A Regression Test Selection Technique for Embedded Software

SWARNENDU BISWAS and RAJIB MALL, Indian Institute of Technology Kharagpur
MANORANJAN SATPATHY, GM India Science Lab

The current approaches for regression test selection of embedded programs are usually based on data- and control-dependency analyses, often augmented with human reasoning. Existing techniques do not take into account additional execution dependencies which may exist among code elements in such programs due to features such as tasks, task deadlines, task precedences, and intertask communications. In this context, we propose a model-based regression test selection technique for such programs. Our technique first constructs a graph model of the program; the proposed graph model has been designed to capture several characteristics of embedded programs, such as task precedence order, priority, intertask communication, timers, exceptions and interrupt handlers, which we consider important for regression-test selection. Our regression test selection technique selects test cases based on an analysis of the constructed graph model. We have implemented our technique to realize a prototype tool. The experimental results obtained using this tool show that, on average, our approach selects about 28.33% more regression test cases than those selected by a traditional approach. We observed that, on average, 36.36% of the fault-revealing test cases were overlooked by the existing regression test selection technique.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*

General Terms: Performance, Verification, Reliability

Additional Key Words and Phrases: Embedded programs, intertask communication, regression test selection, software maintenance, slicing, task execution dependencies

ACM Reference Format:
Biswas, S., Mall, R., and Satpathy, M. 2013. A regression test selection technique for embedded software. ACM Trans. Embedd. Comput. Syst. 13, 3, Article 47 (December 2013), 39 pages.
DOI: http://dx.doi.org/10.1145/2539036.2539043

1. INTRODUCTION

Of late, there has been a rapid surge in the usage and reach of embedded applications. A large variety of embedded applications now touch our daily lives. These include home appliances, communication systems, entertainment systems, and automobiles, just to name a few. In addition to the rapid increase in the popularity of embedded systems, an unmistakable trend is their increasing size and sophistication [Seo et al. 2008]. The enhanced capabilities of embedded systems coupled with the user demands for flexibility have contributed to prolific usage of these systems even in safety-critical areas, such as nuclear power stations, healthcare, automotive, avionics, etc. [Salewski and Taylor 2007]. Embedded systems used in safety-critical applications are required to have far greater reliability than conventional applications. Seo et al. have reported

Authors' addresses: S. Biswas (corresponding author) and R. Mall, Department of Computer Science and Engineering, IIT Kharagpur, India 721302; M. Satpathy, GM India Science Lab, Bangalore, India; corresponding author's email: swarnendu@cse.iitkgp.ernet.in.

DOI: http://dx.doi.org/10.1145/2539036.2539043

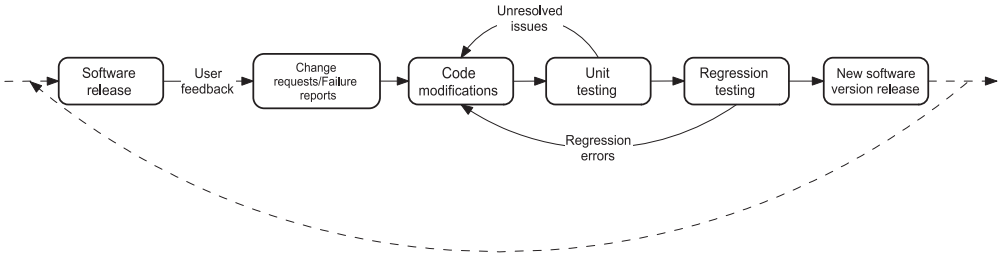


Fig. 1. Maintenance process model.

[2008] that though less than 20% of the functionalities of an embedded system are implemented in software, yet more than 80% of the reported failures could be attributed to software bugs. In this context, effective *regression* testing of evolving embedded software assumes increased significance.

Maintenance of an embedded program is frequently necessitated to fix bugs, to enhance or adapt existing functionalities, or to port it to different environments. Figure 1, adapted from Do et al. [2010], shows a popularly-followed maintenance process model. As shown in Figure 1, after a software is released, the failure reports and change requests for the software are periodically compiled, and the software is modified to make the necessary changes. After the necessary changes have been made, unit testing is carried out to ensure that the changes are proper. The objective of regression testing is to ensure that no new errors have been introduced in the unmodified parts of the code due to the changes made [Leung and White 1989]. Here, we would like to note that some existing papers in the literature also include testing the directly modified parts of the code as part of regression testing. In our work, we consider testing the directly changed parts of the code as repeated execution of unit testing. Unit tests are re-executed to validate the modified parts of the code, while regression testing is carried out to revalidate the unchanged parts of the code that might have been affected by the code change. After testing is complete, a new version of the software is released, which then undergoes a similar maintenance cycle.

Regression testing is carried out during different phases of software development: at unit, integration, and system testing phases, as well as during the maintenance phase [Leung and White 1989]. Regression testing of an evolving application is a crucial activity and consumes significant amounts of time and effort. The extents to which time and effort are being spent on regression testing are exemplified by a study [Do et al. 2010] that reports that it took 1,000 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 on regression testing activities other than test execution, such as setting up test runs, monitoring test execution, analyzing results, and maintaining testing resources, etc. [Do et al. 2010]. In fact, regression testing has been estimated to account for almost half of the total software maintenance costs [Kapfhammer 2004; Leung and White 1989]. To reduce regression testing costs, it is necessary to eliminate all those test cases that solely run the unaffected parts of the code, because they are unlikely to detect any bug. At the same time, it is also important to ensure that no test case that has the potential to detect a regression bug is overlooked. Accurate regression test selection is, therefore, considered to be an issue of considerable practical importance and has the potential to substantially reduce software maintenance costs [Guan et al. 2006].

1.1. Regression Test Selection

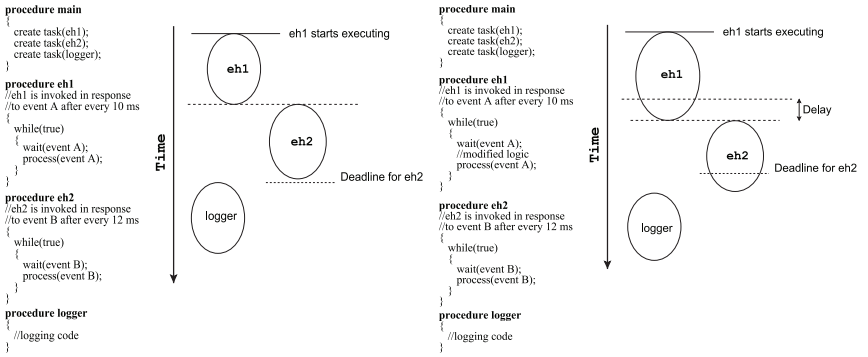
Regression test selection (RTS) techniques select a subset of valid test cases from an initial test suite (T) to test the affected but unmodified parts of a program [Leung and White 1989; Rothermel and Harrold 1997]. A large number of RTS techniques for procedural, object-oriented, component-based, aspect-oriented, and Web-based applications have been reported in the literature [Bates and Horwitz 1993; Binkley 1997; Rothermel and Harrold 1997; Harrold et al. 2001; Zheng et al. 2006; Orso et al. 2004]. However, research results on RTS for embedded programs are scarce. Possibly this is one of the reasons why regression test cases for embedded programs in the industry are selected based either on expert judgment or on some form of manual program analysis [Guan et al. 2006; Cartaxo et al. 2011]. However, the effectiveness of such approaches tends to rapidly decrease as the complexity of software increases [Cartaxo et al. 2011]. Furthermore, manual test selection tends to be conservative and often leads to a large number of test cases to be selected and rerun, even for minor program changes, leading to unnecessarily high regression testing costs. What probably is more disconcerting is the fact that many test cases which have the potential to detect regression errors could get overlooked during manual selection.

1.2. Challenges in RTS of Embedded Programs

As compared to traditional applications, regression testing of embedded programs poses several additional challenges [Sundmark et al. 2007; Sangiovanni-Vincentelli and Natale 2007; Netkow and Brylow 2010]. In the following, we briefly highlight the main complicacies that surface while selecting regression test cases for embedded applications. Embedded applications usually consist of concurrent and cooperating tasks having real-time constraints. Apart from verifying the functional correctness of an embedded program, satisfaction of timing properties of the tasks also needs to be tested. A cursory analysis of this situation reveals that selection of regression test suites for embedded programs based on analysis of data or control dependencies alone would not be satisfactory. Unless timing issues are carefully analyzed and taken into consideration, several potentially *fault-revealing* test cases may be omitted during RTS for embedded programs.

It could be argued that existing RTS techniques [Rothermel and Harrold 1997; Vokolos and Frankl 1997; Binkley 1997] reported in the context of procedural programs are unsuited for embedded programs. Traditional RTS techniques do not take into account the implications of several important features of embedded programs, such as concurrent and time-constrained tasks. In an embedded application, it is possible that the execution of a task τ_i may get delayed due to changes made to the code of some other task τ_j that does not have any data or control dependencies with the task τ_i. For example, when two tasks are communicating using a shared variable, the access to the shared variable is usually guarded by a semaphore. If a task blocks the semaphore for a longer duration due to a change made to the task code, then the execution of other tasks using that semaphore may get delayed. An unmodified task could also get delayed due to a modification made to the code of some other task due to issues such as message passing, precedence ordering, and priorities. Whenever the code of one task is changed, it becomes necessary to test those tasks whose execution time could potentially get affected due to implicit *task execution* dependencies. Therefore, in addition to traditional data and control dependencies, an RTS technique for embedded programs needs to take into account the execution dependencies among the various tasks.

In this context, it is important to note the difference between task timing analysis and execution dependency analysis. While timing analysis deals with prediction of worst-case execution time (WCET) for tasks, the aim of task execution dependency analysis



(a) Original program P. (b) Modified program P'.

Fig. 2. An example of a regression error introduced due to task execution dependency.

is to identify all those tasks in an application that can affect the timing behavior of a given task.

We present an example of a regression error that may go unnoticed if regression tested with test cases selected using a traditional RTS technique [Biswas et al. 2011]. Figure 2(a) shows the pseudocode for an embedded program P which is composed of three tasks, $eh1$, $eh2$, and *logger*. We make the following assumptions: (a) $eh1$ and $eh2$ are invoked in response to events A and B after every 10 ms and 12 ms, respectively, (b) $eh1$ is of higher priority than $eh2$, (c) there is a deadline by which event B needs to be handled by $eh2$ after it is invoked, and (d) there are no data or control dependencies among the program statements in $eh1$ and $eh2$. Note that *logger* is an auxiliary task and is not dependent on the execution of $eh1$ and $eh2$. Suppose the event handling logic in $eh1$ is changed in P', as shown in Figure 2(b), and as a result, $eh1$ takes longer to complete after the modification. Since $eh1$ is of higher priority, $eh2$ cannot start until $eh1$ is complete, that is, task $eh2$ is execution dependent on task $eh1$. Therefore, $eh2$ would get delayed and miss its deadline. Consider a test case t which tests only the event handling logic in task $eh2$. Existing procedural RTS techniques usually select test cases based on only data and control dependencies and therefore are likely to omit test cases, such as t for regression testing of P'.

Programmers extensively use exception handling mechanisms to increase the reliability and robustness of embedded software. Examples of commonly raised exceptions in embedded systems are timer expiration and `NULL` pointer exception. When an exception arises, it causes transfer of control from the point where it is raised (within the *exception* block) to the corresponding exception handler routine. Exceptions raised in a program often change the data-dependence relationships among program elements [Jiang et al. 2006]. This is because the exception handling mechanism may alter the definition-use sequences for some variables. Therefore, satisfactory RTS of embedded programs requires explicit analysis of control flow due to exception handling [Sinha and Harrold 1998].

In this article, we propose an RTS technique for embedded programs in an attempt to overcome the identified inadequacies of traditional approaches. We have considered a subset of MISRA C[1] as the programming language. Its inherent flexibility and the ease with which it can be ported to a wide range of hardware platforms has made MISRA C a popular choice for developing real-time and safety-critical embedded applications. We first propose a graph representation that can capture features of an embedded

[1]`http://www.misra.org.uk/`.

program that are important for RTS. Our proposed model, in addition to capturing data and control dependencies, represents control flow and a few other features of an embedded program, such as tasks, task precedences, and intertask communications. Subsequently, we present an RTS technique that is based on an analysis of the constructed model. We also discuss a prototype implementation of our proposed RTS technique. The following are the three main contributions of our work.

(1) The models proposed in the literature [Rothermel and Harrold 1997; Orso et al. 2004; Harrold et al. 2001] for use in the RTS process ignore many important features of embedded programs, such as tasks, task precedence orders, timeouts, intertask communication using message queues and semaphores, exception handling, and interrupts. However, as we have already discussed, these aspects appear to be important features that need to be considered during the RTS process. Our proposed model has been designed to capture these important features of embedded programs and is, therefore, an original contribution.
(2) Existing RTS techniques for procedural programs are based on analysis of data and control dependencies [Bates and Horwitz 1993; Binkley 1997]. A few techniques are based solely on analysis of control flow information [Rothermel and Harrold 1997]. However, modifications to a task in an embedded application can affect the timing behavior (i.e., completion times) of other execution dependent tasks. For this, we first determine the execution dependencies among tasks that arise due to various issues, such as task precedence orders, task priorities, intertask communication using message queues and semaphores, exception handling, and interrupts. Subsequently, we select regression test cases by analyzing task execution dependencies in addition to the usual control and data-dependence analysis.
(3) We have implemented a prototype tool based on our proposed RTS methodology. We have conducted experimental studies using several industry-standard case studies with our prototype tool and have analyzed the results of our studies. The results obtained from our experimental studies highlight the importance of task execution dependency analysis in RTS of embedded programs.

The rest of this article is organized as follows: In Section 2, we discuss some basic concepts related to our work. We discuss the different execution dependencies that can arise among the tasks in an embedded program in Section 3. In Section 4, we present the graph model that we have designed for embedded C programs. In Sections 5 and 6, we present our proposed RTS technique and the experimental results obtained by using our technique, respectively. In Section 7, we compare our approach with related work. Finally, Section 8 concludes.

2. BASIC CONCEPTS

In this section, we discuss certain basic concepts that underlie our approach to RTS for embedded programs. We first present some definitions used in the context of regression test selection and then discuss a few models proposed for procedural programs. Subsequently, we discuss some features of embedded programs that are relevant to regression test selection and also discuss a procedural RTS technique proposed by Binkley [1997] which we have used to compare our experimental results.

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

2.1. Concepts Related to Regression Test Selection

In this section, we discuss a few important notations and concepts relevant to our work on regression test selection.

2.1.1. Fault-Revealing Test Cases. Rothermel and Harrold [1996] have defined a *fault-revealing* test case for a traditional program P as a test case $t \in T$ that can cause P to fail by producing incorrect outputs for P. However, real-time embedded programs have time constraints associated with the output. Therefore, we define a fault-revealing test case in the context of a real-time embedded program as follows. A test case $t \in T$ is said to be fault-revealing for programs P and P' if and only if it can cause P' to fail by producing an incorrect output or cause the output to be produced too late.

2.1.2. Modification-Revealing Test Cases. Rothermel and Harrold [1996] have defined a *modification-revealing* test case as a test case $t \in T$ that produces different outputs for P and P'. We extend the definition of a modification-revealing test case for embedded programs as follows. A test case $t \in T$ is said to be modification-revealing for P and P' if and only if it produces different outputs when executed with P and P', or if the outputs for P and P' are produced at different instants of time.

2.1.3. Relevant Regression Test Cases, Safety, and Precision. A test case $t \in T$ is *relevant* to a change if it executes those unmodified parts of P' which are affected due to data, control, or task execution dependencies. Therefore, all relevant test cases need to be executed during regression testing of P'.

Rothermel and Harrold [1996] have defined a set of metrics to evaluate the effectiveness of an RTS technique. However, their metrics were proposed in the context of procedural programs and do not consider specific characteristics of embedded programs, such as the changed notion of correctness of an embedded program that also involves the notion of time.

In the context of embedded programs, we argue that a more accurate metric of the efficacy of RTS is the number (or percentage) of test cases that are selected from those that failed when all the valid test cases in the initial test suite are run on the modified program. Thus, the percentage of failed test cases selected by an RTS technique can serve as a figure of merit.

A cause for concern with testing embedded programs with concurrent tasks is nondeterministic interleaving of the tasks. It is possible that based on the timing, there are different interleavings of the tasks leading to different outputs being produced across runs. So, the set of selected test cases that make an RTS safe for one interleaving may not include a test case that is fault-revealing for another interleaving. Therefore, safety of RTS techniques for concurrent embedded programs is limited to safely selecting test cases based on the test execution history gathered over earlier testing sessions.

Precision measures the extent to which an RTS technique omits selecting non-relevant test cases.

2.1.4. Regression Testing Cycle. A regression testing cycle consists of the different activities that are carried out during regression testing, for example, test setup, test case selection, test case execution, etc. Therefore, there can be numerous regression testing cycles during software maintenance phase.

2.2. Procedural Program Models

Graph models of programs have extensively been used in many applications, such as program slicing [Liang and Harrold 1998; Sinha et al. 1999], reverse engineering [Cleve et al. 2006], etc. Some of the popular procedural graph models reported in the literature include control flow graphs (CFG) [Aho et al. 2008], program dependence graphs (PDG) [Ferrante et al. 1987], and system dependence graphs (SDG) [Horwitz et al. 1990]. In the following, we briefly review an SDG graph model since it is related to our work.

2.2.1. System Dependence Graph. A major limitation of a PDG representation is that it can only model a single procedure and is not able to model interprocedural calls.

Horwitz et al. [1990] enhanced the PDG representation to handle procedure calls and introduced the system dependence graph (SDG) representation to model a main program together with all its non-nested procedures.

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

$$\mathbf{VT_{SDG}} = \{V_{assign}, V_{pred}, V_{call}, V_{A_{in}}, V_{A_{out}}, V_{F_{in}}, V_{F_{out}}\},$$

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

—Node types V_{assign} and V_{pred} represent assignment statements and control predicates, respectively.
—*Call-site* nodes (V_{call}) represent the procedure call statements in a program.
—*Actual-in* ($V_{A_{in}}$) and *actual-out* ($V_{A_{out}}$) nodes represent the input and output parameters at a call site. They are control dependent on the corresponding *call-site* node.
—*Formal-in* ($V_{F_{in}}$) and *formal-out* ($V_{F_{out}}$) nodes represent the input and output parameters at the called procedure. These nodes are control dependent on the procedure's entry node.

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

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

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

—*Control* (E_{cd})- and *data* (E_{dd})-dependence edges represent the control and data dependence relationships among the nodes of an SDG, respectively.
—*Call* edges (E_{ce}) link the *call-site* nodes with the corresponding procedure entry nodes.
—*Parameter-in* edges ($E_{P_{in}}$) connect the *actual-in* nodes with the respective *formal-in* nodes.
—*Parameter-out* edges ($E_{P_{out}}$) connect the *formal-out* nodes with the respective *actual-out* nodes.
—*Summary* edges (E_{Sum}) are used to represent the transitive dependencies that arise due to function calls. A summary edge from an *actual-in* node a to an *actual-out* node b is constructed if the value associated with b can get affected by the value associated with the node a due to control or data dependence, that is, a summary edge from a to b is constructed if there exists either a control- or data-dependence edge from the corresponding *formal-in* node a' to the *formal-out* node b'.

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

Example 2. Figure 3 shows a much simplified version of a C program of an automotive application developed on a VxWorks [Wind River Systems 2010] platform. The corresponding SDG model for the program has been shown in Figure 4. In this figure, control-dependence edges are represented by dash-dot-dash edges, while the dotted edges represent data-dependence edges. The other types of SDG edges, such as *parameter-in*, *parameter-out*, *call* edge, etc., have been represented by uniformly-spaced dashed edges. Note that we have not shown all *actual-in* and *actual-out* nodes in the figure to avoid clutter.

```
D0  MSG_Q_ID g_msgq=NULL;

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

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

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

S8    return 0;
    }

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

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

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

Fig. 3. A sample VxWorks program incorporating intertask communication.

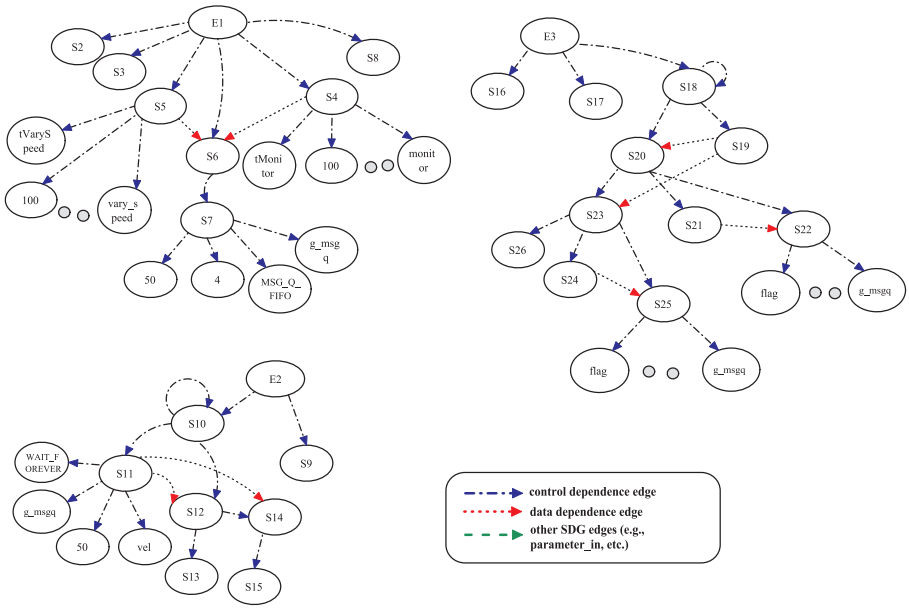


Fig. 4. SDG model for the program of Figure 3.

2.3. Tasks and Their Precedence Relations

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

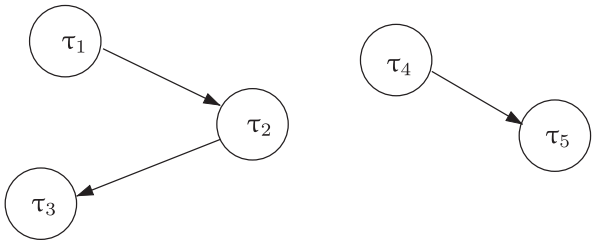


Fig. 5. A representation of the precedence relations among tasks.

2.3.1. Task Model. OSEK/VDX [2001] and POSIX RT[2] [Mall 2007] are two real-time operating system standards that are popularly being used in the development of embedded applications. In this article, we consider a task model that adopts features from both these standards. We assume that the tasks are statically created and have statically assigned priorities. The tasks are either periodic or aperiodic. The aperiodic tasks arise at random instants in response to user events. The tasks are scheduled using a priority-driven preemptive task scheduler. The tasks are assumed to communicate using either shared memory or message passing. This, of course, is usually the case for many embedded applications. During intertask communication, synchronization among tasks is typically achieved by using either of the following two techniques: shared memory or message passing. To achieve deterministic task execution, it is a general practice in embedded programming to guard access to shared memory through the use of synchronization primitives, such as semaphores and locks. Moreover, to achieve predictable results, embedded application developers also usually restrict themselves to using only the synchronous message passing mechanism [Vahid and Givargis 2002].

Development of large and complex embedded systems usually requires relaxation of many of the constraints on the task model that we have assumed. For example, large embedded systems may make use of sporadic tasks and asynchronous message passing. However, our task model is based on the assumptions frequently made in the development of small embedded applications. For example, an adaptive cruise controller (ACC) module in an automotive application is usually implemented with about a dozen periodic real-time tasks that have statically assigned priorities. The tasks are scheduled using a rate monotonic scheduler [Mall 2007]. Some of the important concurrently executing tasks in a typical ACC implementation include host vehicle speed controller (HVSM task) and radar information processor (RIP task), etc.

2.3.2. Task Precedence. Two tasks in an embedded program are said to be *precedence ordered* when the execution of one task is dependent on the actions of the other task. When a periodic task τ_i precedes another task τ_j, then it is implicitly assumed that each instance of task τ_i precedes the corresponding instance of τ_j. Precedence relationships define a partial ordering among tasks. An example of precedence ordering among tasks has been shown in Figure 5. The circles in Figure 5 represent tasks while the edges among them represent precedence relationships. A directed edge from a task τ_i to τ_j indicates that task τ_i should complete before task τ_j executes. From Figure 5, it can be inferred that τ_1 and τ_4 precede τ_2 and τ_5, respectively. However, we cannot ascribe any precedence ordering between the tasks τ_1 and τ_4 or the tasks τ_1 and τ_5.

We denote the precedence ordering of a task τ_i with other tasks by using two functions: $Pred(\tau_i)$ is the set of all those tasks whose execution must be complete before execution of task τ_i can be started. $Succ(\tau_i)$ is the set of tasks whose execution

[2]http://standards.ieee.org/regauth/posix/.

can start only after the task τ_i has completed. For the example shown in Figure 5, $Succ(\tau_1) = \{\tau_2, \tau_3\}$, and $Pred(\tau_5) = \{\tau_4\}$.

2.4. Exception Handling in Embedded Systems

Exception handlers in an embedded program execute at certain priority levels. Triggering of an exception and subsequent execution of the exception handler could delay completion of the lower priority tasks in the embedded program. Such delays to task completion times are often unacceptable for embedded applications having time-constrained tasks. In this context, a few studies have been reported in the literature to minimize the overhead of exception handling [Romanovsky et al. 1998].

The C programming language does not directly provide an exception handling mechanism. Many embedded programmers therefore implement exception handling either using jumps and `switch-case` constructs or through use of specific libraries [Schotland and Petersen 2011]. In this work, we assume that exception handling in embedded programs is implemented using the C++ `try-catch` model.

2.5. Binkley's SDG-Based RTS Technique

In this section, we briefly discuss Binkley's interprocedural RTS technique [1997] which was proposed for procedural programs. Binkley's technique is based on slicing SDG models. Two components are said to have equivalent execution patterns if and only if they are executed an equal number of times on any given input. The concept of *common* execution patterns has been introduced as an interprocedural extension to the *equivalent* execution patterns proposed in Bates and Horwitz [1993]. Code elements are said to have a common execution pattern if they have the same equivalent execution pattern during some call to a procedure. Common execution patterns capture the semantic differences among code elements. 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.

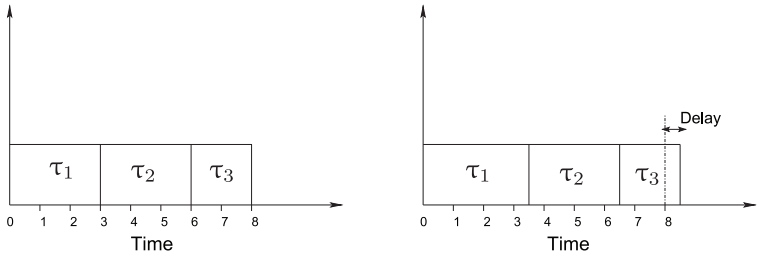
Our RTS technique uses an extended SDG model for representing embedded programs. We therefore compare the effectiveness of our RTS approach with the technique proposed by Binkley [1997].

3. TASK EXECUTION DEPENDENCIES IN EMBEDDED PROGRAMS

In Section 1, we have pointed out that embedded program features, such as tasks, task precedences, task deadlines, etc., give rise to execution dependencies among tasks [Marwedel 2007]. The effect of these dependencies can manifest as a task completion delay or even temporal failures. In this section, we discuss the causes and effects of execution dependencies that arise among tasks due to various factors, such as task precedence, task priority, message passing, use of shared memory, etc. It is important to note that task execution dependencies are not captured by the traditional notions of data and control dependencies.

Task Execution Dependency Due to Precedence Order. Given a set of time-constrained tasks, the completion of a task is dependent on the task precedence order (if any) among the set of tasks. A task τ_i cannot execute unless the set of tasks in $Pred(\tau_i)$ have already completed their execution. In this case, any delay to the completion time of a task τ_i could affect the completion time of the tasks in the set $Succ(\tau_i)$.

We explain the effect of execution dependencies introduced due to precedence order with the help of an example. For our examples, we assume periodic tasks which can be represented by a four tuple, $\tau_i = \langle \phi_i, p_i, e_i, d_i \rangle$, where ϕ_i is the phase of τ_i, p_i is the period of τ_i, e_i is the WCET of τ_i, and d_i is the relative deadline (with respect to ϕ_i) of



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

Fig. 6. Delay to task completions caused due to precedence relationships.

τ_i. In our work, without any loss of generality, we assume that the phases of all tasks are zero. Therefore, we represent a task τ_i by the three tuple $\langle p_i, e_i, d_i \rangle$.

Example 3. Let us consider an embedded program P consisting of three tasks $\tau_1 = \langle 10, 3, 4 \rangle$, $\tau_2 = \langle 10, 3, 8 \rangle$ and $\tau_3 = \langle 10, 2, 8 \rangle$. Let us further assume that task τ_1 precedes task τ_2 and τ_2 precedes task τ_3. A possible schedule of the tasks in P is shown in Figure 6(a). Suppose task τ_1 is modified in P'. It is possible that τ_1 in P' takes longer to execute, say 3.5 units, due to the modification. This is represented in the schedule shown in Figure 6(b). As a result of the change, the completion of the tasks in $Succ(\tau_1) = \{\tau_2, \tau_3\}$ will also get delayed by 0.5 time units.

Execution dependencies that exist among tasks due to their precedence ordering are transitive in nature. The set of tasks that are execution dependent on a task τ_i due to precedence relations is given by $Succ(\tau_i)$. From this discussion, it can be inferred that it is important to regression test those tasks which are execution dependent on some directly modified tasks due to precedence relationships.

Task Execution Dependency Due to Priorities. Execution dependencies among tasks can arise on account of task priorities. This dependency arises because in a priority-driven preemptive scheduling environment, the lower priority tasks will not be able to execute unless all ready higher priority tasks complete their execution.

We illustrate the effect of execution dependencies arising due to task priorities with the help of the following example and Figure 6.

Example 4. Let us consider the three tasks τ_1, τ_2 and τ_3 shown in Figure 6. Let the priority of each task be as follows: priority(τ_1) > priority(τ_2) > priority(τ_3). Assume that the execution order of the tasks are τ_1, τ_2, τ_3. Suppose task τ_1 is changed and as a result takes longer to execute, say 3.5 units. This would delay the other two tasks (as shown in Figure 6(b)).

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

Task Execution Dependency Due to Message Passing. Synchronous message passing in an embedded program gives rise to execution dependencies among a pair of communicating tasks. A modification to either the sender or the receiver task of a pair of tasks communicating using synchronous message passing may delay the completion of the other task.

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

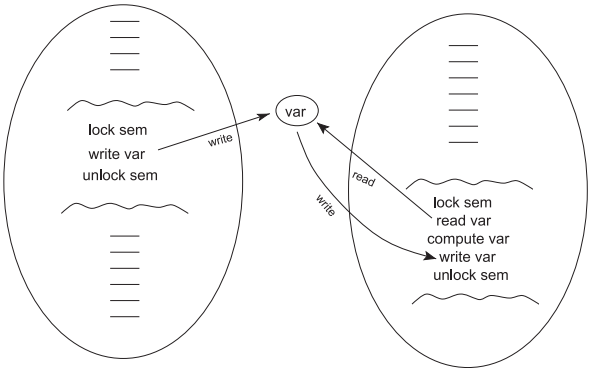


Fig. 7. Two tasks communicating using shared memory.

Task Execution Dependency Due to Access of Shared Resource. We assume that access to a shared variable is usually guarded using a synchronization primitive, such as a semaphore or a lock. The program statements for accessing such primitives provide implicit synchronization points between the concerned tasks. Therefore, any increase to the duration for which one task locks a synchronization variable may cause other tasks needing to lock the variable to get delayed.

Example 5. In Figure 7, we show an example of execution dependency that can exist among two tasks communicating using a shared variable. The tasks τ_1 and τ_2 in Figure 7 communicate using the shared variable *var* and use a semaphore variable *sem* to guard the access to *var*. Let us assume that task τ_2 computes and writes a new value for *var* which is later read by τ_1. Suppose the statement `compute var` in τ_2 is modified in P'. This change may cause τ_2 to block the semaphore for a longer time, thereby delaying τ_1.

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

It can be inferred that it is necessary to include in the regression test suite test cases testing all those tasks which share a resource with the modified tasks.

In the rest of this work, we have assumed that semaphores are used as synchronization primitives.

Dependency Due to Execution of Interrupt Handlers. Interrupts are profusely used in embedded applications to get notification of the occurrence of events of interest. For example, interrupts may be raised by sensors or peripheral devices. On receiving an interrupt, the corresponding interrupt service routine (ISR) gets invoked to perform operations that are necessary to handle the interrupt. The work performed by an ISR is usually split into two parts: the first-level interrupt handler (FLIH) and the deferred procedure call (DPC) [Silberschatz et al. 2010; Sales 2005]. The role of FLIH is to service the interrupt quickly by executing a few instructions only, and the rest of the handler procedure is executed as a DPC later. A DPC is scheduled as a normal task in many operating systems, such as the Symbian [Sales 2005].

Interrupt handling can cause unpredictable delays (called *jitter*) to the completion of some tasks. Such unpredictable delays to task completion times are usually unacceptable for hard real-time embedded applications. In this work, we assume that the execution time of an FLIH is negligible compared to a DPC, and the delay caused to a task due to execution of an FLIH can be ignored, and it is only the effect of the DPCs

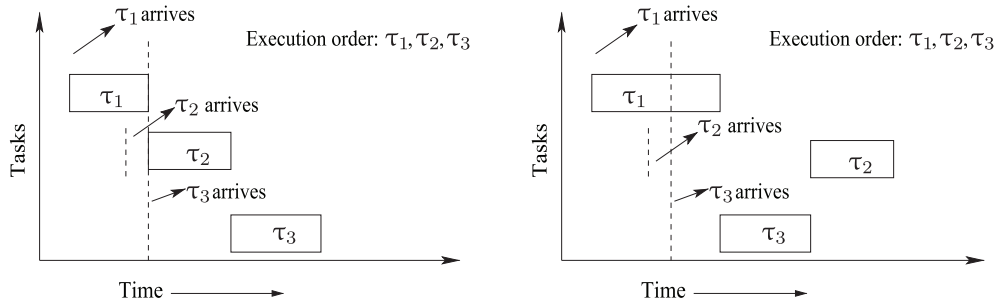


Fig. 8. Change in task execution order due to execution dependency.

which need to be considered. Since we assume that a DPC gets scheduled at the priority level of normal tasks, it can delay other tasks executing at the same or a lower priority.

3.1. Effect of Task Execution Dependencies on the Execution of an Embedded Program

Besides causing delays to the completion of certain tasks, the task execution sequence in an embedded program can also get altered due to execution dependencies among tasks, possibly leading to a different output being produced. In the following, we discuss an example to show how execution dependencies among tasks can alter the task execution sequence of an embedded program.

Example 6. Figure 8(a) shows three tasks and their times of arrival. The relative priorities of the three tasks are as follows: $priority(\tau_3) > priority(\tau_2) > priority(\tau_1)$. The execution order of the tasks in Figure 8(a) are τ_1, τ_2, τ_3. Let us assume that task τ_1 is modified and its WCET is increased, as shown in Figure 8(b). The modified task execution sequence now becomes τ_1, τ_3, τ_2. The modified task execution sequence in P' as a result of the execution dependencies introduced due to task priorities can cause a different output to be produced by P', or cause output to be produced at an altered time.

4. SDGC: A MODEL FOR RTS OF EMBEDDED PROGRAMS

CFG and SDG-based representations of programs are extensively being used in diverse applications, such as program slicing [Liang and Harrold 1998], reverse engineering [Cleve et al. 2006], regression test case selection [Rothermel and Harrold 1997; Bates and Horwitz 1993], etc. However, dependence graph-based representations ignore control flow information. As a result, it becomes difficult to capture important features of embedded programs, such as tasks, their execution dependencies, and exception handling among others using dependence graphs. A task is a basic programming entity that is defined by a group of program statements tied together through control flow relations. Furthermore, as advocated by many researchers, analysis of timing properties requires representation of control flow information [Ward and Mellor 1991; Hatley and Pirbhai 1987]. The semantics of task creation, message passing, semaphore access, timers, etc., are largely ignored by existing CFG, SDG-based models and are instead simplified into *ordinary* function calls. None of the program models proposed in the literature can represent all the important constructs, such as tasks, intertask communication, exception handling, that are used in programming embedded applications and that need to be considered in RTS. In order to overcome these shortcomings, we propose a graph model for representing embedded programs. Our proposed graph model is an

extension of the standard CFG and SDG representations [Bates and Horwitz 1993]. We have named our model SDGC for System Dependence Graph with Control flow.

Definition 4.1. An *SDGC model* for a program P is a directed graph $G = (V, E)$, where V represents the set of nodes and E represents the set of edges. The various types of nodes and edges defined for an SDGC model are represented by the sets $\mathbf{VT_{SDGC}}$ and $\mathbf{ET_{SDGC}}$ respectively, where,

$$\mathbf{VT_{SDGC}} = \{V_{assign}, V_{pred}, V_{call}, V_{A_{in}}, V_{A_{out}}, V_{F_{in}}, V_{F_{out}}, V_{task}, V_{mp}, V_{sem}, V_{timer}, V_{eh}\},$$
$$\mathbf{ET_{SDGC}} = \{E_{cd}, E_{dd}, E_{ce}, E_{P_{in}}, E_{P_{out}}, E_{Sum}, E_{cf}, E_{tdef}, E_{prec}, E_{mp}, E_{sem}, E_{timer}\}.$$

Since an SDGC model is an extension of a CFG and an SDG model, all node and edge types defined for a CFG and an SDG model are also present in an SDGC model. The sets $\mathbf{VT_{SDGC}}$ and $\mathbf{ET_{SDGC}}$ are supersets of the corresponding node and edge sets discussed for an SDG model in Section 2.2, that is, $\mathbf{VT_{SDG}} \subset \mathbf{VT_{SDGC}}$, and $\mathbf{ET_{SDG}} \subset \mathbf{ET_{SDGC}}$.

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

We now discuss the additional node and edge types that we have introduced in an SDGC model to represent those constructs of an embedded program that are important for RTS. We also provide examples of equivalent APIs as specified in POSIX.

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

—A *task create* node (denoted by V_{tc}) is used to model a task creation that has been programmed using a construct such as `fork()/spawn()`. The static priority value associated with the task is also stored in the task create node.
—A *task delay* node (denoted by V_{tdl}) is used to model a task delay that has been programmed using a construct such as `delay()`.
—A *task delete* node (denoted by V_{tdt}) is used to model a task deletion that has been programmed using a construct such as `delete()`.
—A *task exit* node (denoted by V_{tx}) is used to model a task exit that has been programmed using a construct such as `exit()`.

Message-Passing Nodes. We have introduced a node type called *messagepassing* (V_{mp}) to model the special semantics of message passing among tasks. The message-passing node type is divided into the following subtypes.

—A *message queue create* node (denoted by V_{qc}) is used to model a message creation statement that has been programmed using a construct such as `mq_open()`.
—A *message queue send* node (denoted by V_{qs}) is used to model a message send statement that has been programmed using a construct such as `mq_send()`.
—A *message queue receive* node (denoted by V_{qr}) is used to model a message receive statement that has been programmed using a construct such as `mq_receive()`.
—A *message queue delete* node (denoted by V_{qd}) is used to model a message queue deletion statement that has been programmed using a construct such as `mq_unlink()/mq_close()`.

Semaphore Nodes. We have introduced a *semaphore node* type (V_{sem}) to model program statements performing semaphore operations. The semaphore node type is divided into the following subtypes.

—A *semaphore create* node (denoted by V_{sc}) is used to model a statement that creates a semaphore using a construct such as `sem_open()/sem_init()`.
—A *semaphore req* node (denoted by V_{st}) is used to model a statement that requests a semaphore using a construct such as `sem_wait()`.

—A *semaphore rel* node (denoted by V_{sg}) is used to model a statement that releases a semaphore using a construct such as `sem_post()`.
—A *semaphore delete* node (denoted by V_{sd}) is used to model a statement that deletes a semaphore variable using a construct such as `sem_destroy()`.

Timer Nodes. We have introduced a *timer node* type (denoted by V_{tmr}) to model timer operations. The timer node type is divided into the following subtypes.

—A *timer create* node (denoted by V_{tmc}) is used to model a statement that creates a timer using a construct such as `timer_create()`.
—A *timer start* node (denoted by V_{tms}) is used to model a statement that starts/sets a timer using a construct such as `timer_settime()`.
—A *timer delete* node (denoted by V_{tmd}) is used to model a statement that deletes a timer using a construct such as `timer_delete()`.

Exception-Handling Nodes. Our approach to represent exception handling in an SDGC is based on previous work [Allen and Horwitz 2003; Jiang et al. 2006; Biswas et al. 2009]. We introduce an *exception-handling node* type (V_{eh}) to model exception handling. The exception-handling node type is divided into the following subtypes.

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

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

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

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

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

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

Semaphore Edge. The dependency arising due to the use of a semaphore between two communicating tasks is represented by a *semaphore edge* (denoted by E_{sem}). A semaphore edge connects a pair of nodes of type V_{st} and V_{sg} in the two tasks that access the same semaphore variable.

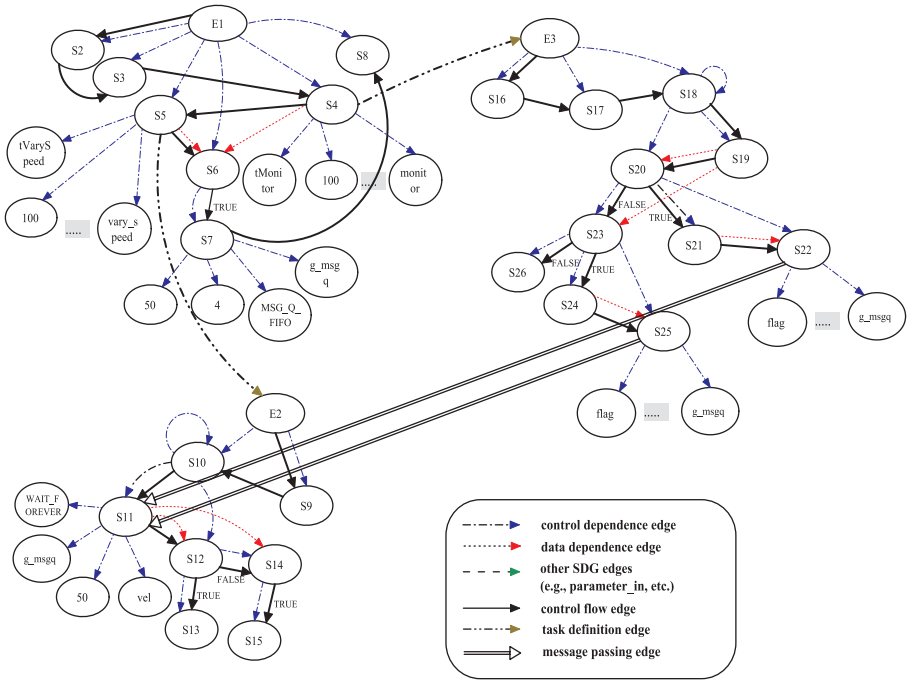


Fig. 9. SDGC model for the program of Figure 3.

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

Edges to Model Exceptions. We have introduced exception-handling edges to model changes to the normal control flow due to exceptions.

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

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

Example 7. The SDGC model of the program of Figure 3 is shown in Figure 9. It can be observed that the SDGC model in Figure 9 is an extension of the SDG model shown in Figure 4 and incorporates additional nodes and edges for modeling embedded program features, such as tasks, message queues, etc. The solid edges in Figure 9 represent control flow edges for each function in the program. We have omitted the `TRUE` labels on control flow edges wherever the flow of control is obvious to avoid cluttering the figure.

The program in Figure 3 has two statements spawning two tasks in lines $S4$ and $S5$. The task names are '`tMonitor`', and '`tVarySpeed`', and the task bodies are the functions

`monitor()` and `vary_speed()`, respectively. Since exact semantics of task creation are ignored by the SDG model, these two constructs in lines $S4$ and $S5$ are shown as simple function calls in Figure 4. In the SDGC model shown in Figure 9, the two task creation statements are represented by the task creation nodes $S4$ and $S5$. These two nodes are connected to the corresponding function entry nodes $E3$ and $E2$, respectively, by using task definition edges ($S4 \rightarrow E3$ and $S5 \rightarrow E2$).

The tasks `tMonitor` and `tVarySpeed` in the program shown in Figure 3 communicate using a message queue. The lines in Figure 3 (and the corresponding nodes) related to message queue management are $S7, S11, S22$, and $S25$. Node $S7$ in Figure 9 is of type V_{mc}, node $S11$ is of type V_{mr}, while nodes $S22$ and $S25$ are of type V_{ms}. Two message-passing edges have been used to connect nodes $S22$ and $S11$ and $S25$ and $S11$, respectively.

4.2. Construction of an SDGC Model

In the following, we discuss the construction of an SDGC model M through a static analysis of the source code of a program. The pseudocode for the SDGC model construction step has been shown in Algorithm 1. The input to the algorithm is the embedded program for which the corresponding SDGC model is to be constructed. The first step (line number 2 in Algorithm 1) constructs the CFG for each function in the program. This step involves parsing the input program and constructing the nodes and edges by

ALGORITHM 1: Pseudocode for Constructing an SDGC Model of an Embedded Program

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

carrying out the appropriate actions corresponding to each rule in the C grammar. All nodes, including the nodes modeling specific embedded program features, such as *task create*, *message queue send*, *timer start*, etc., are created in this step. For example, a *task create* node is created when a `taskSpawn()` statement is parsed. Once construction of the individual CFGs is complete, these are analyzed in lines 4 to 7 to compute and construct the data- and control-dependence edges.

In the next step, the partially constructed SDGC model is searched to find out the task create and timer create nodes. This can be accomplished by traversing the SDGC model along control flow edges and comparing the types of each node. Subsequently, these nodes are connected to the corresponding functions which are either task definitions or timer expiry handlers using *task definition* or *timer* edges, respectively. This helps in considering the special semantics of these program constructs by differentiating them from simple function calls. As an example, for the embedded C program shown in Figure 3, the name of the function `monitor` which is the definition for the task `tMonitor` is modeled as an *actual-in* argument which is connected to the task create node $S4$. Subsequently, we connect every message queue send node to the corresponding message queue receive node with a *message-passing* edge. The message queue receive node corresponding to a message queue send node can be identified from the message queue identifier over which the communication takes place. The message queue identifier is usually passed as an argument to the message queue send/receive statements and are therefore also modeled as an *actual-in* node. Similarly, *semaphore req* nodes are connected to the corresponding *semaphore rel* nodes using semaphore edges. A pair of semaphore req and *semaphore rel* nodes can easily be identified based on an examination of the name/ID of the semaphore variable.

The last step concerns identification of the precedence relationships among tasks. For this, the source code is analyzed for identifying the constructs `join()`/`wait()`. The usage of these two constructs determine whether a task τ_j precedes another task τ_i. We then construct a *task precedence* edge from the task create node of task τ_j to the *task create* node of task τ_i to model the precedence relationship between the two tasks.

Example 8. We illustrate the construction of the SDGC model for a sample program shown in Figure 10. First, the CFG for each procedure in the sample program is constructed. During creation of the CFGs, the *actual-in*, *actual-out*, *formal-in*, *formal-out*, *call-site*, *task create*, *timer create*, etc., nodes are also created. Subsequently, the data-dependence and control-dependence edges are computed and added to the partially constructed SDGC model. The resultant partial SDGC model after this step is shown in Figure 11. Finally, for this example, adding the task definition, *timer*, and the message-passing edges to the partially constructed SDGC model of Figure 11 completes the construction of the the SDGC model which has been shown in Figure 12.

In Figure 12, the node $S16$ is of type timer start. If the timer expires before it is reset in line $S22$, control flows to the handler routine `HandleTimeOut`. This is modeled in the SDGC model of Figure 12 with a timer edge connecting the nodes $S16$ and $E10$.

4.3. Complexity Analysis

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

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

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

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

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

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

Fig. 10. A sample VxWorks program incorporating usage of watchdog timers.

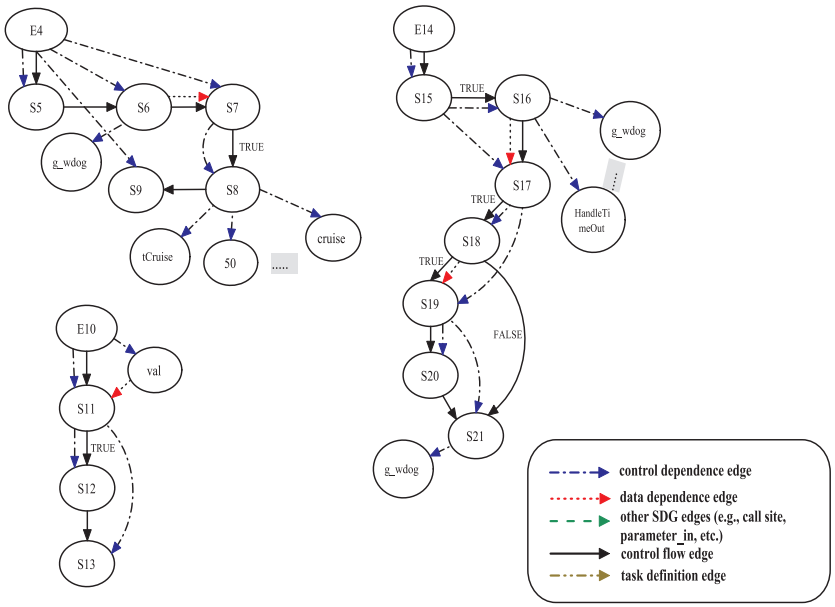


Fig. 11. Partially constructed SDGC model after computation of data and control dependencies.

to semantics of tasks, message passing, semaphores, and timer management. The time complexity of CFG construction is $O(n)$ [Aho et al. 2008], and the time complexity of control-dependence computation is $O(n^2)$ [Ferrante et al. 1987]. Computation of the data-dependence edges requires traversal of each CFG in the SDGC model and is bounded by $p * O(m^2)$. To create edges of type *semaphore*, *task precedence*, etc., requires traversing the SDGC model. This can be expressed by $O(m^2)$. Therefore, the time complexity of SDGC model construction is $O(m^2)$.

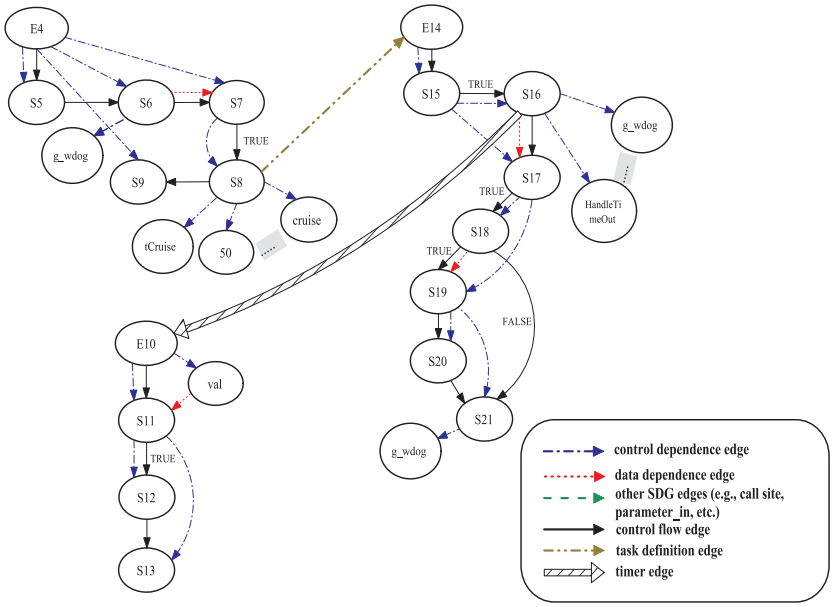


Fig. 12. Complete SDGC model for the sample program shown in Figure 10.

Space Complexity. Let us assume that there are n statements in P, and let n_f denote the number of functions in P (including those corresponding to task, timers, message queue, and semaphore creation), and arg_{max} be the maximum number of arguments of any function in P. For a given embedded program, the SDGC model typically contains more number of edges than the SDG model because the SDGC model captures several types of embedded program features which are not present in the SDG model. The space complexity for representing a graph is of the order of $O(n^2)$, where n is the number of nodes in the graph. Therefore, the space requirement for constructing an SDGC model is $O((n + 2 * n_f * arg_{max})^2)$. Assuming that the maximum number of arguments for a procedure call is a constant, say 10, the space complexity reduces to $O(n^2)$.

5. RTSEM: AN RTS TECHNIQUE FOR EMBEDDED PROGRAMS

Our proposed RTS technique, RTSEM (Regression Test Selection for EMbedded programs), is based on analyzing the SDGC model of an embedded C program. More specifically, our technique selects test cases based on an analysis of control, data and execution dependencies among tasks. In the following, we first list the assumptions that we make and then describe the different processing activities carried out in RTSEM.

5.1. Assumptions

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

—We assume that the embedded programs adhere to the MISRA C coding guidelines. Although MISRA C guidelines were originally intended for the automotive industry, it is now widely being used for developing embedded applications. We list a few important rules to indicate the types of restrictions imposed by MISRA C.
  —*Rule 8.1.* Every function must have a prototype declaration.
  —*Rule 9.1.* All automatic variables are assigned a value before being used.
  —*Rule 12.10.* The comma operator is not used.

—*Rule 12.13.* The increment (++) and decrement (−−) operators are not mixed with other operators in an expression.
—*Rule 14.6.* Only one `break` statement can be used in any iteration construct.
—*Rule 20.4.* Dynamic memory allocation is not used.

We have made two exceptions to the MISRA guidelines in our implementation. MISRA guidelines do not recommend the use of `goto` and `continue` statements to promote structured programming practices. On the other hand, many studies [Knuth 1974; McConell 2004; Kondoh and Futatsugi 2006] have argued that judicious use of `gotos` is beneficial for many types of common programming problems. Embedded programmers still make heavy use of constructs such as `gotos`. Code that is auto-generated using tools, such as MATLAB Real-Time Workshop [Mathworks 2011], also contain unstructured constructs, such as `breaks`. In our work, we have therefore assumed that `goto` (Rule 14.4) and `continue` (Rule 14.5) statements are allowed in the code.

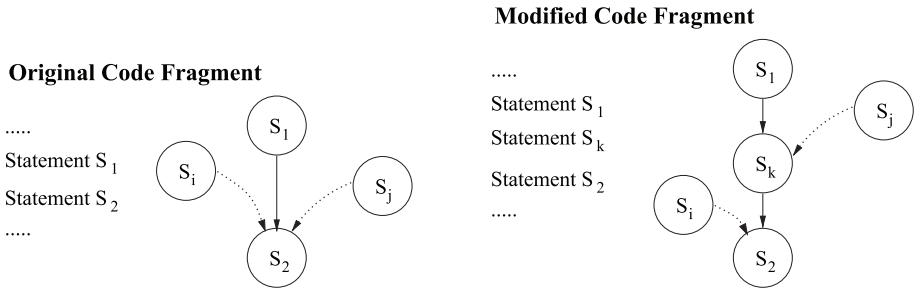
—The tasks can be created both statically or dynamically. Tasks are dynamically created in response to events.
—The tasks are assigned priorities statically and are scheduled using a preemptive and priority-driven operating system.
—At a time, only one instance of a task is active.
—DPCs are assumed to inherit the priority of the tasks which are interrupted.
—The tasks communicate using either shared memory or message passing mechanisms. We further assume that only the synchronous mode of message passing is used.
—Many embedded applications obtain inputs from sensors, and the computed output is used to drive actuators. The analog inputs from the sensors are usually transferred to the program as a set of discrete input data, and the computed output data are transferred to the actuators as analog signals through logic circuits. Without any loss of generality, we ignore the analog signals and consider only their digital counterparts processed by the computer. We therefore assume that the initial test suite T is available as a formatted text file. For each test case $t \in T$, the formatted file contains the following information: a unique identifier assigned to each test case, set of input data, the expected result, and the time constraints, if any.

5.2. Types of Program Changes

An arbitrary change to a program could be any one of the following three types: (1) *addition* of a statement, (2) *deletion* of a statement, or (3) *modification* of a statement. A change to a program P could be confined to a single line or could span multiple lines. A change to P might require addition and deletion of some nodes and edges of the corresponding SDGC model. Any arbitrary modification could be considered to be composed of a deletion operation followed by an addition operation. Therefore, in our work, we assume that addition and deletion are the only two basic change operations. In the following, we identify the changes to the SDGC model required to reflect the changes caused due to the two basic program change operations.

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

—*Addition of Statements.* Adding new statements to P requires creating new nodes and edges in the SDGC model M. The additional edges created could be of types control flow, control or data dependence, *parameter-in*, etc. It may also be required to delete certain existing control flow and dependency edges during edge creation.



(a) Nodes and edges in the original program segment.

(b) Nodes and edges in the modified program segment.

Fig. 13. Effect of addition of a statement on control flow and dependencies.

We give an example of the effect of addition of a statement in Figure 13. Figure 13(a) shows a sample code snippet consisting of two sequential program statements, S_1 and S_2. In the corresponding partial SDGC model, the two nodes (also denoted by S_1 and S_2) are connected using a control flow edge (denoted by a solid edge). The partial SDGC model also shows that the statement S_2 is data dependent on certain other program statements S_i and S_j. Now, suppose a statement S_k is added to the sample code snippet, as shown in Figure 13(a). The corresponding partial SDGC model for the modified program is also shown in Figure 13(b). Due to the addition of the statement S_k, the control flow edge between S_1 and S_2 is now deleted, and instead, two new control flow edges are introduced between the pairs S_1 and S_k, and S_k and S_2. Due to the added statement, the data dependency between S_j and S_2 ceases to exist, and instead, a new data dependency is introduced between the nodes S_j and S_k.

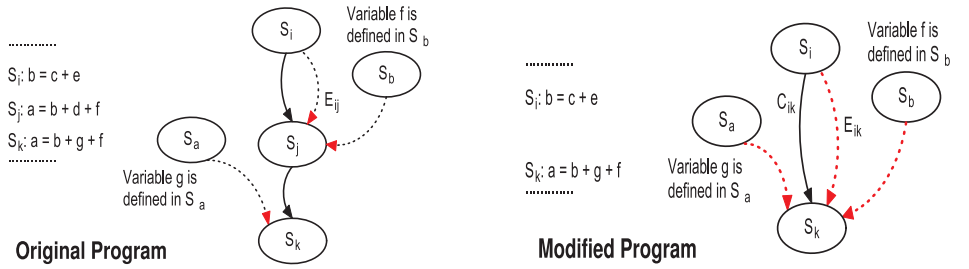
—*Deletion of Statements.* Deletion of one or more statements could affect the dependencies existing among certain other statements, for example, if a statement that defines a variable is deleted, it could lead to a wrong evaluation of a predicate which uses that variable. Therefore, before actual deletion of statements, it is important to identify and mark all those program elements as *affected* which are data dependent on the deleted statement before actual deletion.

Before a statement (i.e., one or more nodes) is deleted, first the other nodes in M that are data or control dependent on the deleted node(s) are identified and are marked as affected. Then, the node(s) in M corresponding to the deleted statement are deleted. The different edges which are incident on or emanate from the node(s) corresponding to the deleted statement are also deleted. In addition, new data- and control-dependency edges can get created on account of the modified dependency relationships.

Figure 14(a) shows a code fragment and the corresponding partial SDGC model consisting of control flow (solid) and data-dependence (dotted) edges. The edge E_{ij} models the data dependency between the nodes S_i and S_j in the original code. Let us assume that the variables f and g are defined, respectively, in the statements represented by the nodes S_b and S_a. Therefore, there exists data-dependence edges from S_b to S_j, and from S_a to S_k. Suppose the statement S_j is deleted, as shown in Figure 14(b). Due to this change, a control flow (C_{ik}) and a data-dependence edge E_{ik} are created between the nodes S_i and S_k. An additional data dependency from S_b to S_k is also introduced, as shown in Figure 14(b).

5.3. Processing Activities in RTSEM

During the maintenance phase of a software, there could be numerous regression testing cycles during which RTS could potentially be carried out. In our approach, we



(a) Nodes and edges for the original program segment.

(b) Nodes and edges for the modified program segment.

Fig. 14. Effect of statement deletion on data dependencies.

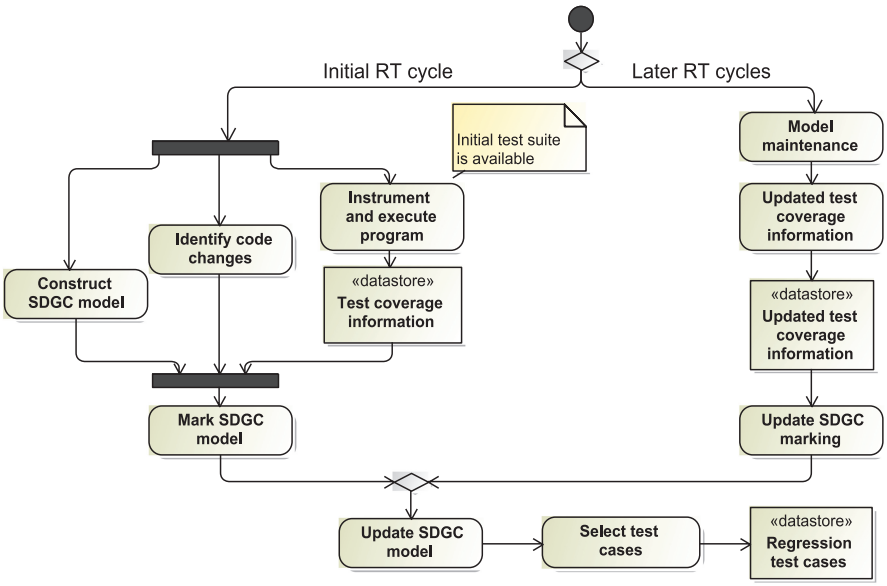


Fig. 15. Activity diagram representation of RTSEM.

divide the regression testing cycles during the maintenance phase into the following: the first regression testing cycle and the subsequent regression testing cycles. This is because some of the processing activities in RTSEM need to be carried out only once during the first cycle and need not be repeated during subsequent RTS cycles. We refer to the first regression testing cycle as the *initial RT cycle*, and the subsequent regression testing cycles as *later RT cycle*s.

The important steps of RTSEM have been represented in the activity model of Figure 15. In the following, we briefly describe the processing steps involved in RTSEM.

5.3.1. Initial RT Cycle. We now describe the main steps involved during the first regression testing cycle.

—*Construct SDGC Model.* In this step, the SDGC model M for the original program P is constructed by using Algorithm 1 presented in Section 4.2.

—*Identify Changes.* In this step, the exact changes that were made to the modified program P' are identified through a semantic analysis of P and P'. Instead of identifying all the structural differences between the two files, our semantic analysis selects only those changes that can affect the output of the program. Therefore, our semantic analysis technique ignores purely structural changes introduced due to re-ordering of functions, changes to the order of the argument list, changes due to commenting, formatting, etc.

The identified statement-level changes between P and P' are stored as a formatted ASCII file which we will refer to as the *diff* file. Each entry in the *diff* file corresponds to a statement-wise change between P and P', and contains the changed program statement in P', the line numbers in P or P', and the function name to which the changed statement belongs.

—*Instrument and Execute Program.* In this step, the original program P is instrumented, for example, by inserting print statements. The instrumentation is done at the level of basic blocks, since instrumentation at the level of basic blocks leads to more efficient execution trace generation and analysis compared to statement-level instrumentation without compromising precision. The instrumented code is executed with the entire test suite T to generate the *execution trace* for each test case. An execution trace of a test case essentially is the set of statements of P that is executed by a test case. Generating the test coverage information is a one-time activity for a given program during one testing cycle, and need not be repeated during the subsequent regression testing cycles. The test coverage information (denoted by $\mathcal{C}$) generated in this step is saved in a file for later processing. This is shown in Figure 15 by the datastore *test coverage information*.

At this point, it should be noted that the execution behavior of a real-time embedded program may change due to instrumentation. The effect of unrestricted instrumentation could introduce additional delays in execution of certain instructions which could result in timeouts, causing the program to take alternate execution paths. In our work, we have tried to mitigate the nondeterminism introduced due to instrumentation by using the instrumented code only for test coverage generation and have stripped the instrumentation from the program while executing the selected regression test cases on the sample test programs. This ensures that the instrumentation will less likely affect the actual results of regression testing.

—*Mark the SDGC Model.* In this step, the test coverage information is marked on M. Marking an SDGC model involves adding additional information to each node in the SDGC model about the test cases that execute the corresponding program statement.

—*Update SDGC Model.* In this step, the SDGC model is updated using information from the *diff* file so as to make it correspond to the modified program P'. The steps required to update the original SDGC model M are explained in Section 5.4. In this context, it is important to note that the updated SDGC model M corresponding to the modified program P' captures any possible change in execution dependencies among the tasks in P'.

The changes between the programs P and P' are also marked on the SDGC model by using the information stored in the *diff* file. This is achieved as follows: for those statements which are deleted from P, the nodes in M which were directly data, or control dependent on the deleted nodes in M (i.e., statements in P) before updating M are affected due to deletion and are tagged as deleted. Once the model M has been updated to correspond to P', for each statement added or modified in P', the nodes corresponding to the changed statements are searched in the updated model M and are tagged as changed. We refer to both the set of tagged nodes by $Tagged$.

—*Select Test Cases*. In this step, the relevant test cases are selected based on control-data, and task-execution-dependency analysis. This step is elaborated in Section 5.5.

The selected regression test cases are represented by the datastore *regression test cases* in Figure 15.

5.3.2. Later RT Cycles. In the following, we explain the steps in RTSEM which are carried out during the regression testing cycles other than the first for program versions P' and P''. Note that we denote the original program version in these cycles by P' and the modified program version by P''.

—*Model Maintenance.* In this step, the markings, tags, and test coverage information added to the SDGC model M during the previous regression testing cycle are deleted. This is achieved by traversing each node in the whole SDGC model and resetting the information.

—*Update Test Coverage Information.* The test coverage information could become outdated across regression testing cycles because of modifications to both the code and to the initial test suite. In each regression test cycle, new test cases get added, and obsolete test cases are deleted from the initial test suite T. This activity is called test suite maintenance. The new test cases are executed with P'' to generate and update the coverage information.

The test coverage information $\mathcal{C}$ generated in the last regression test cycle needs to be updated to take into account the changes made to both the initial test suite and the code. Since obsolete test cases are no longer valid for testing P'', the coverage information corresponding to each obsolete test case is deleted from $\mathcal{C}$. The coverage information for the resolution (generated during resolution testing) and regression test cases (generated during the previous regression test cycle) are added to $\mathcal{C}$. The test coverage information $\mathcal{C}$ is not affected by redundant test cases. The output of this step is shown by the datastore *update test coverage information* in Figure 15.

—*Update SDGC Marking.* The marking on the SDGC model M is updated according to the updated test coverage information $\mathcal{C}$ generated in step *update test coverage information*.

—*Update SDGC Model.* The SDGC model is updated to take into account the changes made to the program P'. Note that the complete SDGC model for P'' is not required to be constructed again in this step. Instead, the SDGC model updation technique discussed in Sections 5.3.1 and 5.4 is used to incrementally update the SDGC model.

5.4. Incremental Updation of an SDGC Model

Many existing RTS techniques construct program models of both the original and the modified programs [Rothermel and Harrold 1997; Harrold et al. 2001]. However, the overhead involved in reconstructing the complete SDGC model for the modified program each time after a change is made can be unacceptably large and should be avoided because often only minor changes are made to P. To overcome this source of inefficiency, we incrementally update the original SDGC model during each regression testing cycle to reflect the changes made to the original version of the program.

The pseudocode *SDGCUpdate* for incrementally updating an SDGC model M is shown in Algorithm 2. *SDGCUpdate* takes the SDGC model M to be updated and the *diff* file (denoted by *DiffFile*). Algorithm 2 computes the set of nodes (denoted by *Tagged*) which contains nodes which are added/modified in the updated model M or nodes which are affected due to deletion of nodes from M. In the following, we discuss how incremental updation of an SDGC model can be achieved by *SDGCUpdate*.

—*Addition.* We create an additional node in M for every new statement that has been introduced in P'. It should be noted that though usually only a single node needs to be created for a new statement, at times, more than one node may need to be created. For example, for a function call statement, one *call-site* node and one or

ALGORITHM 2: Pseudocode to Incrementally Update the SDGC Model

```
1: procedure SDGCUpdate(M, DiffFile, Tagged)
        ▷ M = SDGC model to be updated        ▷ DiffFile = diff file
   ▷ Tagged = set of nodes added/modified in M or affected due to deletion of nodes from M
2:   Tagged ← Φ
3:   for each entry s in DiffFile do
4:     if modification type is addition then
5:       Create node(s) corresponding to s
6:       for each node n created for the statement s do
7:         Tagged = Tagged ∪ n
8:         if n ≠ V_Ain and n ≠ V_Aout and n ≠ V_Fin and n ≠ V_Fout) then
                                  ▷ Node is of type V_assign or V_pred or V_call or V_task, etc.
9:           Connect n with pred_CF and succ_CF nodes using control flow edges
10:          Delete control flow edge between pred_CF and succ_CF node
11:        else
12:          Connect n with the parent call site node using control dependence edges
13:        end if
14:      end for
15:    else                                  ▷ Modification type is deletion
16:      Find node(s) n ∈ M corresponding to s
17:      for each node n corresponding to s do
18:        Find node(s) q data or control dependent on n
19:        Tagged = Tagged ∪ q
20:        Delete all edges emanating from n
21:        Delete node n
22:      end for
23:    end if
24:  end for
25:  Create edges of type E_tdef, E_mp, E_sem, E_tm to connect new nodes
26:  Recompute data dependency information for M
27:  Recompute control dependency information for CFGs modified in M return Tagged
28: end procedure
```

many *actual-in* and *actual-out* nodes may be created. If the newly created node type is not an argument or a parameter of a function, then control flow edges are created to connect the newly created node to its control flow predecessor (denoted by $pred_{CF}$) and successor (denoted by $succ_{CF}$) nodes in the SDGC model. Creation of a new node in the existing SDGC model between $pred_{CF}$ and $succ_{CF}$ requires that any existing control flow edge between $pred_{CF}$ and $succ_{CF}$ be deleted. If the newly created node type is an argument or a parameter of a function, then it is connected to its parent call site node using control-dependence edges.

—*Deletion.* For a statement s which has been deleted from P, we also delete the corresponding nodes from the SDGC model M. However, before deleting a node $n \in M$ corresponding to s, we mark the set of nodes which are data or control dependent on n as affected. Once the affected nodes are marked, the node n is deleted from M. Deletion of a node n from M requires deletion of all the edges (control flow, data dependence, etc.) which are incident on n or emanate from n.

—*Create Additional SDGC Model Edges.* In this step, any additional edge types defined for an SDGC model are created to connect relevant pair of nodes, for example, a V_{qs} and the corresponding V_{qr} nodes are connected using an E_{mp} edge.

—*Dependency Computation.* After the control flow information is updated on model M for all the entries of the *diff* file, we recompute the data- and control-dependency information. The data-dependency information needs to be recomputed for the whole modified program P' to take into account interprocedural data dependencies. This is

done by carrying out data-dependency analysis on the SDGC model M. The control-dependence information is recomputed for only the directly modified functions. Data- and control-dependency computation for a changed function is done by analyzing the CFG corresponding to the function.

5.5. Test Case Selection

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

$$T_{reg} = T_{dep} \cup T_{time}, \tag{1}$$

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

5.5.1. RTS Based on Control and Data-Dependency Analysis. We select regression test cases based on forward slicing using control and data dependencies. For this, we have designed an algorithm to compute the SDGC model slice. Our forward slicing algorithm is an extension of the two-phase SDG slicing algorithm proposed by Bates and Horwitz [1993]. The slicing criterion is the set of nodes in the SDGC model M that are tagged during *update SDGC model*.

We have named our algorithm to slice SDGC models as *SDGCSlice*. The pseudocode of SDGCSlice is shown in Algorithm 3. Algorithm *SDGCSlice* takes as input the marked and updated SDGC model M, the set of nodes tagged in M (denoted by $Tagged$) during the step *update SDGC model*, and computes the set of test cases (T_{dep}) relevant for regression testing. Our SDGC model slice computation essentially performs a reachability analysis using data- and control-dependence edges [Horwitz et al. 1990]. Slicing the SDGC model based on data- and control-dependence edges helps to identify the set of model elements that may get affected due to the modifications. SDGCSlice

ALGORITHM 3: Pseudocode to Select Regression Test Cases by Computing the SDGC Model Slice Based on Control and Data Dependence

```
1: procedure SDGCSlice(M, Tagged, T_dep)
        ▷ M = updated marked SDGC model        ▷ T_dep = selected regression test cases
      ▷ Tagged = set of nodes in M which are added/modified or are affected due to deletion
2:    T_dep ← NULL                                          ▷ Initially, T_dep is empty
3:    Dependent ← NULL
4:    for each node m in Tagged do
                                          ▷ Check for data or control dependence edges from m
5:       Find the nodes control or data dependent on m ∈ M
6:       AffectedSet = AffectedSet ∪ {Set of all nodes control or data dependent on m}
                ▷ Slicing to consider calls related to tasks/timers/message queues/semaphores
      ▷ SDGC traversal to include task definition, timer, message passing, semaphore edges
7:    end for
8:    if AffectedSet ≠ Φ then
9:       for each node n ∈ AffectedSet do
10:         Add the list of test cases that execute n to T_dep
11:      end for
12:   end if
13:   AffectedSet ← NULL
14: end procedure
```

computes the set (denoted by *AffectedSet*) of all nodes affected on account of data- and control-dependence relations by computing the transitive closure of the set *Tagged*. These steps are shown in lines 5 to 13 in Algorithm 3.

Once all the affected SDGC model elements are identified through forward slicing, the test cases executing those model elements are selected for regression testing. This involves traversing the SDGC model and visiting each node $n \in AffectedSet$ to find out test cases which execute n.

5.5.2. RTS Based on Task-Execution-Dependency Analysis. An important step in our RTS technique is the selection of all test cases that test tasks which are affected due to execution dependencies. We have named our algorithm for this step *TimingSelect*. The pseudocode for *TimingSelect* is shown in Algorithm 4.[3] *TimingSelect* takes as input the marked SDGC model M, the set of marked nodes in M (denoted by *Marked*), and produces the selected set of test cases (denoted by T_{timing}) as the output. The algorithm is explained briefly in the following. First, the tasks in P' to which one or more nodes in *Marked* belong are identified as directly modified and are denoted by Γ. Then, for each task $\tau_i \in \Gamma$, *TimingSelect* invokes four functions: *PrecSelect*, *PrioritySelect*, *MPSelect*, and *SemSelect*. These functions compute the set of tasks that are execution dependent on τ_i due to precedence ordering, priorities, and intertask communication using message passing and semaphores, respectively, that is, for each task $\tau_i \in \Gamma$, algorithm *TimingSelect* computes $Succ(\tau_i)$, $Prior(\tau_i)$, $ITC_{mp}(\tau_i)$, and $ITC_{syn}(\tau_i)$, respectively.

The information about the test cases which execute the program statements corresponding to the affected tasks in P' are already stored in the *Start* node of the CFG for the tasks. The test cases executing the affected tasks are selected for regression testing.

The timing dependencies have been defined only among tasks (and not statements), and therefore intra-procedural summary edges are not required to model task-level dependencies. The task precedence edges and message-passing and semaphore edges are sufficient to capture task-level dependencies.

6. EXPERIMENTAL STUDIES

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

6.1. MTest: A Prototype Implementation of RTSEM

`MTest` has been developed using C++ programming language on a Microsoft Windows 7 environment running on a Compaq SG3770IL desktop having a 2.8GHz processor and 2GB main memory. The code size of `MTest` is approximately 17 KLOC, excluding the external packages used. `MTest` currently has a rudimentary user interface developed using Microsoft Visual Basic 6.0. During execution, `MTest` takes a program P, modified program P', and the test suite T as inputs. The test suite T is prepared as a formatted text file containing the test case identifiers. The output produced by `MTest` is a formatted text file containing the identifiers of the test cases selected for regression testing.

[3]Note that the algorithm is split across two pages.

ALGORITHM 4: Pseudocode to Select Regression Test Cases Based on Task Execution Dependencies

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

6.1.1. Open-Source Software Packages Used. We have implemented `MTest` using the following open-source software packages: Eclipse,[4] MinGW,[5] ANTLR,[6] Graphviz.[7] We have used Eclipse CDT (C/C++ Development Tools) as the IDE and MinGW as the C/C++ compiler. An advantage of MinGW is that it gets automatically and seamlessly integrated with Eclipse. We have used ANTLR v2.7.7 as the parser generator and have adapted the ANTLR grammar file for C language.[8] We have used a subset of the grammar rules for C language which are compliant with MISRA C guidelines. We have

[4] `http://www.eclipse.org/`.
[5] `http://www.mingw.org/`.
[6] `http://www.antlr.org/`.
[7] `http://www.graphviz.org/`.
[8] ANTLR C++ grammar. `http://antlr.org/grammar/list`.

ALGORITHM 4 (Continued): Pseudocode to Select Regression Test Cases Based on Task Execution Dependencies

```
36: procedure MPSELECT(M, τ_i)                          ▷ Compute ITC_mp(τ_i)
37:     ITC_mp(τ_i) ← NULL
38:     Traverse M to reach task create node of τ_i
39:     Traverse along task definition edge to the corresponding task function
40:     Traverse the CFG for the function
                          ▷ Check for nodes of type message queue send or message queue receive
41:     if node type is V_ms OR node type is V_mr then   ▷ Task τ_i is a send/receiver of data
42:         Traverse along message queue edge to reach the message passing node (denoted by
        n_dest) of the other task (denoted by τ_j)
43:         Traverse up along the CFG for τ_j starting from n_dest to reach the task create node
44:         ITC_mp(τ_i) = ITC_mp(τ_i) ∪ τ_j              ▷ Add τ_j to ITC_mp(τ_i)
45:     end if
46:     AddTest(T_timing, ITC_mp(τ_i))
47: end procedure
48: procedure SEMSELECT(M, τ_i)                          ▷ Compute ITC_syn(τ_i)
49:     ITC_syn(τ_i) ← NULL
50:     Traverse M to reach task create node of τ_i
51:     Traverse along task definition edge to the corresponding task function
52:     Traverse the CFG for the function
                          ▷ Check for nodes of type semaphore req or semaphore rel
53:     if node type is V_st OR node type is V_sg then   ▷ Task τ_i is a send/receiver of data
54:         Traverse along semaphore edge to reach the semaphore node (denoted by n_dest) of the
        other task (denoted by τ_j)
55:         Traverse up along the CFG for τ_j starting from n_dest to reach the task create node
56:         ITC_mp(τ_i) = ITC_syn(τ_i) ∪ τ_j             ▷ Add τ_j to ITC_syn(τ_i)
57:     end if
58:     AddTest(T_timing, ITC_syn(τ_i))
59: end procedure
```

used ANTLR version 2.7.7, since the grammar has been developed with ANTLR v2.7. ANTLR also gets seamlessly integrated into Eclipse as a plugin. The steps to install the ANTLR plugin in Eclipse is available online.[9] We have used Graphviz to graphically display the SDGC models constructed by `MTest`.

6.1.2. Components of `MTest`. The architecture of `MTest` is shown in the component diagram in Figure 16. From the figure, it can be observed that the primary components of `MTest` are *SDGC model constructor*, *test coverage generator*, *model marker*, and *test case selector*. The ball and socket connections among the components identify the producer and consumer components. For example, the test coverage information generated by the test coverage generator is used by the model marker component. In the following, we briefly describe the roles of the different components of `MTest`.

—*SDGC Model Constructor.* The *SDGC model constructor* implements the algorithm ConstructSDGC presented in Section 4.2 for constructing SDGC models. As shown in Figure 16, the SDGC model constructor component takes P as input and constructs the SDGC model M. SDGC model constructor first constructs CFGs for each function of the input program. Once the construction of the CFGs is complete, the iterative dataflow computation technique described in Aho et al. [2008] is performed on the CFGs to identify the data dependencies. Finally, the CDG for a function is constructed using the approach proposed by Ferrante et al. [1987].

[9] `http://antlreclipse.sourceforge.net/`.

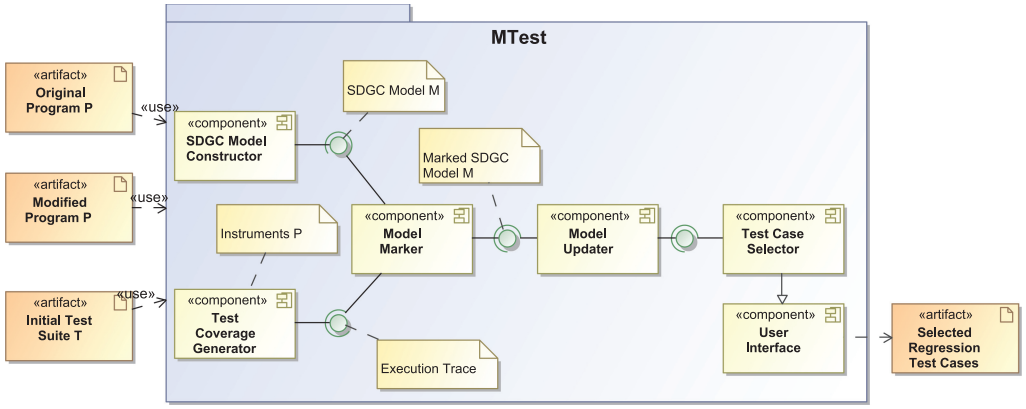


Fig. 16. Component model of `MTest`.

—*Test Coverage Generator.* The *test coverage generator* generates test coverage information $\mathcal{C}$ semi-automatically by executing the input program with all the test cases from the initial test suite. This step is semi-automatic, since embedded programs usually require inputs at fixed time steps, which are input manually by a tester. Our implementation of the test coverage generator makes use of Gcov, an open-source profiling tool.[10] The generated test coverage information is in the form of an ASCII file listing the functions, tasks, and the line numbers covered by each test case in T.

—*Model Marker.* The *model marker* module stores the test coverage information on the SDGC model M. A statement s in P is modeled using one or more nodes in the SDGC M. For each program statement s in P, the SDGC M is traversed hierarchically to search for the corresponding node(s) modeling statement s. The complexity in repeatedly searching the whole SDGC model is addressed using a two-level searching technique. In the first level, only the CFG *Start* nodes in an SDGC model are checked to find the CFG of the function which the node n corresponding to s belongs to. The search time involved in this step is limited to a linear search of the maximum number of functions/tasks defined in the program. In the second level, the CFG is searched to find out node n. Then, for each node corresponding to s, the model marker stores the list of test cases that execute s in the node data structure itself. This technique of storing the test coverage information on the model itself circumvents the use of a database or file storage and improves the efficiency of our approach. This approach of storing both the program and the test coverage information in the memory may be infeasible for very large programs having thousands of test cases. However, this approach is beneficial for programs for small embedded systems.

—*Model Updater.* The *model updater* updates the SDGC model M corresponding to the original program P so that it reflects the changes made to P and is in sync with the modified program P'. The model updater implements the algorithm SDGCUpdate discussed in Section 5.4.

—*Test Case Selector.* The *test case selector* selects regression test cases by using data-, control- and task-execution-dependency analysis, as discussed in Section 5.5. The output of this module is a file containing the identifiers of the test cases selected for regression testing.

[10] `http://gcc.gnn.org/onlinedocs/gcc/Gcor.html`.

Table I. Characteristics of the Programs Used in our Experimental Studies

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

6.2. An Evaluation of the Effectiveness of MTest

The aim of our experimental studies using `MTest` was to evaluate the performance and effectiveness of our RTS approach (RTSEM). An intuitive and appealing metric for evaluating the effectiveness of an RTS technique is the size of the selected regression test suite. Obviously, it is desirable to have this number as small as possible. However, for effective RTS, it is more important for a technique not to miss out selecting any fault-revealing test cases, and at the same time, to minimize instances of false positives. Therefore, we have defined a new metric called *fault-revealing effectiveness*. In the following, we briefly describe these two metrics with which we evaluated the effectiveness of RTSEM.

—*Percentage of Test-Cases Selected for RTS (Υ).* This measure indicates the size of the regression test suite as a percentage of the initial test suite.
—*Fault-Revealing Effectiveness (Ψ).* The fault-revealing effectiveness metric can be defined as the percentage of test cases selected by an RTS technique from the set of test cases that fail when the valid test cases in the initial test suite are run. That is, the fault-revealing effectiveness of the test suite selected by a safe RTS technique is equal to 100%, that is, it is equal to that of the initial test suite.

6.3. Experiments

We have used eight programs from the automotive control domain for our experimental studies. These applications include a simplified adaptive cruise controller, power window controller, and climate controller. These C programs have been auto-generated from Simulink models using the Real-Time Workshop tool in MATLAB [Mathworks 2011] . The size of the uncommented source lines in the programs range from 156 to 737 LOC. Table I summarizes the average size of the sample programs (LOC) and that of the corresponding SDGC models in terms of the number of nodes and edges. A snapshot of the SDGC model for the Climate Controller program is shown in Figure 17. The figure was generated using the dotty tool of Graphviz. The solid edges in the figure represent control flow edges, while the other SDGC model edge types are annotated in the figure. We have shown only a partial view of the model in Figure 17 to avoid clutter.

For each program, we systematically created several modified versions by adding, modifying, or deleting one or more lines of code, or by making a change to the Simulink model and then auto-generating the code. In order to avoid the possibility of making unrealistic changes to programs, we consulted several industry professionals involved in Simulink/Stateflow-based embedded program development. The relative frequency of occurrence of the different types of changes that we introduced are based on feedback from the industry experts. These are categorized into three levels: Extremely frequent, Frequent, and Less frequent. In Table II, we list the different types of modifications

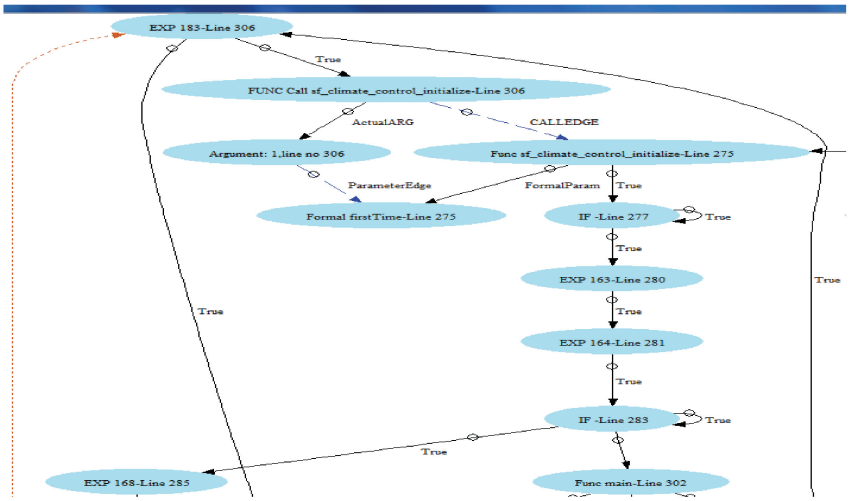


Fig. 17. A screenshot of the SDGC model for the climate controller program.

Table II. Types of Program Modifications and Their Relative Frequencies

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

that we applied to the programs or to the models and their relative frequencies. Based on the feedback from some practitioners from the industry on the types of changes usually made, we introduced the following types of modifications to the programs: (a) a change is introduced in a Simulink model block and then the code is auto-generated to realize the modified program; (b) a modification is made directly to the auto-generated code. The changes made to the Simulink models resulted in auto-generated code that had new functions, modified function prototypes, etc. An example of a change of type (a) is changing the value of a gain or a constant block or adding new inputs to a block; and that for changes of type (b) are changed predicates, changed datatype of variables, and delays to tasks.

We designed test cases for each program to test the functional and temporal correctness of the programs. The functional test cases were designed using blackbox techniques of category partitioning and boundary value analysis, and performance test cases were designed to check whether the timing constraints of tasks are met. The test cases were executed to generate the test coverage information. The modifications made to the modified program versions were based on the types of changes listed in Table II. For example, for a change of type *predicate change*, one predicate was randomly selected for change. For each program, all the test cases from the initial test suite were run with each modified program version to find out the number of test cases that failed, that is, produced incorrect results when run with the modified program. Each time, after making a change, regression test cases were selected using `MTest` and the number of regression test cases that failed was determined. To compare the performance

Table III. Summary of Experimental Results

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

Table IV. Summary of Results on Fault-Revealing Effectiveness

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

and effectiveness of our approach with an established approach for RTS of procedural programs, we also selected regression test cases using the SDG-based RTS approach proposed by Binkley [1997]. We have chosen Binkley's approach since we were unable to find any RTS technique that was specifically designed for embedded programs or any recent approach that advances Binkley's approach in nontrivial ways. We have tried to remove any bias in the results by carrying out each experiment ten times with different changes each time and then averaging the results. This also helped to remove any bias introduced in the results due to selection of only a specific type of change.

6.4. Results and Analysis

The results obtained from our experiments have been summarized in Tables III and IV. Table III shows the value of the metric Υ for `MTest` and Binkley's approach. The first column in Table III shows the type of programs that were tested. Column 2 shows the number of test cases that were used to test the modified programs. Column 3 shows the number of test cases that were selected (as a percentage) on the average by `MTest` from the initial test suite. Column 4 shows the number of test cases that were selected using Binkley's approach [1997]. Column 5 shows the difference in the number of regression test cases selected by the two approaches as a percentage of the number of regression test cases selected by Binkley's approach.

The results of Table III have been presented in the form of a bar graph in Figure 18. In the figure, the y-axis shows the percentage of selected test cases while the labels on the x-axis represent the different programs. It can be observed from Table III and Figure 18 that `MTest` selected around 45% to 65.22% of test cases for regression testing

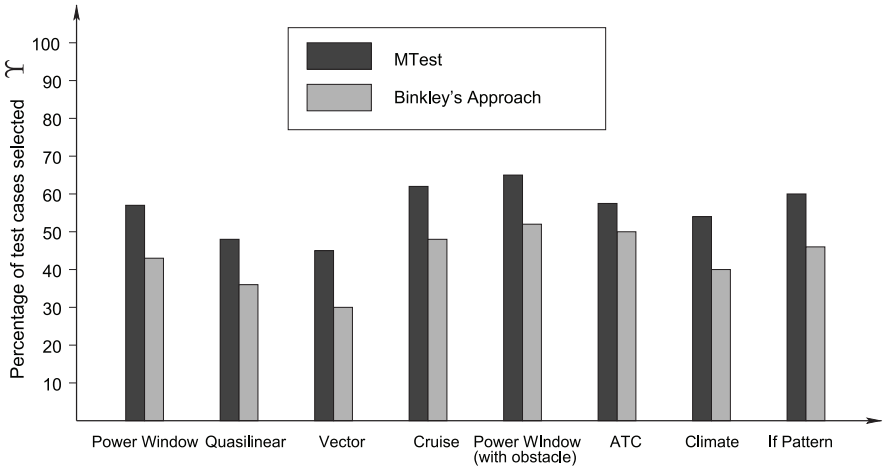


Fig. 18. Percentage of regression test cases selected (Υ).

of the modified programs. Considering the results for all the programs, the number of test cases selected by `MTest` was on average 28.33% greater than Binkley's approach [1997]. This increase can be explained by the fact that, in addition to data and control dependence, our approach also selects test cases based on task-execution dependencies that are ignored by Binkley's approach.

We now give an example to highlight the type of test cases which were omitted by Binkley's approach. In a typical ACC implementation, the host vehicle speed controller (HVSM) task is execution dependent on the radar information processing (RIP) task due to precedence ordering. In the modified version of the ACC program, the RIP task was modified which subsequently delayed the completion of the HVSM task causing the HVSM task to timeout. For such a modification, Binkley's approach failed to select test cases that tested the performance constraints of the HVSM task because there were no data and control dependencies between the RIP and HVSM tasks.

Table IV shows the fault-revealing effectiveness (Ψ) of `MTest` and Binkley's approach. Column 2 in Table IV lists the total number of fault-revealing test cases in the initial test suite as determined by running the entire test suites on the modified programs. Columns 3 and 4 present the the fault-revealing effectiveness of `MTest` and Binkley's approach. Column 5 shows the percentage difference between the two. Only the summary data of the average of the ten systematically selected changes of a given type have been presented in Table IV.

The results of Table IV have been presented as a bar graph in Figure 19. In the figure, the y-axis shows the percentage of failed test cases selected while the labels on the x-axis represent the different programs. The results show that `MTest` is able to select all the fault-revealing test cases present in T. In other words, the regression test suite selected by `MTest` has the same fault-revealing effectiveness Ψ as the initial test suite. The fault-revealing effectiveness of Binkley's approach is lower by 36.36% on average compared to `MTest`.

Discussion on Safety. RTSEM is safe in selecting test cases based on only control and data dependencies. However, the technique may not be safe for selecting all potentially fault-revealing test cases given the arbitrary number of interleavings possible among concurrent tasks.

Another challenge lies in nonintrusively instrumenting embedded programs to minimize the impact on the execution of the tasks. However, some of the task-execution

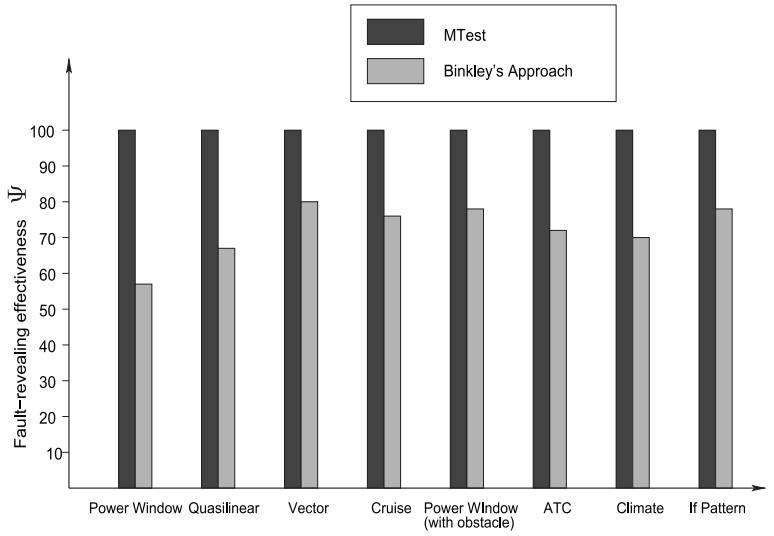


Fig. 19. A comparison of the fault-revealing effectiveness (Ψ) of RTSEM and Binkley's approach.

dependencies discussed in Section 3 are instrumentation-neutral, for example, task-execution dependency due to precedence order and priorities. If a task τ_i precedes task τ_j or is of higher priority, then RTSEM conservatively selects all test cases to test the ordering of the tasks. Nonintrusively instrumenting programs to monitor task dependencies introduced due to message passing or synchronization primitives is a more challenging problem, since the order of accesses are often nondeterministic.

6.5. Threats to Validity

Even though we have carefully developed the experimental setup and consciously tried to avoid various sources of errors, there exist many sources of risks that threaten the validity of our results. We have considered only eight embedded C programs from the automotive domain, and the results that we have obtained during our experimental studies are limited to programs of a maximum size of approximately 740 LOC. Some of the case studies we have chosen are realistic, since they have been developed based on industry-relevant applications, and the others are demo models available from the MATLAB distribution. The number of experimental programs could have been higher, but we intended the programs to be based on realistic applications. Though the size of the programs agrees with our objective of selecting regression test cases for small embedded applications, it would still be interesting to study the results obtained when `MTest` is applied to select regression test cases for more complex and larger embedded programs having large test suites.

Other threats to validity pertain to the implementation of the two prototype tools: `MTest` and the tool implementing Binkley's approach. To address the issues stemming from defective implementation, we have tested the different components in our prototype tools, such as the SDGC model constructor, before carrying out experimental studies. Another source of incorrect results could have been unrealistic changes to the programs giving rise to biased results. However, as discussed in Section 6.4, we have tried to remove any bias in our studies by carrying out each experiment ten times with different changes each time and then averaging the results.

7. COMPARISON WITH RELATED WORK

In spite of our best efforts, we did not find any reported results on RTS of embedded applications. However, a few results have been reported on other aspects concerning regression testing of embedded programs in general [Netkow and Brylow 2010]. Netkow and Brylow [2010] have proposed a framework called Xest for automating the execution of regression test cases in a test-driven development environment. Their test setup helps to automatically execute regression test cases for kernel development projects on embedded hardware. However, their work does not address the problem of RTS and is, therefore, not directly related to our work.

In the absence of any directly comparable work, we compare our technique with a few important procedural RTS techniques that have indirect bearing on our work. Existing procedural RTS techniques [Vokolos and Frankl 1997; Rothermel and Harrold 1997; Binkley 1997] select regression test cases based mainly on analysis of either one or more of the following relations among program entities: control flow, control dependence, and data dependence. These techniques are targeted for procedural programs and therefore ignore features specific to embedded programs, such as tasks, timers, intertask communication, task precedence, exceptions, etc. As a result, these techniques completely ignore execution dependencies that might be existing among tasks during RTS. Consequently, these techniques are likely to omit test cases that can expose timing errors. Our RTS technique models tasks, task precedence ordering, task priorities, intertask communication, timers, and exception handling using an extended SDG model. To model tasks, we capture control flow information in addition to data and control dependencies. Apart from selecting test cases based on data and control dependencies, our RTS technique also selects test cases based on task execution dependencies that are identified by analyzing task precedence, task priority, and intertask communication using message queues and semaphores. Experimental studies conducted by using a prototype implementation of our approach shows that on an average, an additional 28.33% test cases were selected for regression testing and there was a 36.36% increase in the fault-revealing effectiveness as compared to existing techniques [Binkley 1997]. In fact, our proposed RTS technique, RTSEM, achieved 100% fault-revealing effectiveness in our experimental studies.

8. CONCLUSION

Existing RTS techniques largely ignore the implications of important embedded program features, such as time-constrained tasks, task precedences, intertask communication, timers, and exception handling, and as a consequence ignore the execution dependencies that might arise among the tasks. In order to consider execution dependencies among tasks in RTS, we have proposed an augmented SDG model called SDGC. Our proposed RTS technique RTSEM selects relevant regression test cases by analyzing the SDGC model. During our experimental studies, we observed an increase in the number of selected regression test cases by approximately 28.33%. We also observed an increase of 36.36% in the fault-revealing effectiveness of RTSEM as compared to existing RTS approaches. A promising aspect of RTSEM is that it did not miss out on selecting fault-revealing test cases for regression testing during our experimental studies. These results highlight the necessity of modeling tasks and other embedded program features and of incorporating task execution dependency analysis in RTS of embedded programs.

Our work is mainly targeted for RTS of small embedded programs. For these programs, a few simplifying assumptions, such as static task creation and static task priorities, and synchronous message passing among tasks hold. We plan to investigate task execution dependencies that might arise due to dynamic creation of tasks and

asynchronous message passing in large and complex programs. We also intend to extend our technique to take into account the dependencies introduced among elements of embedded programs (such as automobile infotainment applications) that are developed using object-oriented development techniques, such as UML and C++.

REFERENCES

AHO, A., SETHI, R., AND ULLMAN, J. 2008. *Compilers: Principles, Techniques and Tools* 2nd Ed. Dorling Kindersley (India) Pvt Ltd.

ALLEN, M. AND HORWITZ, S. 2003. Slicing java programs that throw and catch exceptions. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'03)*. ACM, New York, NY, 44–54.

BATES, S. AND HORWITZ, S. 1993. Incremental program testing using program dependence graphs. In *Proceedings of the Conference Record of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, New York, NY, 384–396.

BINKLEY, D. 1997. Semantics guided regression test cost reduction. *IEEE Trans. Softw. Eng. 23,* 8, 498–516.

BISWAS, S. 2011. Model-based regression test selection and optimization for embedded programs. M.S. thesis, Indian Institute of Technology, Kharagpur, India.

BISWAS, S., MALL, R., SATPATHY, M., AND SUKUMARAN, S. 2009. A model-based regression test selection approach for embedded applications. *ACM SIGSOFT Softw. Eng. Notes 34,* 4, 1–9.

BISWAS, S., MALL, R., SATPATHY, M., AND SUKUMARAN, S. 2011. Task dependency analysis for regression test selection of embedded programs. *IEEE Embed. Syst. Lett. 3,* 4, 117–120.

CARTAXO, E., MACHADO, P., AND NETO, F. 2011. On the use of a similarity function for test case selection in the context of model-based testing. *Softw. Test. Verification Reliab. 21,* 2, 75–100.

CLEVE, A., HENRARD, J., AND HAINAUT, J. 2006. Data reverse engineering using system dependency graphs. In *Proceedings of the 13th Working Conference on Reverse Engineering*. IEEE Computer Society, Los Alamitos, CA, 157–166.

DO, H., MIRARAB, S., TAHVILDARI, L., AND ROTHERMEL, G. 2010. The effects of time constraints on test case prioritization: A series of controlled experiments. *IEEE Trans. Softw. Eng. 36,* 5, 593–617.

FERRANTE, J., OTTENSTEIN, K., AND WARREN, J. 1987. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst. 9,* 3, 319–349.

GUAN, J., OFFUTT, J., AND AMMANN, P. 2006. An industrial case study of structural testing applied to safety-critical embedded software. In *Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering*. ACM, New York, NY, 272–277.

HARROLD, M., JONES, J., LI, T., LIANG, D., ORSO, A., PENNINGS, M., SINHA, S., SPOON, S. A., AND GUJARATHI, A. 2001. Regression test selection for java software. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*. ACM, New York, NY, 312–326.

HATLEY, D. AND PIRBHAI, I. 1987. *Strategies for Real-Time System Specification*. Dorset House Publishing Company.

HORWITZ, S., REPS, T., AND BINKLEY, D. 1990. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst. 12,* 1, 26–61.

JIANG, S., ZHOU, S., SHI, Y., AND JIANG, Y. 2006. Improving the preciseness of dependence analysis using exception analysis. In *Proceedings of the 15th IEEE International Conference on Computing*. IEEE Computer Society, Los Alamitos, CA, 277–282.

KAPFHAMMER, G. 2004. *The Computer Science Handbook* 2nd Ed. CRC Press, Boca Raton, FL, (Chapter on Software Testing.)

KNUTH, D. 1974. Structured programming with go to statements. *ACM Comput. Surv. 6,* 4, 261–301.

KONDOH, H. AND FUTATSUGI, K. 2006. To use or not to use the goto statement: Programming styles viewed from Hoare Logic. *Sci. Comput. Program. 60,* 1, 82–116.

LEUNG, H. AND WHITE, L. 1989. Insights into regression testing. In *Proceedings of the Conference on Software Maintenance*. 60–69.

LIANG, D. AND HARROLD, M. 1998. Slicing objects using system dependence graphs. In *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society, Los Alamitos, CA, 358–367.

MALL, R. 2007. *Real-Time Systems Theory and Practice* 1st Ed. Pearson Education.

MARWEDEL, P. 2007. *Embedded System Design*. Springer.

MATHWORKS. 2011. MATLAB. `http://www.mathworks.com`.

McConell, S. 2004. *Code Complete: A Practical Handbook of Software Construction* 2nd Ed. Microsoft Press.

Netkow, M. and Brylow, D. 2010. Xest: An automated framework for regression testing of embedded software. In *Proceedings of the Workshop on Embedded Systems Education (WESE'10)*. ACM, New York, NY, 7:1–7:8.

Orso, A., Shi, N., and Harrold, M. 2004. Scaling regression testing to large software systems. In *Proceedings of the 12th ACM SIGSOFT 12th International Symposium on Foundations of Software Engineering*. ACM, New York, NY, 241–251.

OSEK. 2001. OSEK/VDX time-triggered operating system specification 1.0. `http://portal.osek-vdx.org`.

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

Rothermel, G. and Harrold, M. 1996. Analyzing regression test selection techniques. *IEEE Trans. Softw. Eng. 22,* 8, 529–551.

Rothermel, G. and Harrold, M. 1997. A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Methodol. 6,* 2, 173–210.

Sales, J. 2005. *Symbian OS Internals: Real-Time Kernel Programming*. John Wiley & Sons.

Salewski, F. and Taylor, A. 2007. Fault handling in FPGAs and microcontrollers in safety-critical embedded applications: A comparative survey. In *Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering (ISESE'06)*. 124–131.

Sangiovanni-Vincentelli, A. and Natale, M. D. 2007. Embedded system design for automotive applications. *IEEE Computer 40*, 42–51.

Schotland, T. and Petersen, P. 2011. Exception Handling in C without C++. `http://www.on-time.com/ddj0011.htm`.

Seo, J., Ki, Y., Choi, B., and La, K. 2008. Which spot should I test for effective embedded software testing? In *Proceedings of the 2nd International Conference on Secure System Integration and Reliability Improvement (SSIRI'08)*. IEEE Computer Society, Los Alamitos, CA, 135–142.

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

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

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

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

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

Vokolos, F. and Frankl, P. 1997. 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)*. Chapman & Hall, Ltd., London, 3–21.

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

Wind River Systems. 2010. Wind River VxWorks: Embedded RTOS with support for POSIX and SMP. `http://www.windriver.com/products/vxworks/`.

Zheng, J., Robinson, B., Williams, L., and Smiley, K. 2006. Applying regression test selection for COTS-based applications. In *Proceedings of the 28th International Conference on Software Engineering (ICSE'06)*. ACM, New York, NY, 512–522.

Received August 2011; revised May 2012; accepted September 2012