A Hybrid Regression Test Selection Technique for Object-Oriented Programs

Chhabi Rani Panigrahi and Rajib Mall
Department of Computer Sc. & Engg., IIT Kharagpur, India-721302
chhabi@cse.iitkgp.ernet.in, rajib@cse.iitkgp.ernet.in

Abstract
We propose a regression test selection technique that is based on analysis of both the source code of an object-oriented program as well as the UML state machine models of the affected classes. We first construct a dependency graph model of the original program from the source code. When the program is suitably modified, the constructed model is updated to reflect the changes. Our model in addition to capturing control and data dependencies represents the dependencies arising from object-relations. To find the model elements affected due to a program change, we construct a forward slice of the constructed graph model, where each changed model element is used as a slicing criterion. Subsequently, we determine the affected methods from an analysis of the state machine models based on the changed statements. The test cases that exercise the affected model elements in the program model as well as the transitions caused by the affected methods in state machine models are selected for regression testing. Empirical studies carried out by us show that our technique selects on an average of 27.89% more fault-revealing test cases compared to a purely dependence graph model-based technique while incurring about 38.21% increase in regression test suite size.

Keywords: Software maintenance, Regression testing, Regression test selection, State machine model

1. Introduction
Regression testing is an expensive maintenance activity and is carried out after each modification to a software [14, 18]. Regression Test Selection (RTS) is carried out to ensure that changes do not adversely affect unmodified portions of the software. It often accounts for almost half of the software maintenance costs [18]. Regression testing is carried out at various phases of software development life cycle such as, at unit, integration, system testing as well as during maintenance phase [18]. RTS techniques help to reduce the time and effort required to carry out regression testing.

RTS techniques based on analysis of both source code [11, 25, 24] and model [27, 8, 21] have been proposed in the literature for object-oriented software. Many RTS techniques first construct either the control flow [11, 24] or the dependency representation [25] of programs based on code analysis and then select test cases. These techniques compare the original and modified versions of the program model and select test cases that execute the affected model elements. However, these techniques ignore the fact that state behavior of objects may get affected due to code changes and the affected state behavior needs to be analyzed to select appropriate test cases. In case of model-based RTS techniques, regression test cases are selected by comparing the original model with the model of the modified program [27, 8, 21]. A problem with this approach is that models being abstraction after all, are often insensitive to minor code changes.
It has been argued that state-dependent behavior of objects is a prominent issue that needs to be addressed during RTS of object-oriented software [6]. A code modification may affect the state behavior of an object. For example, a code modification can lead to a sneak transition path to a state [6]. During regression test selection, the state behavior of an object cannot easily be analyzed from code analysis [16]. On the other hand, the state model of an object is usually constructed during the design phase of software development life cycle. Existing code-based RTS techniques for object-oriented programs [11, 25, 24] by and large ignore the state behavior of objects. Unless the state behaviors of objects are taken into consideration, several fault-revealing test cases may be omitted during RTS. In this context, we propose an RTS technique that considers control and data dependence information and dependencies arising due to object-relations as well as the state behavior of objects as documented in state models during RTS of object-oriented programs.

This paper is organized as follows: In Section 2, we discuss certain background concepts that provide the basic details needed to understand our approach. We explain our proposed approach in Section 3 and RTS in Section 4. We describe our empirical study in Section 5. Review of the existing RTS techniques and comparison with our approach is presented in Section 6 and finally conclude the paper in Section 7.

2. Background Concepts

In this section, we review some important concepts that form the basis of our work. We first discuss an established model of object-oriented programs based on which our model has been developed. Then we discuss some definitions and concepts used in the context of RTS in our approach.

2.1 SDG of Object-oriented Programs

System Dependence Graph (SDG) was first introduced by Horowitz et al. and was used to model procedural programs [12]. Later on, SDG was extended by Larsen and Harrold to model object-oriented programs [17]. In the rest of the paper, we refer to Larsen and Harrold's SDG as LH-SDG which is the basis of our proposed model.

An LH-SDG is a directed, connected graph \( G = (V, E) \), consisting of a set \( V \) of vertices and a set \( E \) of edges. In the following, we describe the different types of vertices and edges in an LH-SDG.

LH-SDG Vertices

- **Statement** vertices: Program statements that are present in the body of the methods are represented by *statement* vertices. There are two types of statement vertices: *simple statement* vertices and *call* vertices. Method call statements (call sites) are represented by *call* vertices, whereas all other program statements such as assignments, loops and conditionals are represented by *simple statement* vertices.

- **Entry** vertices: In an LH-SDG, classes and methods have *entry* vertices. A class entry vertex represents an entry into a class and a *method entry* vertex represents an entry into a method.

- **Parameter** vertices: These are used to represent parameter passing between a caller and a callee method. The parameter vertices are of four types. These include *formal-in*, *formal-out*, *actual-in*, and *actual-out*. The *actual-in* and *actual-out* vertices are created for each *call* vertex and *formal-in* and *formal-out* vertices are created for each method entry vertex.
• **Polymorphic choice** vertex: This vertex is used to represent dynamic choice among the possible bindings in a polymorphic call.

**LH-SDG Edges**

• **Data dependence edge**: Data dependence edges are used to represent the data dependence relations existing among different statement vertices.

• **Control dependence edge**: A control dependence edge is used to represent control dependence relations between two statement vertices.

• **Call edge**: A call edge is used to connect a call site to a method entry vertex and also to connect various possible polymorphic method call vertices to a polymorphic choice vertex.

• **Parameter dependence edge**: Parameter dependence edges are used for passing values between actual and formal parameters in a method call. Parameter dependence edges are of two types: parameter-in and parameter-out edges.

• **Summary edge**: A summary edge represents the transitive dependence between actual-in and actual-out vertices.

• **Class member edge**: A class member edge is used to represent the membership relation between a class and its methods. A class entry vertex is connected to a method entry vertex by using a class member edge.

A class is represented in an LH-SDG by a Class Dependence Graph (CIDG) [25]. Each method in a CIDG is represented by a procedure dependence graph [10]. The root node of a CIDG is represented by a class entry vertex. Each method in a class has a method entry vertex. The class entry vertex is connected to the method entry vertex for each method in a class by class member edges. In a CIDG, the method calls are represented by call vertices. A formal-in vertex is added for each formal parameter in the method, and one formal-out vertex for each formal reference parameter that is modified by the called method.

The methods of a class can interact among each other or with methods of other classes in an object-oriented program. In a CIDG, a method call is represented by a call vertex. For each method call vertex, the actual-in and actual-out vertices as well as formal-in and formal-out vertices are created for each called method. The actual-in parameter vertices are connected to the corresponding formal-in vertices in the called method by parameter-in edges. The formal-out vertex of the called method is connected to the corresponding actual-out vertex at the calling method by a parameter-out edge. A CIDG represents effects of return statements by connecting each return statement to its corresponding call vertex using a parameter-out edge. Similarly, for the actual-in parameters that may affect the returned value, summary edges are added between the actual-in vertices and the actual-out vertex. For a derived class, the representation of the base class method is reused for representing the inherited methods. Apart from connecting the class entry vertex of a class to its method entry vertices of locally defined methods, class member edges are constructed to connect the class entry vertex of the derived class to the method entry vertices of the methods inherited by the derived class.

In an object-oriented program, a class may instantiate another class. When a class \( C_1 \) instantiates another class \( C_2 \), an implicit call to the constructor of \( C_2 \) is made. To represent instantiation of a class \( C_2 \) by \( C_1 \) in LH-SDG, a call vertex is created in \( C_1 \). A call edge connects this call vertex to the constructor of \( C_2 \). When a method \( m_1 \) in a class \( C_i \) calls a public method \( m_j \) of another class \( C_2 \), the call vertex in \( C_i \) is connected to the entry vertex of
By a call edge. Figure 1 shows a sample Java program and the partial LH-SDG representation of the instantiation of the class Calculator in the class AdvancedCalculator.

An LH-SDG can also represent polymorphic relationships between a calling method and its called methods. A polymorphic method call in an object-oriented program implies that the destination of a method call is determined at run-time. An LH-SDG uses a polymorphic choice vertex to represent the dynamic choice among the possible destinations. The polymorphic choice vertex has call edges that are connected to the sub graphs representing calls to each possible destination.

2.2 Regression Testing Cycle

Do et al. reported a maintenance process model that is followed during maintenance phase of software development life cycle [9]. According to their process model, after a software is released, the failure reports and change requests from the user are accumulated using a defect tracking system [3]. At suitable intervals the program is modified to address all change requests and failure reports. After the required changes have been carried out, resolution testing is performed to validate the modified parts of the code and regression testing is carried out to revalidate the unmodified parts of the code. The failures identified by the regression testing are fixed and this cycle is repeated until there are no more regression test failures. Each cycle of regression testing and corresponding bug fixes is referred to as a regression testing cycle. After resolution and regression testing are successfully completed, a new version of the software is released, which then undergoes similar maintenance cycles. Please note that, there can be multiple regression testing cycles before every release of a software. A regression testing cycle comprises of a set of activities that are undertaken after every maintenance activity.

2.3 Effectiveness of a Regression Test Suite

A regression test suite should include only that subset of original test suite that is likely to detect a regression error. To determine the effectiveness and quality of a regression test suite, Rothermel et al. have defined the concept of fault-revealing test cases for a program $P$ [23].
Let $P$ and $P'$ be the original and modified programs respectively and $T$ be a test suite designed for $P$. According to Rothermel et al. a test case $t \in T$ is said to be fault-revealing for $P'$, if it causes $P'$ to fail by producing incorrect outputs when executed with $t$ [23].

2.4 Program Slicing

The concept of a program slice was first introduced by Weiser to aid debugging of programs [28]. Since then a significant amount of research results on code slicing have been reported in the literature [26]. A program slice consists of all those program statements that can potentially affect the values computed at some point of interest called the slicing criterion [26, 12, 17]. A backward program slice at a program point $p$ with respect to a variable $v$ contains all statements in the program, including conditionals, that might affect the value of $v$ at $p$ [12], whereas a forward program slice at a program point $p$ with respect to a variable $v$ contains all statements in the program, including conditionals, that might be affected by any modifications to $v$ at $p$ [12]. A comprehensive survey on program slicing techniques has been reported in [20].

2.5 State Machine Diagram

UML 2 state machine diagrams can be used to represent the state behavior of a class. It represents the various states an object may assume during its life time, the transitions allowed at each state, the events that cause transitions to occur, and the actions that take place in response to events. States, events, transitions and actions are the primary constituents of a state machine diagram [7]. The state of an object is usually determined by the values that certain attributes may assume. Conceptually, an object remains in a state, until an event causes it to transit to another state. An event is an occurrence of a stimulus that may trigger an object to change its state. However, the same event can have different effects in different states. A transition is a progression from one state to another and is triggered by an event that is either internal or external to the object. A transition may have an associated guard condition - a Boolean expression that must be true for the transition to take place. Finally, an action is usually a computational procedure that typically involves invocation of an operation.

Actions of a state are triggered by transitions to and from the state or by events raised while in the state. Actions have been categorized into: entry actions, exit actions and doActivity actions. Entry actions are the operations that are carried out when an object enters into a state. Exit actions, on the other hand, are the actions performed when an object exits from a state. Do actions execute after the entry actions and can run as long as a state is active. A transition has a source state and a destination state. We refer to the destination state of a transition $e$ as $D_e$.

3. H-RTS: Our Proposed Approach

We have named our proposed approach for regression test case selection as H-RTS (Hybrid Regression Test Selection). Our technique selects regression test cases based on an analysis of control and data dependencies as well as dependencies arising from object-relations and state machine models. In the following, we describe the important activities that are carried in H-RTS. As mentioned in Section 2.2, the maintenance phase consists of multiple maintenance cycles, and in each maintenance cycle there can be many regression testing cycles. RTS is an important activity carried out in each regression testing cycle. The important steps of our approach H-RTS carried out in the first regression test selection cycle
have been shown in Figure 2 using an activity diagram. As shown in Figure 2, the important activities in the first regression test selection cycle include constructing LH-SDG model, collecting test coverage information and marking the test coverage information in LH-SDG model are not repeated for subsequent regression test selection cycles in our approach. We now describe the different activities that are carried out during the first regression testing cycle.

**Figure 2. Activity Diagram Representation of H-RTS**

- **Construct LH-SDG model**: In this step, the LH-SDG model for the original program $P$ is constructed using a technique similar to that reported in [17].

- **Identify changes**: The changes between $P$ and the modified program $P'$ are identified through semantic analysis and the identified statement-level changes are kept in a file named as `differ`. Each entry in `differ` file contains the changed statement in $P'$, the line number in $P$ or $P'$, the name of the method and the class to which the changed statement belongs. This is shown by the data store `Change information` in Fig. 2. The types of changes that are made to a program in our approach is described in more details in Section 3.1.

- **Instrument and execute the program**: In this step, $P$ is instrumented by inserting print statements and instrumentation is done at basic block level. The print statements are inserted to collect test coverage information. The instrumented code is executed with the original test suite $T$ to generate information, that is, which statements are executed for each test case. The instrumented statements are stripped during regression test execution, so that the original program behavior is not affected. The test coverage information generated in this step denoted by $Covg$ is logged in a file, and is saved for later processing. This is represented by the data store `Test coverage information` in Figure 2.

- **Mark the LH-SDG model**: The test coverage information is marked on LH-SDG model. That is, the specific test cases that execute the specific program statement(s) are recorded in the corresponding node(s) of the LH-SDG model.
• **Update the LH-SDG model:*** In this step, the model constructed for original program \((P)\) is updated during each regression testing cycle to make it correspond to the modified program \((P')\) using information stored in file *differ*.

Also, the changes that are made to \(P\) are marked on the LH-SDG model. The forward slice is constructed during regression test selection step based on this information. The description of updation in LH-SDG model is discussed in more detail in Section 3.2.

• **Determine affected methods:*** In this step, the methods that are affected due to a change in code are determined from state machine models. For that, only the state machine models of changed classes are taken. This step is described in more detail in Section 3.3.

• **Select test cases:** In this step, regression test cases are selected both based on analysis of LH-SDG model as well as separate analysis of the state machine model of the changed classes. This step is discussed in more detail in Section 4.

### 3.1 Types of Program Changes

The changes made to a program may be one or more of the following types: (1) *addition* of a statement, (2) *deletion* of a statement, or (3) *modification* of a statement. A modification operation can be expressed as a deletion operation followed by an addition operation. Hence, in our RTS technique, we assume that addition and deletion are the two basic types of changes that are made to a program.

Any changes made to a program may affect the dependency relations already existing in the program. Addition of a statement to a program requires creation of new nodes and edges in the LH-SDG model and deletion of some existing edges. This is explained in the following with an example. Figure 3(a) shows a code segment and the corresponding partial LH-SDG model consisting of control (solid) and data dependence (dotted) edges. In Figure 3(a), \(S_5\) and \(S_6\) are two program statements in the method \(m()\). In the partial LH-SDG, these two nodes are also denoted by \(S_5\) and \(S_6\) and are connected to the *method entry* node \(m()\) by *control dependence* edges. The partial LH-SDG model shows that the nodes \(S_5\) and \(S_6\) are data dependent on \(S_1\). Now, suppose we add a statement \(S_j\) to the original code segment as shown in Figure 3(b). Due to addition of statement \(S_j\), the statement \(S_6\) is no longer data dependent on \(S_1\) and instead is data dependent on statement \(S_j\). Due to addition of statement \(S_j\), the *data dependence* edge between nodes \(S_1\) and \(S_6\) is deleted, and two new *data dependence* edges are added between the pairs of nodes \(S_1\) and \(S_j\), and \(S_j\) and \(S_6\). Also, a *control dependence* edge from *method entry* node \(m()\) to the node \(S_j\) is introduced.

Deletion of one or more statements can affect the dependencies existing among other program statements. For example, if a statement defining a variable is deleted, then it would affect the dependency structure of the dependent statements and these would save to now have dependency on some other statement. So, in case of deletion of a statement, before deleting a statement it is required to identify and mark all those program statements that are data and control dependent or dependent due to object-relations on the deleted statement. In LH-SDG, before a node(s) corresponding to the deleted statement is deleted, the other nodes in LH-SDG that are data or control dependent or dependent due to object-relations on the deleted node(s) are identified and are marked as *affected*. Then the node corresponding to the deleted statement is deleted. Subsequently, the *in* and *out* edges from the deleted node are also deleted and new dependency edges as required are created.
Figure 3. Effect of Addition of a Statement on Data and Control Dependencies

We give an example of the effect of the deletion of a statement in Figure 4. Figure 4(a) shows a code segment and the corresponding partial LH-SDG model consisting of control (solid) and data dependence (dotted) edges. In the partial LH-SDG model, the node $S_q$ is data dependent on node $S_p$ and the node $S_r$ is data dependent on nodes $S_p$ and $S_q$ in the original code. Suppose the statement $S_q$ is deleted as shown in Figure 4(b). Due to deletion of the statement $S_q$, an additional data dependence edge is introduced between the nodes $S_1$ and $S_r$. Therefore the node $S_q$ and all in and out edges of $S_q$ are deleted from the partial LH-SDG.

3.2 Updation of an LH-SDG model

This Section describes how our constructed LH-SDG model is updated when a change is made to the original program. Existing RTS techniques construct program models for both the original and modified programs [25, 24]. However, construction of an LH-SDG model each time a change is made to a program incurs significant overhead especially for large programs, and should be avoided. To overcome this problem, instead of creating an LH-SDG model each time changes are made, we update the original LH-SDG model denoted by $M$ to reflect the changes made to $P$.

For addition of a statement, we create one or more nodes in LH-SDG model. For addition of a statement, usually a single node is created. But for statements such as method call (also called call-site) or called method, more than one node need to be created in a LH-SDG model. If it is a method call statement, then in addition to a node for calling method, we create nodes one for each actual-in and actual-out parameters.

Figure 4. Effect of Deletion of a Statement on Data and Control Dependencies
In case of deletion of a statement, we delete the corresponding node (or nodes) from LH-SDG model. But, before deleting the node (or nodes), we determine the nodes that are dependent on the deleted node due to dependencies that arise due to either control and data dependencies or object-relations such as association or inheritance and are marked as affected. After marking these dependent nodes as affected, we delete all in and out edges from the deleted node and then the node corresponding to a deleted statement is deleted from LH-SDG model. For each deleted statement, we traverse M to reach the deleted node n and then determine all nodes that are dependent on n.

In addition to updating the changes to LH-SDG model, we also mark the changes that are made to P and are done on LH-SDG model as follows: For each statement added in P’, the corresponding node (or nodes) in Mu are searched and are tagged as changed. And for each deleted statement in P, we determine the nodes that are dependent on the deleted node and are considered as affected due to deletion before updating LH-SDG model. Therefore, these affected nodes are searched in Mu and are tagged as deleted. We denote both the set of tagged nodes as Tagged.

3.3 Determination of Affected Methods

The procedure $SMAnalysis$ as given in Algorithm 1 has been designed to determine the affected methods from state machine model of a changed class. The input to the procedure $SMAnalysis$ is the state machine model of a changed class denoted by $SM$, the changed statement denoted by $s$ and the method to which $s$ belongs denoted by $m$ and produces the affected methods denoted by $AffectedMethods$ as output. Let $E$ be the set of all transitions in a $SM$. First, $SMAnalysis$ determines the variable $v$ whose value is modified in a changed statement $s$ (line 2). Now, if the method $m$ to which $s$ belongs causes a transition $e \in E$ in $SM$ and if the value of $v$ is modified in the destination state of $e$ that is $D_e$, then all transitions from $D_e$ are affected due to change. For this, $SMAnalysis$ determines the methods that cause transitions from $D_e$ (line 6). Then, all methods that cause transitions from $D_e$ are added to $AffectedMethods$ (line 7).

4. Regression Test Selection

Besides selecting test cases based on an analysis of the LH-SDG model, we also select test cases by analyzing UML state machine models. The set of selected regression test cases ($T_{REG}$) can be expressed as:

$$T_{REG} = T_{DEP} \cup T_{SM}$$

(1)

Where, $T_{DEP}$ denotes the test cases selected through control and data dependence analysis and dependencies due to object-relations and $T_{SM}$ denotes the test cases selected through state behavior analysis from state machine models.

4.1 Determination of $T_{DEP}$

Regression test cases, $T_{DEP}$, are determined based on an analysis of the constructed LH-SDG model. To select $T_{DEP}$, we first compute the forward slice on updated marked LH-SDG model. Our forward slicing algorithm is based on two pass graph reachability algorithm proposed by Horwitz et al. [12], where each marked model element that are tagged during Update LH-SDG model step, is taken as the slicing criterion. Our construction of LH-SDG model slice performs a reachability analysis using control and data dependence edges, class
member edges etc. This helps us to identify the set of model elements that are affected due to the modifications that are made to the original program.

<table>
<thead>
<tr>
<th>Algorithm 1: Pseudocode to determine affected methods through state machine model analysis.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> SM, s, m</td>
</tr>
<tr>
<td><strong>Output:</strong> AffectedMethods</td>
</tr>
<tr>
<td>1 procedure SMAnalysis (SM, s, m, AffectedMethods)</td>
</tr>
<tr>
<td>2 Find the variable v whose value is modified in s</td>
</tr>
<tr>
<td>3 AffectedMethods ← NULL</td>
</tr>
<tr>
<td>4 for each transition e ∈ E in SM that are caused by m do</td>
</tr>
<tr>
<td>5 if the value of v is changed at D_e then</td>
</tr>
<tr>
<td>6 Find the methods that cause transitions from D_e</td>
</tr>
<tr>
<td>7 AffectedMethods = AffectedMethods ∪ {all methods that cause transitions from D_e}</td>
</tr>
<tr>
<td>8 end</td>
</tr>
</tbody>
</table>

The pseudocode to select test cases from LH-SDG model is given in Algorithm 2. We have named our algorithm as LH-SDGSelect. LH-SDGSelect takes M_u and the set of tagged nodes denoted by Tagged obtained during update LH-SDG model step as input, and produces the selected set of regression test cases as the output, T_DEP. LH-SDGSelect computes the set of all affected nodes denoted by AffectedNodes on account of data and control dependencies or dependencies arising due to object-relations such as inheritance or association by graph-reachability analysis and the steps are given in lines 2 to 5 in Algorithm 2. After all the affected nodes in LH-SDG have been identified through forward slicing, the test cases that execute these affected nodes are selected for regression testing. This is done by traversing the LH-SDG model and visiting each node in AffectedNodes to determine the test cases that execute these affected nodes.

4.2 Determination of T_SM

In order to select regression test cases based on the state machine model, we analyze the state machine model of each changed class. For each changed statement s in differ, we determine the affected methods from the corresponding state machine model. The pseudocode to select test cases from state machine models of affected classes is presented in Algorithm 3. We have named the RTS algorithm for selecting regression test cases from state machine model as SMSelect. SMSelect takes the file differ as input and produces the selected set of regression test cases as the output and is denoted by T_SM. The working of SMSelect is explained as follows.

First, for each changed statement s in differ, SMSelect invokes the procedure SMAnalysis as given in Algorithm 1, to determine the AffectedMethods for each s (line 4). Then, for each affected method p in AffectedMethods, SMSelect invokes the procedure AddTestCases that takes an affected method p from AffectedMethods and the corresponding state machine model as input and produces a set of affected test cases denoted by T' as output. The procedure AddTestCases first determines the transitions that are caused by each method p ∈ AffectedMethods in SM (line 12) and are added to the set denoted by E' and the test cases executing the transitions in E' are added to the set T'. Then, the test cases in T' for each affected method are selected for regression testing and are added to T_SM (line 7). In the following section, we describe how the test cases are selected by H-RTS with an example.
4.3 An illustrative Example

We explain our RTS approach H-RTS using the example of a Vending Machine (VM). A VM has Coinbox, Dispenser and Coinreturn components. We consider this VM example to illustrate how a change in code affects the state of objects and how it can be analyzed from state machine model.

The Coinbox allows vending only when two coins are received. When a user inserts a coin into VM, insert increments the count of coins, curCoins. After the user has inserted at least two coins, insert sets a flag alVend. When the user makes a selection, vend checks the current value of flag alVend by invoking a method isalvend which returns the value of alVend. If alVend is set, vend adds curCoins to the total number of coins, totCoins, resets the state variables curCoins and alVend and invokes Dispenser to dispenses the user’s selection. In the Dispenser class of Figure 5, the dispense method simulates dispensing of a drink by outputting a message.

At any time, the user may request for return of coins and in that case, return method of Coinbox invokes the retcoins of Coinreturn, and resets curCoins. The retcoins method in the Coinreturn class simulates returning of coins by outputting a message. In addition to these methods, Coinbox has a constructor Coinbox that creates a coin box and a destructor ~Coinbox. Fig. 5 gives the Java code for the Coinbox component of the VM, along with other components to which it interacts. In the Coinbox component of VM, if the variable alVend is modified in the vend method as shown in Fig. 6, then it is clear from code analysis that, any other methods except isalvend, will not be affected due to control and data dependencies and dependencies arising from object-relations. To select regression test cases TSM for the VM example, we consider the state model of Coinbox class, as the changed statement belongs to method vend of Coinbox class.

An object in Coinbox class has four states as shown in Figure 6. After constructor is executed, an object of Coinbox goes to state S1. The invocation of insert method in state S1 causes Coinbox to transit to state S2 and invocation of insert method in state S2 causes Coinbox to transit to state S3. From analysis of state machine model of Coinbox, it can be seen that, alVend is a state variable and the value of alVend becomes 1 in state S1 instead of 0.

---

**Algorithm 2: Pseudocode to select regression test cases by computing LII-SDG model slice.**

<table>
<thead>
<tr>
<th>Input:</th>
<th>Mn, Tagged</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output:</td>
<td>TDEP</td>
</tr>
<tr>
<td>procedure LII-SDGSelect(Mn, Tagged, TDEP)</td>
<td></td>
</tr>
<tr>
<td>for each node x in Tagged do</td>
<td></td>
</tr>
<tr>
<td>Find the nodes that are data or control dependent or dependent due to object-relations such as inheritance and association on x</td>
<td></td>
</tr>
<tr>
<td>AffectedNodes ← NULL</td>
<td></td>
</tr>
<tr>
<td>AffectedNodes = AffectedNodes ∪ {all nodes that are data or control dependent or dependent due to object-relations such as inheritance or association on n}</td>
<td></td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
<tr>
<td>if AffectedNodes ≠ ∅ then</td>
<td></td>
</tr>
<tr>
<td>for each node n ∈ AffectedNodes do</td>
<td></td>
</tr>
<tr>
<td>Add all test cases that execute n to TDEP</td>
<td></td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
</tbody>
</table>

---
Therefore, all transitions that start from state \( S_1 \) are affected. In the state machine model of \textit{Coinbox} class, the transitions \( t_1 \) and \( t_2 \) start from state \( S_1 \) as shown in Figure 6. Therefore, both \( t_1 \) and \( t_2 \) are affected and for that the methods that cause the transitions \( t_1 \) and \( t_2 \) are determined. From state model of \textit{Coinbox} class, it can be seen that transitions \( t_1 \) and \( t_2 \) are caused by the invocation of methods \textit{insert} and \textit{return} respectively. When the \textit{insert} and \textit{return} methods are invoked at state \( S_1 \), the state variable \textit{alVend} should be 0. But due to modification of the value of variable \textit{alVend}, it becomes 1 in state \( S_1 \).

\begin{algorithm}
\caption{Pseudocode to select regression test cases through state machine model analysis.}
\begin{algorithmic}
\Input \text{differ} \\
\Output \text{TSM} \\
\Procedure{SMSelect}{differ, TSM}
\State TSM \leftarrow NULL
\For {each statement \textit{s} \in differ}
\State \text{SMAnalysis} (SM, s, m, \text{AffectedMethods})
\For {each method \textit{p} \in \text{AffectedMethods}}
\State AddTestCases (SM, \textit{p}, T')
\EndFor
\State TSM = TSM \cup T'
\EndFor
\EndProcedure
\Procedure{AddTestCases}{SM, \textit{p}, T'}
\State T' \leftarrow NULL
\For {each transition \textit{e}' \in E'}
\State Add all test cases that execute \textit{e}' to \textit{T}'
\EndFor
\EndProcedure
\end{algorithmic}
\end{algorithm}

Therefore, the methods \textit{insert} and \textit{return} are affected. Then, the procedure \textit{AddTestCases} determines all transitions that are caused by the methods \textit{insert} and \textit{return} and the test cases executing these transitions are selected for regression testing. From state transition diagram it is cleared that, the scenario \textit{insert}, \textit{insert}, \textit{return} is possible and is selected for regression testing by state machine model analysis but is not selected by only analysis of LH-SDG model.

5. Experimental Studies

We have implemented a prototype tool based on our proposed approach for RTS. We have named our prototype tool as H-RETEST (Hybrid Regression TEST case selector).

5.1 H-RETEST: A Prototype Implementation of H-RTS

H-RETEST has been developed using the programming language Java on a Microsoft Windows XP environment. The code size of H-RETEST is approximately 15 KLOC, excluding the external packages that are used in implementation of H-RTS technique. The user interface of H-RETEST is developed using Java Swing.

In the following, we describe the various open source software packages used to implement H-RTS.
5.1.1 Open source software packages used

We have developed the tool H-ReTEST using the following open source software packages: Eclipse [4], ANTLR [1] and Graphviz [5]. We have used eclipse as an IDE and ANTLR as the parser generator, and have adapted ANTLR grammar file for Java language [30]. The ANTLR is integrated into eclipse as a plug-in. The procedure to install ANTLR plug-in in Eclipse is available in [2]. To graphically visualize the LH-SDG model constructed by H-ReTEST, we have used Graphviz.

5.2 Experiments

In this section, we discuss the specific experimentation carried out by us using H-ReTEST to measure the effectiveness of our approach. We have used the following programs namely, Automated Teller Machine (ATM), Climate Controller (CIC), Vending Machine (VM), Subway Turnstile (SWT), Power Window Controller (PWC), Elevator Controller (EC), and Cruise Controller (CrC) in our experimentation.

The size of the considered programs range from 471 to 943 LOC as given in Table 2. Each of the considered programs had on an average of 27 test cases. For each program, we created several modified versions. We have considered the different types of modifications that are made in each version of a program from Ren et al. [22]. The modifications made to each version of a program are arbitrary.

We tested each modified version of a program by running the original test cases on each modified version of a program to note the number of test cases failed after modification. Then, each time the test cases were selected using H-RTS and also from LH-SDG analysis. We repeated the experiment for each modified version of each considered program in order to remove any bias introduced in the results due to a specific type of change. To measure the effectiveness of our RTS technique, we have calculated the average percentage of fault-revealing test cases selected by H-RTS and by LH-SDG model analysis.
5.3 Results

In this section, we describe the results obtained from experimental studies carried out by us to determine the effectiveness of our RTS technique. Table 1 and 2 summarize our experimental results. Table 1 summarizes the percentage of test cases selected by our approach and LH-SDG model based approach. In Table 1, the example programs used in our experimental studies is given in column 1 and column 2 shows the lines of code (LOC) for each of our example programs. In column 3, we list the total number of test cases in the initial test suite and the percentage of test cases selected while executing the entire test suite on the modified programs by H-RTS and by LH-SDG based approach is reported in column 4 and column 5 respectively. The percentage increase in the regression test suite size is given in column 6. H-RTS on an average selects 38.21 % more than the only LH-SDG model-based approach. This increase may be due to the fact that, our approach selects test cases based on an analysis of state models in addition to selecting from code analysis.

Table 2 summarizes the average percentage of fault-revealing test cases selected by both approaches. In Table 2, the test cases failed is given in column 2. The average percentage of fault-revealing tests selected by H-RTS and LH-SDG based approach is given in columns 3 and 4 respectively.

The results show that H-RTS selects all the fault-revealing test cases from the selected test cases and the percentage of fault-revealing test cases selected by H-RTS is on an average of 27.89 % higher than a LH-SDG model-based analysis.

6. Related Work

A large number of research results on regression test selection of object-oriented software have been reported in literature [25, 24, 29, 11, 15, 13, 27, 8, 21, 19, 13]. We have discussed only the RTS techniques for object-oriented programs that are code-based [25, 24, 11] and model-based [8, 27, 21] and how these techniques differ from our proposed technique.

Rothermel and Harrold proposed an RTS approach for object-oriented software that uses dependence graph representations for modified classes, derived classes and application programs [25]. For RTS of modified application program, they used inter-procedural program dependence graph (IPDG) as an intermediate representation for both original and modified programs. For RTS of modified and derived classes, they used CIDG of both the original and modified programs. The test coverage information is used to associate the predicate and statement nodes of the CIDG model with each test case. For a modified class, two CIDGs $G$ and $G'$ are constructed corresponding to the original and modified classes. A representative
driver node (RDN) is used that serves as a root of the graph. The test coverage information is used to associate the predicate and statement nodes of the CIDG models with each test case. Then, a traversal (depth-first) of $G$ and $G'$ is performed starting from the root node of both the graphs. If a pair of nodes is found in $G$ and $G'$ such that they are not identical, then the test coverage information associated with that node is used to select regression test cases.

### Table 1. Summary of Regression Test Selection Results

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Number of LOC</th>
<th>Number of test cases</th>
<th>% of test cases selected by H-RTS</th>
<th>% of test cases selected by LH-SDG analysis</th>
<th>% Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATM</td>
<td>623</td>
<td>25</td>
<td>62</td>
<td>45</td>
<td>37.77</td>
</tr>
<tr>
<td>ClC</td>
<td>521</td>
<td>35</td>
<td>47</td>
<td>30</td>
<td>56.66</td>
</tr>
<tr>
<td>VM</td>
<td>471</td>
<td>25</td>
<td>49</td>
<td>36</td>
<td>36.11</td>
</tr>
<tr>
<td>SWT</td>
<td>634</td>
<td>18</td>
<td>52</td>
<td>41</td>
<td>26.82</td>
</tr>
<tr>
<td>PWC</td>
<td>762</td>
<td>30</td>
<td>71</td>
<td>52</td>
<td>36.53</td>
</tr>
<tr>
<td>EC</td>
<td>943</td>
<td>18</td>
<td>74</td>
<td>56</td>
<td>32.14</td>
</tr>
<tr>
<td>CrC</td>
<td>64</td>
<td>42</td>
<td>58</td>
<td>41</td>
<td>41.46</td>
</tr>
</tbody>
</table>

An RTS technique for C++ software was proposed by Rothermel et al. [24]. Their technique can be applied to both modified and derived classes as well as to the application programs. Their approach for test selection is based on control flow graph representations of programs. For RTS of C++ software, they used inter-procedural control flow graph (ICFG) for application programs and class control flow graph (CCFG) for modified and derived classes as an intermediate representation. As in [25], they identify the differences between the two graphs by comparing corresponding nodes during graph-traversal. A depth-first traversal is performed starting from the entry nodes of both the graphs until a pair of nodes is found whose associated statements are not lexicographically equivalent. If a pair of nodes $n$ and $n'$ in $G$ and $G'$ respectively, is found such that the statements associated with $n$ and $n'$ are not identical, then the edges that lead to the non-identical nodes are identified as affected edges. Test cases that execute the set of identified affected edges are selected for regression testing.

### Table 2. Summary of Quality Results

<table>
<thead>
<tr>
<th>Program Name</th>
<th>% of test cases failed</th>
<th>% of fault-revealing tests selected by H-RTS</th>
<th>% of fault-revealing tests selected from LH-SDG model analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATM</td>
<td>24</td>
<td>100</td>
<td>81</td>
</tr>
<tr>
<td>ClC</td>
<td>30</td>
<td>100</td>
<td>78</td>
</tr>
<tr>
<td>VM</td>
<td>27</td>
<td>100</td>
<td>76</td>
</tr>
<tr>
<td>SWT</td>
<td>22</td>
<td>100</td>
<td>82</td>
</tr>
<tr>
<td>PWC</td>
<td>21</td>
<td>100</td>
<td>75</td>
</tr>
<tr>
<td>EC</td>
<td>24</td>
<td>100</td>
<td>81</td>
</tr>
<tr>
<td>CrC</td>
<td>28</td>
<td>100</td>
<td>75</td>
</tr>
</tbody>
</table>

Harrold et al. extended their previous work on RTS [24] and proposed a safe RTS technique for JAVA software [11]. Their technique of regression test selection consists of three steps: constructing intermediate representations for both original and modified programs, analyzing the graphs and determining the set of dangerous edges, and test case selection. Their technique uses a Java Interclass Graph (JIG) as an intermediate representation of both the original and modified programs. JIG represents all Java language features such as inheritance, polymorphism, exception handling etc. They use the same algorithm as described in [24] to select regression test cases.
Farooq *et al.* have presented a UML state machine based regression test selection strategy that uses information from UML 2.1 behavioral state machine and class diagrams for regression test selection [27]. They provide change definitions for UML 2.1 state machines and class diagrams. The modifications made to one model may also affect other design models. Their approach uses information from the modified class and state machine diagrams to find out the directly and indirectly affected elements of the model.

Briand *et al.* have proposed a regression test selection methodology based on object-oriented designs [8]. Their approach assumes the existence of traceability between the design models, the code and the test cases. When a change is made to a design model, the traceability between the design and test cases helps in associating the changes in a design model to the test cases. Then, the associated test cases are run to exercise the affected parts in design. Their approach uses UML use case, class and sequence diagrams for analysis. Also, their technique assumes that for each use case, there is a unique sequence diagram that models all possible object interactions.

Naslavsky *et al.* [21] presented a model-based RTS technique that uses UML class and sequence diagrams for test selection. They transformed sequence diagrams of both the original and modified versions of a program into model-based control flow graphs. The traceability between test cases and the sequence diagrams is used to determine the elements of control flow graphs that are executed by each test case. Finally, the control flow graphs of both original and modified versions are analyzed and the test cases are selected using traceability information.

The reported code-based RTS approaches for object-oriented programs [11, 25, 24] do not consider the effect of state behavior of objects during RTS. Our RTS approach uses both source code and state machine models and hence considers state behavior of objects along with dependencies information. Also, existing code-based RTS approaches [11, 25, 24] construct graph models for both the original and modified programs, whereas, our approach constructs graph model for the original program only and the constructed model is updated each time changes are made to the system. Existing code-based approaches [11, 25, 24] store the test coverage information in terms of a coverage matrix and is created each time new test cases are added or when obsolete test cases are deleted from the test suite $T$. But in our approach the test coverage information is updated only once in a regression testing cycle. In the existing model-based approaches [27, 8, 21], the original and modified models are compared to select regression test cases. However, minor code changes may not lead to model changes. Therefore, pure model-based approaches are unsafe when minor code changes that do not impact the model. But, in our approach, we consider the changes that are made to the source code and we analyze both the source code and the state machine models of changed classes to select regression test cases.

7. Conclusion

We have presented an approach for regression test selection of object-oriented programs that selects test cases by analyzing both source code and UML state machine models. Our approach considers the effect of state behavior of objects during regression test selection from UML state machine models. We have applied the proposed RTS technique to small example programs to prove the applicability of our approach. The results of our study show the effectiveness in selecting more fault-revealing test cases from the original test suite. In our empirical studies, we observe an average increase of 27.89% selection of fault-revealing test cases in H-RTS as compared to only LH-SDG model based analysis.
References

Authors

Chhabi Rani Panigrahi is currently a PhD Student in the Department of Computer Science & Engineering at Indian Institute of Technology (IIT), Kharagpur, India. Her research interest include Regression testing of object-oriented programs. She can be reached at chhabi@cse.iitkgp.ernet.in.

Rajib Mall is currently a Professor in the Department of Computer Science & Engineering, Indian Institute of Technology (IIT), Kharagpur, India. He is a Senior Member in IEEE. He has published over 100 refereed research papers and has authored two books. His current research interests include analysis and testing of object-oriented programs. He can be reached at rajib@cse.iitkgp.ernet.in.