. If there are no more enclosing try blocks, then the exception is propagated to the calling function.
- An exception is raised in a try block but no corresponding catch block is found for handling the exception. This exception is then handled by the enclosing try blocks (if nested) or is passed to the calling function.

Exception handling can induce the following possible *inter-function* execution paths [JZSJ06]:

- The called function propagates the exception back to the calling function which then handles the exception.
- The called function propagates the exception back to the calling function, but the calling function is not able to handle the exception. The exception is then propagated upwards along the method invocation chain until a handler is found. If no handler is found for the exception, then the default handler is invoked.

Improvements in representing the exception handling mechanism have been proposed in previous works [AH03, JZSJ06, SH98, SH00, SOH04]. However, the work proposed in [SH98, SH00] suffer from the fact that the catch/throw nodes in the model have only one outgoing edge and hence are not able to adequately represent the data and control dependence information [JZSJ06]. Our approach to represent the exception handling information in the EClDG is based on the work reported in [AH03, JZSJ06]. We have introduced the following additional nodes in the EClDG to represent exception handling information: try, catch, throw, normal exit, exceptional exit, normal return and exceptional return nodes. The throw and catch statements are treated as conditional statements which alters the flow of control depending on the evaluation of the conditional expression. Therefore, the throw and the catch nodes are treated similar to predicate nodes. The try node depicts the start of the try block in the code. The normal return, normal exit, exceptional return and exceptional exit nodes represent normal and exceptional exits from a method.

Consider the sample program shown in Figure 3. The EClDG model corresponding to the program is shown in Figure 7. The exception handling information is represented in the EClDG by means of additional nodes. For example, in Figure 3, method B::mB() has a try-catch block. This information is represented in the EClDG model (see Figure 7) by two nodes, a normal exit and an exceptional exit. Nodes normal return and exceptional return have also been introduced in the EClDG model of Figure 7.

3.2.3 Representation of Information from UML Models

UML diagrams such as state charts, activity diagrams (sequence and collaboration diagrams) are used to model the dynamic behavior of embedded applications. Therefore, information pertaining to object states, state transitions, and message paths which are required for RTS can be extracted from an analysis of the UML models and represented in the EClDG.

- **Representation of method sequences and message paths:** Along with the control flow information, it is important to represent the different sequences in which methods may be invoked in an embedded program. For this, it is necessary to first identify the messages which trigger a particular sequence of method calls. Zhao and Lin [ZL06] and Jorgensen and Erickson [JE94] have proposed the concept of a Method-Message path (MM-Path) which represents the sequence of methods and the corresponding messages which triggered the

http://doi.acm.org/10.1145/1543405.1543413 DOI: 10.1145/1543405.1543413

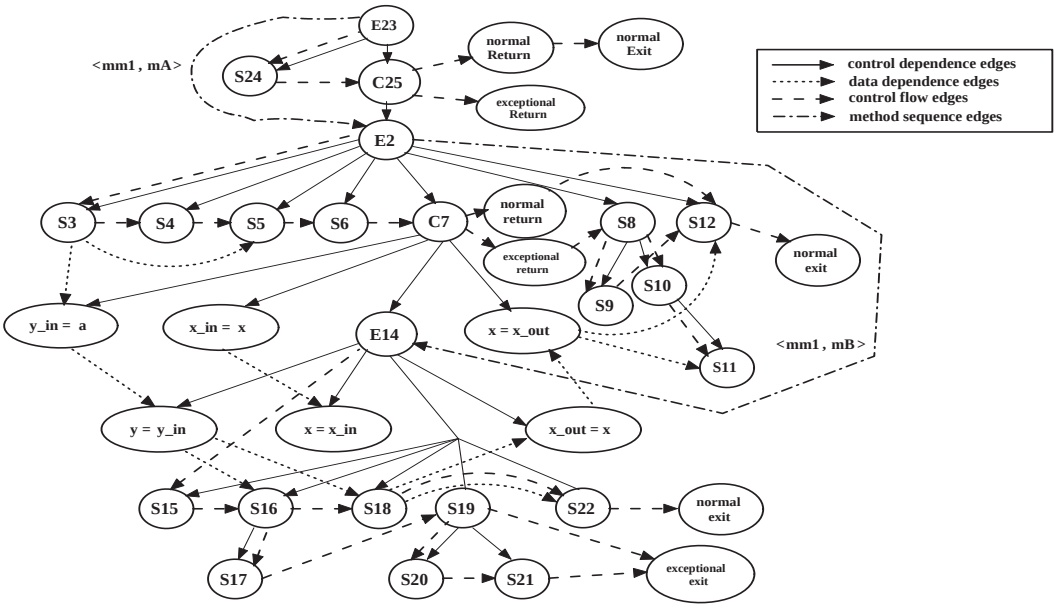

Figure 7: EClDG for the program given in Figure 3

methods. We assign an unique identifier to each method sequence. We augment the ClDG with the method sequence information by introducing method-sequence edges. For example, the EClDG model in Figure 7 represents the method sequence information with method-sequence edges. The method-sequence edges are labeled with the tuple <*method sequence identifier, message*>. The message information is also required for identifying message sequences because, in object-oriented programs, dynamic polymorphism enables the same message to invoke different methods during runtime.

- **Representation of timing and priority information:** In embedded applications, threads are used to represent concurrent activities. For real-time tasks, each thread may be associated with specific timing information and different threads may have different priorities. A thread is usually implemented either as a single method or a sequence of methods. We represent the priority information of each thread in the start node of the corresponding method sequence in the EClDG.

- **Representation of object state information:** An object state is defined by the values of the state variables stored in that object. A state transition for an object occurs when the corresponding state variables are modified. This usually occurs as a consequence to some event or on invocation of certain operations on that object. The initial state and the guard condition for a transition determine the final state to which the object would transit. The pair (initial state, guard condition) can be used to uniquely identify all possible state transitions for an object. We represent the different object states and possible transitions among those states for each object by means of a transition table. This information is represented in the EClDG by storing the state transition table of a class object in each class entry node of that particular class.

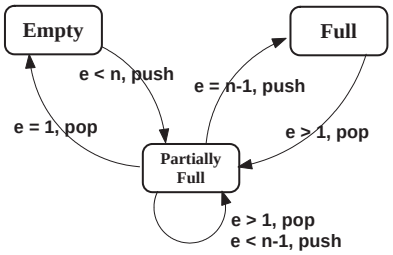

Figure 8: Statechart diagram for the Stack class

Example 2 We illustrate the construction of the transition table using the example of a Stack class. The Stack class is characterized by three states: empty, full and partially-full, and it can hold a maximum of 'n' elements. Let us assume that the guard conditions for the Stack class is defined based on the number of elements stored in the stack at a given instant. The transitions of a Stack object for a given operation are defined on the combination of the current state and the guard condition. The operations defined on the Stack are push and pop. The push operation inserts an element into the stack if it is not full. Pop operation removes an element from the stack if it is not empty.

Figure 8 shows the statechart diagram for such a Stack class. In Figure 8, the rectangular blocks with curved edges denote states, and the edges represent possible transitions

http://doi.acm.org/10.1145/1543405.1543413 DOI: 10.1145/1543405.1543413

Table 1: State transition table for the statechart of a Stack class shown in Figure 8.

Initial State	Condition	Operation	Final State
empty	e < n	push	partially full
empty		pop	ND
partially full	e > 1	pop	partially full
partially full	e < n-1	push	partially full
partially full	e = 1	pop	empty
partially full	e = n-1	push	empty
full	e > 1	pop	partially full
full		push	ND

between the states. The edges are labeled with the condition and the operations triggering the transition. The transition table for the statechart diagram in Figure 8 is shown in Table 1.

4 Regression Test Selection

Our RTS approach is based on those program elements of the EClDG model that are exercised during the execution of a test case. For this, the original program is instrumented, and the instrumented code is executed with each test case from the initial regression test suite. The instrumented code when executed marks the EClDG such that each model element is made to store an identifier for the test case which executed it. This approach of storing the test history information is space efficient especially for large programs and makes the later processing and searching steps efficient. After the program is modified, the modifications are identified by the *Model Differencer* and the EClDG model is augmented with the information of all the changed model elements. We call the EClDG model augmented with the test history and the change information as the Modified Marked Model (see Figure 2).

The parts of the program that are affected by the code modifications are identified by the *Slicer* by computing forward slices [HRB90] on the modified marked model. Each modified element of the modified marked model and the variables defined in the modified model elements are used as the slicing criterion [MM06]. Forward slicing on the modified marked model determines all the directly or indirectly affected elements of the model. After all the affected model elements are identified by the *Slicer*, the test cases that execute those modified model elements are selected for regression testing (see Figure 2). The test case identification step becomes trivial since the test cases executing the modified elements are already stored in the corresponding model elements.

Note that object state information in the EClDG has not been incorporated into the above RTS approach. How to use this information to improve the RTS approach is currently under investigation.

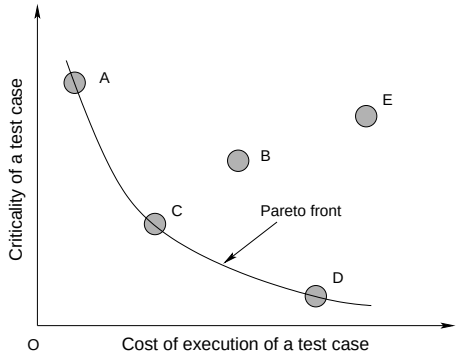

Figure 9: Pareto optimality in multi-objective optimization

5 Regression Test Suite Optimization

As compared to traditional approaches, our RTS technique takes into consideration many additional relationships between program elements that arise in an embedded application. However, due to the additional dependencies that are considered, the number of test cases selected for regression testing can become considerably larger than the traditional techniques. But executing a large number of regression test cases especially for minor changes to the code is impractical for many embedded applications such as automobiles where execution of each test case is usually very expensive. Therefore, we optimize the selected regression test suite to reduce the size.

Our optimization approach aims at selecting a subset of regression test cases according to the criticality of the program elements. Our optimization approach reduces the size of the regression test suite without affecting the achieved coverage of the test suite selected by our RTS technique. The objective of our optimization approach is to maximize the number of times the critical program elements are executed during regression testing. The criticality of the program elements can be determined by identifying the critical and high priority tasks in the embedded application. The information from the operation profile [Mus93] of the system helps in determining the features that are executed most frequently.

Our test suite optimization approach is based on the following criteria:

(i) Maximize the sum of priorities of the selected test cases.
(ii) Ensure maximum coverage of the different model elements in proportion to their priorities.
(iii) Minimize the cost of testing.

These various parameters used for optimization are competing in nature, i.e., increase in one may be at the cost of decreasing the value of some other parameter. For example, test cases which execute critical and high priority program elements may also be more expensive to execute as compared to test cases which execute lower priority and less critical elements. These competing factors make our test suite

http://doi.acm.org/10.1145/1543405.1543413 DOI: 10.1145/1543405.1543413

optimization problem a multi-objective optimization problem. Evolutionary techniques like Genetic Algorithms (GA) [Gol00] have successfully been used to solve various multi-objective optimization problems. These evolutionary techniques have also been used in studies related to prioritizing regression test cases [HCP08, LHH07]. We use a GA-based approach for our test suite optimization problem. For multi-objective optimization problems, the concept of Pareto-optimal solution is used to identify a set of good solutions (called the Pareto-optimal set) [FF93, Gol00, YH07, ZT98]. The concept of *dominance* is used to explain whether one solution is better than the other with respect to some or all the parameters considered in the optimization process.

For example, in Figure 5, solution B is dominated by solution C with respect to both the parameters, cost of test cases and the criticality value of a test case. But the solutions A, C, and D in Figure 5 are not dominated by any other solution, and hence constitute the Pareto-optimal set. Our technique chooses an optimal set of test cases from the computed Pareto-optimal set.

6 Comparison with Related Work

Many RTS techniques have been proposed in the literature for regression testing of traditional programs [Bin97, GHS96, HJL+01, RH94a, RH97, RHD00]. But we could not find any work focusing on RTS of embedded programs. In the absence of any directly comparable work, we compare our technique with the reported work on RTS of procedural and object-oriented programs. Existing model-based approaches for traditional programs [HJL+01, RH94a, RH97, RHD00] construct graphical models based solely on source code analysis of the programs. In contrast, our proposed EClDG model is constructed from source code analysis and is also augmented with information from the analysis and design models. Our model improves upon the existing ones by explicitly considering object state, exception handling, message path information, criticality etc. Since our approach considers control and data dependencies along with other types of possible code relations among program elements, our proposed test selection technique is safe[3]. Another advantage inherent to our approach is that the test history information is not stored in a separate file as in some other RTS techniques [RH94a, RH97]. In contrast, each model element is associated with the test cases that execute it. This helps to save storage space and makes the RTS process more efficient especially for large programs having hundreds of test cases. We also optimize the set of test cases selected for regression testing to reduce the regression test effort.

[3]A safe RTS technique selects all the relevant test cases from the initial test suite that can potentially reveal defects in the modified program [RH96].

7 Conclusion

RTS of embedded applications requires addressing additional issues such as control flow, message path information, criticality etc. as compared to RTS techniques for traditional programs. Therefore, the test suites selected using traditional RTS techniques for regression testing of embedded programs are unsafe. Many of the required information for RTS of embedded programs are easily obtained from the SRS document and the design models. Therefore, we have proposed a model for embedded programs which is enriched with the required information. However, due to the additional dependencies that are considered in our approach, larger number of test cases are likely to be identified as compared to the traditional techniques. Subsequently, to reduce the number of selected test cases, we use a GA-based optimization technique. Case studies carried out by us on several small applications show that our approach indeed selects all the test cases that are important for regression testing, and at the same time, does not unduly increase the size of the selected test suite.

References

[AH03] M. Allen and S. Horwitz. Slicing java programs that throw and catch exceptions. In *In PEPM 03: Proceedings of the 2003 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*, pages 44–54, 2003.

[Bin97] D. Binkley. Semantics guided regression test cost reduction. *IEEE Transactions on Software Engineering*, 23(8):498–516, August 1997.

[Bin99] R. Binder. *Testing Object-Oriented Systems:Models, Patterns, and Tools.* Addison-Wesley, 1999.

[CHH06] 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.

[FF93] C. Fonseca and P. Fleming. Genetic algorithms for multiobjective optimization: Formulation, discussion and generalization. In *Genetic Algorithms: Proceedings of the 5th Internation Conference*, pages 416–423, July 1993.

[GHS96] 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.

[Gol00] D. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley, 2000.

[HCP08] K. Hla, Y. Choi, and J. Park. Applying particle swarm optimization to prioritizing test cases for embedded real time software retesting. In *Proceedings of the 2008 IEEE 8th International Conference on Computer and Information Technology Workshops*, pages 527–532, 2008.

http://doi.acm.org/10.1145/1543405.1543413 DOI: 10.1145/1543405.1543413

[HJL+01] 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.

[HRB90] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. on Programming Languages and Systems*, 12(1):26–61, January 1990.

[JE94] P. Jorgensen and C. Erickson. Object-oriented integration testing. *Communications of ACM*, 37(9):30–38, 1994.

[JZSJ06] S. Jiang, S. Zhou, Y. Shi, and Y. Jiang. Improving the preciseness of dependence analysis using exception analysis. In *Proceedings of the 15th International Conference on Computing IEEE*, pages 277–282, 2006.

[KK07] 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.

[LH98] 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.

[LHH07] Z. Li, M. Harman, and R. Hierons. Search algorithms for regression test case prioritization. *IEEE Transactions on Software Engineering*, 33(4):225–237, April 2007.

[LM96] L. Larsen and M.Harrold. Slicing object-oriented software. In *Proceedings of the 18th International Conference On Software Engineering*, pages 495–505, March 1996.

[LW89] H. Leung and L. White. Insights into regression testing. In *Proceedings of the Conference on Software Maintenance*, pages 60–69, 1989.

[Mal08] R. Mall. *Fundamentals of Software Engineering*. Prentice Hall of India, 2nd edition, 2008.

[Mat08] A. Mathur. *Foundations of Software Testing*. Pearson Education, 2008.

[MM06] G. Mund and R. Mall. An efficient interprocedural dynamic slicing method. *Journal of Systems and Software*, 79(6):791–806, June 2006.

[MS01] J. McGregor and D. Sykes. *A Practical Guide to Testing Object-Oriented Software*. Addison-Wesley, March 2001.

[Mus93] J. Musa. Operational profiles in software-reliability engineering. *IEEE Software*, 10(2):14–32, March 1993.

[RH94a] G. Rothermel and M. Harrold. Selecting regression tests for object-oriented software. In *International Conference on Software Maintenance*, pages 14–25, March 1994.

[RH94b] G. Rothermel and M. Harrold. Selecting tests and identifying test coverage requirements for modified software. In *Proceedings of the ACM International Symposium on Software Testing and Analysis*, pages 169–184, August 1994.

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

[RH97] 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.

[RHD00] G. Rothermel, M. Harrold, and J. Dedhia. Regression test selection for C++ software. *Software Testing, Verification and Reliability*, 10:77–109, June 2000.

[SH98] S. Sinha and M. Harrold. Analysis of programs with exception-handling constructs. In *In Proceedings of the International Conference on Software Maintenance*, pages 348–357, 1998.

[SH00] S. Sinha and M. Harrold. Analysis and testing of programs with exception-handling constructs. *IEEE Transactions on Software Engineering*, 26(9):849–871, 2000.

[SHR99] 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.

[SOH04] S. Sinha, A. Orso, and M. Harrold. Automated support for development, maintenance, and testing in the presence of implicit control flow. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 336–345, 2004.

[SPE+07] D. Sundmark, A. Pettersson, S. Eldh, M. Ekman, and H. Thane. Efficient system-level testing of embedded real-time software. In *Work in Progress Session of the 17th Eurmicro Conference on Real-Time System, Spain*, pages 53–56, December 2007.

[SPT05] 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.

[Str04] B. Stroustrup. *The C++ programming language*. Addison-Wesley, 3rd edition, 2004.

[TFB90] J. Tsai, K. Fang, and Y. Bi. On real-time software testing and debugging. In *Proceedings of the Fourteenth Annual International Computer Software and Applications Conference,*, pages 512–518, 1990.

[YH07] 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.

[ZL06] R. Zhao and L. Lin. An uml statechart diagram-based mmpath generation approach for object-oriented integration testing. In *Proceedings of World Academy of Science, Engineering and Technology*, volume 16, November 2006.

[ZT98] E. Zitzler and L. Thiele. Multiobjective optimization using evolutionary algorithms - a comparative case study. In *Proceedings of the 5th International Conference on Parallel Problem Solving from Nature*, pages 292–304, 1998.

http://doi.acm.org/10.1145/1543405.1543413 DOI: 10.1145/1543405.1543413