International Scholarly Research Network
ISRN Software Engineering
Volume 2012, Article ID 561502, 9 pages
doi:10.5402/2012/561502

Research Article

State-Model-Based Regression Test Reduction for Component-Based Software

Tamal Sen and Rajib Mall

Department of Computer Science and Engineering, Indian Institute of Technology, Kharagpur 721302, India

Correspondence should be addressed to Rajib Mall, rajib@cse.iitkgp.ernet.in

Received 25 July 2012; Accepted 16 August 2012

Academic Editors: D. Tang and R. J. Walker

Copyright © 2012 T. Sen and R. Mall. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

We present a novel regression test selection approach based on analysis of state and dependence models of components. Our technique targets to select a smaller regression test suite compared to the pure dependence-based RTS approaches while maintaining the fault revealing effectiveness. In our approach, after a modification, control and data dependencies are analyzed to identify the potentially affected statements. Subsequently, the state model of the component is analyzed to compute a precise publishable change information to support efficient regression test selection by the application developers.

1. Introduction

A component is an implementation of a cohesive group of reusable services in a single executable unit. Components are developed independently, available off the shelf and integrated into a component-based application by the developers. Component-based development has found rapid acceptance among software developers due to its promise of helping lower the overall development costs and at the same time speeding up the development process.

Components can be written using a variety of programming languages and may be distributed across different platforms [1]. To protect the IPR (Intellectual Property Rights) of the developers, usually source code is not included in the component licence. Component services are accessed through supported interfaces. Components usually support two different types of interfaces: provided and required. Provided interfaces specify the services offered by the component. A service offered by a component is also variously referred to as *provided operation*, *published operation* or *published method*. On the other hand, concrete implementation of the required interfaces is left to the users of the component. The interfaces are generally defined using an interface definition language (IDL). The application developers make use of interface definitions of the components to develop an application. Though component-based development offers many advantages, unavailability of source code makes testing activities such as coverage analysis and regression test selection difficult to carry out.

Regression test selection (RTS) is the process of selecting a subset of initial system test cases for regression testing. Regression testing is particularly important in a component-based environment, since the components evolve independently and are upgraded frequently. In this context, analyzing change impact as well as selecting a safe subset of the system test cases for regression testing in an efficient and precise manner is important.

For component-based software, traditional RTS techniques are difficult to use because the application developers do not have the source code for analyzing the change impact, neither can they obtain coverage data of the test suite through code instrumentation. It may be unrealistic to expect component vendors to provide those information due to obvious reasons. To facilitate regression testing, it becomes necessary for component vendors to provide built-in-test interfaces [2–4] or to publish abstracted change information such as affected method signature, pre/postconditions, among others after each modification.

The simplest form of change information can be a collection of affected methods which could be published after every modification to a component. Techniques for

identifying affected methods can be as simple as choosing those methods which have been modified or those that directly or indirectly call a modified method [5]. But errors can also show up in the unmodified parts that are control or data dependent on the actually modified parts. Analysis of control and data dependence relationships therefore is necessary for detecting faults that get induced due to code changes elsewhere [6]. Hence, all affected *published methods* need to be published in the change information in order to make regression testing safe.

In contrary to what is implicitly assumed by many existing techniques [5, 7], invoking an affected method does not ensure that the affected statements inside that method will be executed. A pure dependence-based technique such as [7] selects test cases which invoke one or more component methods that have been found affected by dependence analysis. As a result, redundant test cases might get selected which invoke affected methods but do not actually execute affected statements. In this context, we propose an RTS technique in which, after identifying the affected statements by dependence analysis, the state model of the component is analyzed to identify the transitions that may cause execution of the affected statements. The set of affected transitions is published and only affected-transition-traversing test cases are chosen for regression testing, instead of choosing all of them which invoke affected methods. Since a transition may not cause execution of every statement of a *published method*, it is possible that some test cases invoke an affected published method but do not trigger an affected transition. Consequently, we end up in selecting a smaller regression test suite. We have named our technique as State model analysis based Regression Test Selector for component-based system (S-RTS).

This paper is organized as follows: Section 2 describes some background concepts such as the dependence and state models of a program. Section 3 discusses model-code translation using an example. Section 4 presents the methodology of our approach. In Section 5, we report experimental results. Related literature is reviewed in Section 6. In Section 7, we conclude the paper.

2. Background Concepts

We discuss a few background concepts that have been used to develop our regression test selection approach.

2.1. Java System Dependence Graph. As defined by Podgurski et al. [6],

> *Program dependences are relationships, holding between program statements, that can be determined from a program's text and used to predict aspects of the program's execution behavior.*

A dependence model of a program is an abstract representation of various dependencies existing in a program. Dependence models have widely been used in software engineering activities such as program testing, debugging, and test optimization. A dependence model for monolithic single procedure program, called *Program Dependence Graph* (PDG), was proposed by K. J. Ottenstein and L. O. Ottenstein [8]. Later, a notion called *System Dependence Graph* (SDG) was introduced by Horwitz [9] to capture interprocedural dependence for procedural programs. Later, SDG has been extended by many researchers to incorporate object-oriented features considering languages like C++ and Java [10–12].

The *Java System Dependence Graph* (JSDG) proposed by Walkinshaw et al. [12] is briefly discussed to enhance the readability of the paper.

(i) A *statement vertex* represents a single statement in the source code. If the statement includes a method call, a *call-site* node is created with *formal-in* and *formal-out vertices*.

(ii) For each method in a class, a *Method Dependence Graph* (MDG) is created. MDG captures intramethod control and data dependencies similar to a PDG [8]. MDG includes a method entry vertex to represent the entry point to the method and also includes *actual-in*/*actual-out* vertices to model parameter passing. To represent a method call with parameters, the *call-site vertex* is connected to the method entry vertex with *call dependence edge*, actual and formal vertices are also connected accordingly. Parameter vertices are also created for instance variables which are referenced, or modified inside the method.

(iii) Each class is represented by a *Class Dependence Graph* (ClDG). For every class, the ClDG consists of a *class entry vertex* representing the entry point to the class. The *method entry vertex* of the MDG of a method in a class is connected to the class entry vertex via *class member edge*. *Class member edges* are tagged according to the visibility of the method (public, private, protected or default). The *class entry vertex* is also connected to vertices representing its instance variables via *data member edges*. To represent inheritance, the *class entry vertex* of the derived class is connected to the *class entry vertex* of its parent via a *class dependence edge*.

(iv) In JSDG, different instances of a class are represented separately. A *statement vertex* which references an object is expanded into a tree depending on how the object is referenced.

  (a) If an object is passed as a parameter to a method, the parameter is connected to the nodes representing possible subtypes of the parameter object. The node representing each object type is further connected to the nodes representing the objects data members via a *control dependence edge*. If a data member is again an object, it is further expanded into a subtree.

  (b) For a polymorphic method call on an object, the corresponding *call site vertex* is connected to a vertex representing the object defining the method and further connected to vertices

representing possible polymorphic bindings of the method.

(v) To represent an interface, *Interface Dependence Graph* (InDG) is created. InDG consists of an *interface entry vertex* which is connected to its abstract methods via *abstract member edges*. Abstract methods are represented by a *method entry node* with *parameter-in/parameter-out vertices*. The *method entry vertex* of an abstract method is connected to the *method entry vertex* of its implementing method via *implement abstract method edge*. *Interface entry vertex* of an interface and the *class entry vertices* of its implementing classes are connected via *implements edge*.

(vi) Representation of an abstract class includes a *class entry vertex* which connects entry vertices of concrete methods via normal *class member edge* and connects entry vertices of abstract methods via *abstract member edge*.

(vii) A package is represented by a *Package Dependence Graph* (PaDG). PaDG consists of a *package entry vertex* connecting representations of its classes and interfaces via *package member edge*.

2.2. State Model. The state model of a system shows the possible states that the system can assume and the transitions among the states. Usually state models are constructed by designers during design phase. However, researchers have proposed several automated techniques for reverse engineering the state model from source code [13–15]. The state model of a program is defined as follows.

Definition 1 (state model). A state model is quadruple: $\langle Q, S, V, T\rangle$.

(i) Q: finite set of states. Each state $q \in Q$ is labelled with an unique identifier denoted as $q \cdot id$.

(ii) S: finite set of events.

(iii) V: finite set of variables.

(iv) T: finite set of transitions. A transition t is a quintuple: $\langle q_{\text{pre}}, st, guard, action, q_{\text{post}}\rangle$.

(a) $q_{\text{pre}}, q_{\text{post}} \in Q$: source and target state of t.

(b) $st \in S$: event that triggers t.

(c) *guard*: predicate on parameters of st.

(d) *action*: finite set of manipulations on V.

Consider the state model of `PrintHello` component depicted in Figure 1(a). The `PrintHello` component has two states called *Small* and *Capital*, and it has a *published method* called *print*. On invocation of *print* in different states, the word "Hello" is printed in different cases. When *print* is invoked in state q_1, "Hello" is printed in small case on the other hand, "Hello" is printed in upper case when *print* is invoked in state q_2. If *print* is invoked in state q_1 and the value of parameter p is *zero*, a transition occurs to state q_2 but the state is not changed if *print* is invoked with a nonzero parameter. A transition occurs from q_2 to q_1 only if *print* is invoked exactly twice with nonzero parameter. In q_2, a variable named *a* keeps track of the number of invocations of *print* with nonzero parameter and it is made to *zero* when a transition occurs to state q_2.

3. Code Generation from State Model

In this work, we assume that components are developed using model-driven development (MDD) paradigm in which the state model is designed first and later, and the model is translated into Java code. We further assume that during various maintenance activities the state model is altered and then the code is regenerated from the updated model, and/or the source code is changed without affecting the state model. Several tools at present support model-to-code translation. For example, Simulink State Flow [16] supports state chart to C/C++ code generation, and Rhapsody [17] supports C++/Java code generation from UML state charts. Techniques for UML state chart to Java translation also have been reported by Mehlitz [18] and Niaz and Tanaka [19]. These tools carry out model-to-code translation by using a set of rules and therefore a strong correspondence is maintained between model elements (e.g., state, transition) and the generated code.

In this section, we consider a specific model-code translation scheme based on *state design pattern* [20] and we discuss how code is generated from a state model. Subsequently, we define a special case of model-code correspondence called Transition Statement Correspondence (TSC).

At first, the *main component class* is created with the name same as the component. Later, during translating transitions into code, *published methods* are created inside it.

States in the model are translated into *concrete classes* by implementing the *State* interface. The *State* interface includes an *abstract handler method* for each *published method* in the *main component class*. These *abstract handler methods* are implemented in the *concrete state classes*. A *published method* inside *main component class* internally invokes its *handler method* implemented in the *concrete state class* of the present state.

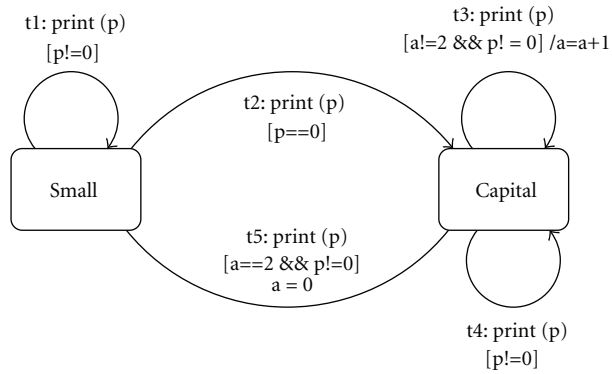
As mentioned in Definition 1, a transition has five attributes: *source state*, *target state*, an *event*, a *guard condition,* and an *action*. While generating code from a transition, various attributes are translated using the following rules.

(i) To represent the *event*, a *published method* is created in the *main component class*, an *abstract handler method* is created inside the *State* interface, and the *handler method* is implemented in *concrete state classes*.

(ii) To represent the *guard*, an `if-then` block is introduced inside the *handler* of the *published method* implemented in the *concrete state class* of *source state*.

(iii) The *action* is translated into statements inside the `if-then` block that represents the *guards*.

(iv) The *target state* of a transition is specified by calling `setState()` method of the main component class.



(a) State model of `PrintHello` component

```
1 interface State {
2     abstract void printHandler (PrintHello ph, int p);
3 }
4 class Small implements State {
5     public String id="small";
6     public void printHandler (PrintHello ph, int p) {
7          System. out. print ("hello");
8          if(p !=0){}
9          if(p ==0) // change state only if p is 0
10              ph. setState (new Capital ());
11 } }
12 class Capital implements State {
13     public String id="capital";
14     private int a=0;
15     public void printHandler (PrintHello ph, int p) {
16          System. out. print ("HELLO");
17          if(p !=0){
18               a=a+1;
19               if(a ==2){
20                    a =0;
21                    ph. setState (new Small ());
22 }}}}
23 public class PrintHello {
24     private State myState ;
25     public PrintHello () {
26       setState (new StateA ());
27     }
28     void setState (State newState) {
29        this. myState = newState ;
30     }
31     public void print (int p) {
32        this. myState. printHandler (this, p);
33     }
34     // state reporting method
35     public String getCurrentState (){
36          if(myState instanceof Small)
37            return ((Small ) myState). id;
38          if(myState instanceof Capital)
39            return ((Capital ) myState). id;
40     }}
```

(b) Java code generated from the state model

Figure 1: State model of `PrintHello` component and the generated code.

The *main component class* keeps a reference of the current state. We assume that a state reporting method named `getCurrentState()` is created inside the main component class, which returns the symbolic identifier of the current component state.

Figure 1(b) shows the Java code generated from the state model shown in Figure 1(a). As can be seen there, the class `PrintHello` is created as *main component class*. The states in the model (*Small* and *Capital*) are represented by the classes called `Small` and `Capital`. Transitions are translated by translating their attributes one by one. For $t1$, the *published method* `print()` is created inside the *main component class* `PrintHello`. Inside the *State* interface, an *abstract handler* called `printHandler()` is created for the *published method print*. Since the *source state* of the transition is `Small`, the code is generated inside the *printHandler* implementation of `Small` class. An `if-then` block is created in line 8 to represent the *guard*, and since there is no *action* specified in the transition, no statement is created inside the if-then block. In the same way, for the transition $t2$, statements in lines 9 and 10 are generated. The state change caused by $t2$ is represented by a call to the `setState()` method of the *main component class*. Other transitions are also translated in the same way based on the above-mentioned rules.

When a transition t occurs in a component under execution, the set of statements generated from t is executed. In this context, we define Transition Statement Correspondence or TSC assuming that $\mathcal{S}$ is the set of statements in the generated code.

Definition 2 (TSC). For a given state model and the generated code from it, TSC is defined as a mapping from the set of transitions T to set of all subsets of $\mathcal{S}$, that is, $\text{TSC} : T \rightarrow 2^{\mathcal{S}}$, such that, for all $t \in T$, the statements in $\text{TSC}(t)$ are executed when the transition t takes place.

For instance, when the transition $t2$ occurs, statements $\{9, 10\}$ are executed. Therefore, $\text{TSC}(t2) = \{9, 10\}$.

4. S-RTS: Our Proposed Approach

In this section, we present the methodology of our S-RTS approach. We first present a brief overview and subsequently provide a more detailed discussion of the different steps.

Overview of S-RTS. A component undergoes a number of corrective, perfective, or adaptive changes throughout its life cycle. After the planned changes are made, a Component Dependence Graph (CDG) is constructed from the new version of the component. The dependence graphs as well as the state model of the component are analyzed by the developers of the component in order to compute a precise change information. Finally, the change information is published during release of the modified version of the component.

On the application developers' end, transition coverage is recorded for each test case during initial testing. When a new version of a component is integrated, regression test cases are selected based on:

(i) change information published by the vendor of the upgraded component,

(ii) transition coverage of the initial test cases.

Figure 2 shows various activities carried out at two different ends. A rectangular box represents an activity, and an ellipse represents input/output artifact of an activity. In the following subsections we elaborate the activities carried out in different stages of S-RTS.

4.1. Dependence Model Construction. A component in isolation is an incomplete system. Assuming the component is developed in Java, a driver class is introduced in order to make it a working software. The *driver* class consists of a method called `frame` [10] which simulates invocations of *published methods* of the component in all possible ways. Inside a `frame`, each *published method* is invoked within an infinite loop. In this setup, component developers construct a Component Dependence Graph (CDG) by using JSDG construction technique [12] discussed in Section 2.1. The node representing the entry point to `frame` is denoted as *component entry node*. Figure 3 shows the *driver* class created for the `PrintHello` component that is shown in Figure 1.

4.2. Change Impact Analysis. After modifications are made, component developers identify the statements which are either directly changed or affected indirectly due to the modifications. At first, using a similar comparator algorithm as in [21], the CDG for the modified component (CDG_n) is compared with the CDG of the previous version (CDG_o). The algorithm recursively finds matching nodes between CDG_o and CDG_n starting from their *component entry nodes*. Whenever an unmatched node is found in CDG_n, the algorithm reports all nodes as modified which belongs to the subgraph rooted at the unmatched node. Later, a forward slice is performed to determine the set of indirectly affected model elements. Each modified nodes reported by the model comparator, and the variables defined in those modified nodes are used as the slicing criteria. Each statement that corresponds to directly/indirectly affected nodes in CDG_n is reported as affected statement. The set of all affected statements in the modified program is denoted by S_a.

Subsequently, component developers identify the affected transitions based on the Transition Statement Correspondence (TSC). A transition which may cause execution of affected statements (i.e., statements in S_a) may lead to unexpected results. So, for a transition t, if $\text{TSC}(t)$ includes any of the statements in S_a, it is marked as an affected transition.

4.3. Publishing Change Information. The set of affected transitions is made available to the application developer as change information (denoted by M_{rts}) during release of the upgraded component. A transition $t \in M_{\text{rts}}$ is represented as a vector $\langle t \cdot q_{\text{pre}} \cdot id, s, g, t \cdot q_{\text{post}} \cdot id \rangle$ where s is the *event*, g

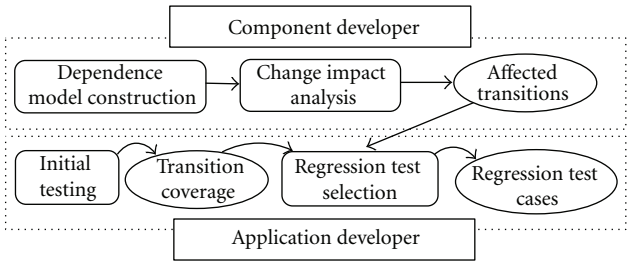


Figure 2: A schematic of our S-RTS methodology.

```
1  public class PHDriver {
2      public void frame (){
3          // initialize PrintHello
4          PrintHello hc = new PrintHello ();
5          while (true){
6               // call published methods
7               hc. print ();
8          }
9      }
10 }
```

Figure 3: *Driver class* for `PrintHello` component shown in Figure 1.

is the *guard condition*, and $t \cdot q_{\text{pre}} \cdot id$ and $t \cdot q_{\text{post}} \cdot id$ are the identifiers of *source* and *target states* of transition t.

4.4. Application Developer Activities. In our approach change information is provided to the application developer with each modified version of the component. Regression test cases are selected based on the change information along with the transition coverage of the initial test cases.

4.4.1. Obtaining Transition Coverage. Transition coverage is obtained during execution of the initial test suite. For each method invoked by a test case, the following information is recorded in order to determine the transition that might occur by the method call.

(i) State identifiers of the component states before (*preId*) and after (*postId*) the method invocation. The `getCurrentState()` method of the component is called before and after the method invocation to record *preId* and *postId*.

(ii) Signature of the method (s).

(iii) Vector of parameter values of the method (sv).

A $\langle preId, s, sv, postId \rangle$ vector stands for a transition having source state id preId, target state id *postId*, event s, and having a guard condition that is satisfiable by sv. The set of $\langle preId, s, sv, postId \rangle$ vectors obtained during execution of a test case t, therefore, corresponds to the transition coverage for t. Transition coverage of a test case t is denoted by the term $coverage_m(t)$.

4.4.2. Regression Test Selection. In this section, we present an algorithm to select regression test cases based on the transition coverage and the change information.

The RTS algorithm (Algorithm 1) scans the transition coverage ($coverage_m$) for every test case t in the initial test suite (T_{init}) and identifies the test cases which may cause affected transitions to occur. A test case t may cause an affected transition to occur iff:

(i) $\exists(preId1, s1, sv, postId1) \in coverage_m(t)$,

(ii) $\exists(preId2, s2, g, postId2) \in M_{\text{rts}}$,

(iii) $(preId1, s1, sv, postId1)$ and $(preId2, s2, g, postId2)$ are equivalent.

$(preId1, s1, sv, postId1)$ and $(preId2, s2, g, postId2)$ are said to be equivalent if $preId1 = preId2$, $postId1 = postId2$, $s_1 = s_2$, and g_2 is satisfiable by sv. Satisfiability is checked using a subroutine called *satisfy* which takes two arguments: a condition over a set of variables and a vector of values of those variables. *satisfy* returns *true* only if its first argument is satisfiable by the variable values specified in the second argument.

The RTS algorithm is presented in Algorithm 1. Inputs to the algorithm are the initial test suite (T_{init}), transition coverage of each test case in initial test suite ($coverage_m(t_i)$) *for each* $t_i \in T_{\text{init}}$ and the set of affected transitions (M_{rts}) which is obtained as change information. The output of the algorithm is the set of regression test cases, denoted by T_{rts}.

In line 1 of Algorithm 1, T_{rts} is initialized as an empty set. The statements in lines 2–18 are iterated for each test case t_i in T_{init} and t_i is included into T_{rts} if required. Inside the loop, at first, a flag called *selected* is initialized to *false*. After that, each element of $coverage_m(t_i)$ is checked whether it is equivalent with an element in M_{rts}. If an equivalent pair is found, the flag *selected* is made *true* in line 7 to indicate that t_i covers an affected transition in that case, control jumps out from the two nested loops and in line 16 the test case t_i is included into T_{rts}. If no equivalent pair is found among $coverage_m(t_i)$ and M_{rts}, the statement in line 7 is never reached; consequently, *selected* is never made *true* and t_i is never included into T_{rts} in line 16.

At termination T_{rts} contains the set of regression test cases.

5. Experimental Studies

We have developed a prototype tool in Java to implement our S-RTS methodology. The prototype tool was used to carry out RTS on several component-based systems developed in Java. We considered five applications: Television Remote Simulator (TRS), Robocop Simulator (RS), Elevator Control System (ECS), Vehicle Controller (VC), and ATM system simulator (ATM). The study subjects had been characterized based on the number of classes (nCLS) in the program and number of test cases in the initial suit (nITS).

TRS is a small program consisting of a television remote controller component and a main component which uses services provided by the remote controller component. The remote controller component provides services like

Input: T_{init}: Initial test suite,
$coverage_m(t)$ for all $t \in T_{\text{init}}$,
M_{rts}: set of affected transitions
Output: T_{rts}
1: $T_{\text{rts}} \leftarrow \phi$
2: **for each** $t \in T_{\text{init}}$ **do**
3: $selected \leftarrow$ **false**
4: **for each** $(preId1,\ s_1,\ sv,\ postId1) \in coverage_m(t)$ **do**
5: **for each** $(preId2,\ s_2,\ g,\ postId2) \in M_{\text{rts}}$ **do**
6: **if** $preId1 = preId2$ **and** $s_1 = s_2$ **and** $satisfy(g, sv)$ **and** $postId1 = postId2$ **then**
7: $selected \leftarrow$ **true**
8: **break**
9: **end if**
10: **end for**
11: **if** $selected =$ **true then**
12: **break**
13: **end if**
14: **end for**
15: **if** $selected =$ **true then**
16: $T_{\text{rts}} \leftarrow T_{\text{rts}} \cup t$
17: **end if**
18: **end for**
19: **return** T_{rts}

Algorithm 1: Select regression test cases.

Table 1: Summary of experimental results.

Study subject			nTCS		Percent of reduction		Percent of detected faults	
Name	nCLS	nITS	D-RTS	S-RTS	D-RTS	S-RTS	D-RTS	S-RTS
TRS	2	13	10	7	23.1	46.2	100	100
RS	4	11	10	8	9.1	27.3	100	100
EC	6	12	6	5	50	58.3	100	100
VC	8	14	6	4	46.4	71.4	100	100
ATM	12	20	20	18	0	10	100	100
Average percent of reduction: S-RTS: 42.64, D-RTS: 25.72								

power on/off, volume control, and channel change. Robocop Simulator is a computer game that simulates a scenario where a thief can move in a given map and a robotic police is responsible to catch the thief. RS consists of controller components for simulating a thief and a robotic police. There is another component which simulates a scenario map. ECS implements the basic functionalities of an elevator system. It includes components for door controller, car controller, door sensor, floor sensor, and a central elevator controller which integrate other components and provide user level functionalities such as calling the elevator car and moving to a specified floor. Vehicle Controller system consists of several controller components such as wiper controller, ac controller, and headlight controller. It also includes sensor components like daylight sensor, rain sensor, and temperature sensor. A main component integrates the other component to simulate a vehicle controller system. ATM simulator is made out of several cohesive components such as banking component, physical ATM component, a transaction component, and a GUI component. The GUI component integrates other components and also provides an interface to interact with the ATM system.

At first we injected faults into the study programs. The types of injected faults include changing arithmetic operators and changing the definition and use of variables. After that, we performed RTS to find the number of test cases selected (nTCS) as well as the number of faults detected. The metrics based on which we compared our approach (S-RTS) with a pure dependence-based approach (D-RTS) [7] are percentage reduction in size of regression test suite and the percentage of detected faults over the faults injected earlier. The results obtained from the experiments are summarized in Table 1.

From the experimental results, it can be observed that in all cases both approaches detected 100% faults which indicate that both approaches revealed every fault that we had injected earlier. In contrast, S-RTS achieves higher regression suite reduction in all the cases as compared to D-RTS. It can also be observed that test case reduction effectiveness of the approaches does not depend on the

size of the input program, rather the degree of dependence inside a component can affect the effectiveness of a test case selection approach. The more the program elements are interdependent on each other, the more number of statements get affected by a given change. Therefore, when there is a strong dependency among program elements, injected faults affect a large number of statements, causing larger size of change information and causing increased size of the regression test suite. This aspect is probably reflected in the ATM case study. It is found that, for ATM case study, D-RTS does not reduce the size of regression test suite and even for S-RTS, percent of reduction is only 10.

Our empirical studies show that on the average D-RTS results in 25.72% reduction in initial test suite, whereas S-RTS achieves about 42.64% reduction without affecting the fault revealing effectiveness.

6. Comparison with Related Work

Several research results have been reported in the area of regression testing for a component-based system [1, 3, 5, 7, 22–24]. We compare these with our work.

Orso et al. [24] proposed an RTS approach that makes use of metadata and metamethods provided by the component vendor. As described by them, metadata consists of various forms of static data about the source code, and metamethods compute or retrieve such information as well as collect dynamic information such as execution traces during test case execution. In their approach, both instrumented and noninstrumented versions of each *published method* are supported by the component and the application developers can decide which one to use. During test case execution, application developers execute the instrumented versions of the component methods. By invoking metamethods, they obtain the coverage data in terms of edges of the control flow graph (CFG) of the component. For every new release of the component, change information is published in terms of CFG edges affected by the modification. Finally, application developers choose regression test cases based on the change information and the collected coverage data. In their method, however, component vendors have to provide too detailed change information, and they also need to incorporate built-in coverage facilities so that application developer can obtain execution traces during test case execution. Moreover, their approach does not seem to consider data and control dependencies among statements while computing the change information.

Sajeev and Wibowo [5] proposed an RTS approach which selects all test cases that directly or indirectly call a changed component method. But all test cases that invoke a changed method may not execute the modified region of the code. As a consequence, the approach may select redundant test cases for regression testing.

Beydeda and Gruhn [22] proposed an abstract graphical representation called Component-Based Software Flow Graph (CBSFG) to represent a component. It combines the features of *Class Control Flow Graph* [25] and *Finite State Machine* [26] representation of the component. They generated black box test cases based on the CBSFG constructed from the component-based system. They also proposed a test selection technique based on the technique proposed by Rothermel et al. [25]. However, it is not clear how application developers can obtain the necessary information for constructing the CBSFG when source code is unavailable. It is also not clear that how coverage data of the test cases can be obtained in terms of the elements of the CBSFG to select regression test cases.

Pan et al. [7] proposed a technique for selecting regression test cases by slicing the system dependence graph constructed from a component-based system. Their proposed SDG model captures interstatement/interoperation/inter-interface control and data dependence relationships as well as dependence between components and their execution contexts. At first, a method dependence graph is constructed for each method, and then its summary information is used to compose an operation dependence graph for each operation defined in an interface. Similarly, interface dependence graphs, component dependence graphs, and the system dependence graph are recursively constructed. They presented a slicing algorithm with which the affected operations, calling contexts, and dependence relationships can be identified. However, when source code of a component is not made available to the application developer, interstatement dependencies cannot be determined by them. Consequently, affected elements can be identified no better than affected operations. As a result, the selected regression test suite may contain redundant test cases which invoke the affected operations but do not execute any affected statements.

In our approach, we analyze a dependence model of a component to identify the affected elements. We further analyze the state model of the component to identify the affected transitions due to a modification. A test case is chosen for regression testing only when it exercises an affected transition. Consequently, our approach yields a smaller regression test suite as compared to the approaches which select test cases only based on affected *published methods*. At the same time, our approach does not compromise the quality of regression testing.

7. Conclusion

We have proposed an RTS technique that uses dependence analysis to identify the affected elements. It involves an analysis of the component state model to reduce the number of test cases for regression testing. Our experimental studies indicate that S-RTS selects lower number of test cases for regression testing as compared to related approaches. The approach is based on the assumption that there is a strong correspondence between the design level state model and the executable code. Our technique can be easily used when the code is autogenerated from the state model. However, the technique can also be used in case the code is not autogenerated from the state model, if the state model can be reverse engineered from the source code.

References

[1] Y. Wu, D. Pan, and M. H. Chen, "Techniques of maintaining evolving component-based software," in *Proceedings of the IEEE Interantional Conference on Software Maintenance (ICMS '00)*, pp. 236–246, October 2000.

[2] S. Beydeda and V. Gruhn, "Black- and white-box self-testing cots components," in *Proceedings of the International Conference on Software Engineering and Knowledge Engineering*, pp. 104–109, 2004.

[3] C. Mao, "Built-in regression testing for component-based software systems," in *Conference on Computer Software and Application*, vol. 2, pp. 723–728, July 2007.

[4] C. R. Rocha and E. Martins, "A method for model based test harness generation for component testing," *Journal of the Brazilian Computer Society*, vol. 14, no. 1, pp. 7–23, 2008.

[5] A. S. M. Sajeev and B. Wibowo, "UML modeling for regression testing of component based systems," *Electronic Notes in Theoretical Computer Science*, vol. 82, no. 6, pp. 190–198, 2003.

[6] A. Podgurski and L. A. Clarke, "A formal model of program dependences and its implications for software testing, debugging, and maintenance," *IEEE Transactions on Software Engineering*, vol. 16, no. 9, pp. 965–979, 1990.

[7] Y. Pan, D. Pan, and M. H. Chen, "Slicing component-based systems," in *Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '05)*, pp. 155–164, June 2005.

[8] K. J. Ottenstein and L. O. Ottenstein, "The program dependence graph in a software development environment," in *Software Development Environments (SDE)*, pp. 177–184, 1984.

[9] S. Horwitz, "Identifying the semantic and textural differences between two versions of a program," in *Proceedings of the Conference on Programming Language Design and Implementation (PLDI '90)*, pp. 234–245, June 1990.

[10] L. Larsen and M. J. Harrold, "Slicing object-oriented software," in *Proceedings of the 18th International Conference on Software Engineering (ICSE '96)*, pp. 495–505, March 1996.

[11] D. Liang and M. J. Harrold, "Slicing objects using system dependence graphs," in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM '98)*, pp. 358–367, November 1998.

[12] N. Walkinshaw, M. Roper, and M. Wood, "The java system dependence graph. In," in *International Working Conference on Source Code Analysis and Manipulation*, pp. 55–64, 2003.

[13] J. Z. Gao, D. C. Kung, P. Hsia, Y. Toyoshima, and C. Chen, "Object statetesting for object-oriented programs," in *Conferenceon Computer Software and Application*, pp. 232–238, 1995.

[14] J. Z. Gao, D. C. Kung, and P. Hsia, "An object state test model: object state diagram," in *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research (CASCON '95)*, p. 23, 1995.

[15] D. C. Kung, N. Suchak, J. Gao, P. Hsia, Y. Toyoshima, and C. Chen, "On object state testing," in *Proceedings of the 18th Annual International Computer Software & Applications Conference*, pp. 222–227, November 1994.

[16] Mathworks, Simulink stateflow, http://www.mathworks.com/products/stateflow/.

[17] D. Harel and E. Gery, "Executable object modeling with statecharts," *Computer*, vol. 30, no. 7, pp. 31–42, 1997.

[18] P. C. Mehlitz, "Trust your model—verifying aerospace system models with Java Pathfinder," in *Proceedings of the IEEE Aerospace*, March 2008.

[19] I. A. Niaz and J. Tanaka, "Mappinguml statecharts to java code," in *Proceedings of the The International Association of Science and Technology for Development, Software Engineering*, pp. 111–116, 2004.

[20] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Massachusetts, Mass, USA, 1995.

[21] G. Rothermel and M. J. Harrold, "Selecting regression tests for object-oriented software," in *Proceedings of the International Conference on Software Maintenance (ICSM '94)*, pp. 14–25, May 1994.

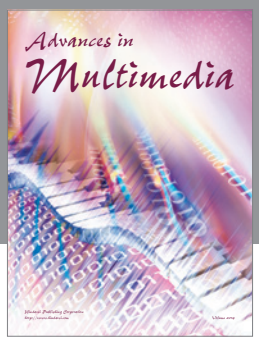
[22] S. Beydeda and V. Gruhn, "Testing component-based systems using fsms," in *Testing Commercial-off-the-Shelf Components and Systems*, S. Beydeda and V. Gruhn, Eds., pp. 363–379, Springer, Berlin, Germany, 2005.

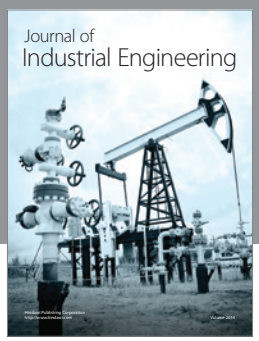
[23] C. Mao, Y. Lu, and J. Zhang, "Regression testing for component-based software via built-in test design," in *ACM Symposium on Applied Computing*, pp. 1416–1421, March 2007.

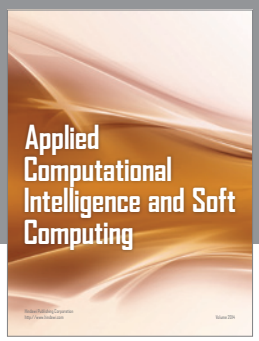
[24] A. Orso, M. J. Harrold, D. Rosenblum, G. Rothermel, M. L. Soffa, and H. Do, "Using component metacontent to support the regression testing of component-based software," in *Proceedings IEEE International Conference on Software Maintenance (ICSM '01)*, pp. 716–725, November 2001.

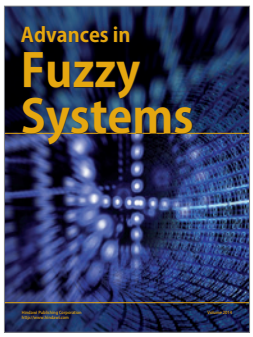
[25] G. Rothermel, M. J. Harrold, and J. Dedhia, "Regression test selection for C++ software," *Software Testing, Verification and Reliability*, vol. 10, no. 2, pp. 77–109, 2000.

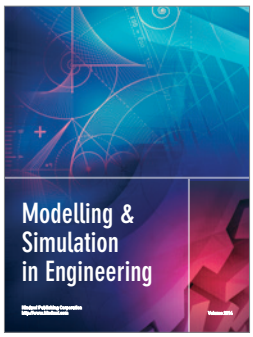
[26] H. S. Hong, Y. R. Kwon, and S. Deok Cha, "Testing of object-oriented programs based on finite state machines," in *Proceedings of the Software Engineering Conference*, pp. 234–241, Asia Pacific, December 1995.

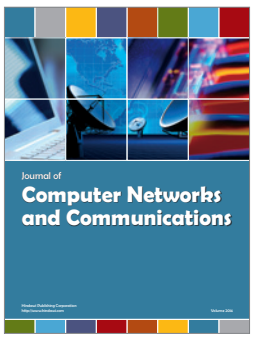












Hindawi
Submit your manuscripts at
http://www.hindawi.com