International Journal of Engineering, Mathematical and Physical Sciences
ISSN: 2517-9934
Vol:4, No:2, 2010

A General Regression Test Selection Technique

Walid S. Abd El-hamid, Sherif S. El-etriby, and Mohiy M. Hadhoud

***Abstract*—**This paper presents a new methodology to select test cases from regression test suites. The selection strategy is based on analyzing the dynamic behavior of the applications that written in any programming language. Methods based on dynamic analysis are more safe and efficient. We design a technique that combine the code based technique and model based technique, to allow comparing the object oriented of an application that written in any programming language. We have developed a prototype tool that detect changes and select test cases from test suite.

***Keywords*—**Regression testing, Model based testing, Dynamic behavior.

I. Introduction

SOFTWARE maintenance is an expensive phase accounting for near 60% of overall cost of software life cycle expenditure [3]. Regression testing is an important step in software development to ensure that modifications do not break previously working functionality. However, regression testing is often expensive and time consuming. Regression test suites can be very large, e.g. including tens of thousands of test cases requiring days or weeks to execute [1].

Regression test selection is the activity that choosing from an existing test set, test cases that can and need to be rerun to ensure that changed parts behave as intended and the changes did not introduce unexpected faults. Reducing the number of regression test cases to execute is an obvious way of reducing the cost associated with regression testing. The main objective of selecting test cases that need to be rerun is to identify regression test cases that exercise modified parts of the system. This is referred to as safe regression testing as, it identifies all test cases in the original test set that can reveal one or more faults in the modified program [12].

There are many techniques that handle regression testing, some of them based on source code and other based on design.

The techniques that based on source code are more safe and easy to make. But, it requires that the changes be already implemented. These techniques are very specific to the programming language used to develop the software. Where, if an application is builted using functional languages such as C. Hence, it is not suitable to analyze applications built using C# and Java because the tool cannot identify indirect changes due to object oriented features of these languages like dynamic binding, exceptions etc.

Other techniques that based on specification are more general where the designs are represented using the Unified Modeling Language (UML) that independent on programming language. But, some changes to the source code may not be detectable from UML documents so cannot detect all test cases for the changes.

In this paper we present a new approach that overcomes these shortcomings has been proposed. The approach is based on combining the code based technique and model based technique together to generate a safe and general regression test selection technique. Our approach capture and analyzing the dynamic behavior of the software applications from UML diagram. Then identify the impact of changes made to software, and based on this it selects test cases to be re-executed. These test cases are fewer in number when compared to the complete system test suite.

The rest of the paper is organized as follows. Section II discusses different regression test selection techniques that are available in literature. Section III presents in detail the proposed approach to regression test selection. The results of the case studies are presented in Section IV. Conclusions and future works are summarized in Section V.

II. Related Work

Typically regression test selection techniques are either code-based or model-based. Code-based techniques use the information obtained from two different versions of the code to analyze the change impact and select the tests. In the case of model based techniques, change information is obtained through two versions of models constructed during the requirements analysis phase or system design phase.

Code based techniques [2], [5], [6], [7], [11], [12] select tests based on changes made to two versions of the code. These techniques are very specific to the programming language used to develop the code. Chianti [10] and JDiff [5] are comprehensive techniques for managing changes in Java programs. Chianti selects regression tests after analyzing the change impact analysis whereas JDiff performs only change impact analysis. As both these tools analyses the changes at statement level and are specific to Java programming language, hence, they are neither generic nor efficient.

Model-based techniques [3], [4], [8], [9] are based on UML design models used during the design phase of the system. Reference [15] use UML activity diagrams to detect changes in design and then use a traceability matrix between activity diagram and the test suite. It covers activities at an abstract level and does not cover the attributes of a class. Also, it does not support object-oriented features. Reference [8] proposes a

W. S. Abd El-hamid is with the Computer Science Department, Menofyia University, Egypt, (e-mail: walid_mufic@yahoo.com.)

S. S. El-etriby is with Computer Science Department, Egypt, Menofyia University, (e-mail: El_Etriby10@yahoo.com).

M. M. Hadhoud is with Information Technology Department, Egypt, Menofyia University, (e-mail: mnhadhoud@yahoo.com).

regression testing technique based on UML sequence and class diagrams. Their approach does not take into account the pre and post conditions of the operations which affect behavior of a class. Also, their approach does not handle concurrency.

III. Our Regression test Selection Technique

Our proposed approach to regression test selection is based on changes made to software specification that represented in UML diagram and code that represented in any programming language. Our approach is consists of three functions as shown in Fig. 1. These three functions are (1) Capture dynamic behavior, (2) Identify changes, (3) Select regression test suite. Each of these functions has been described in details.

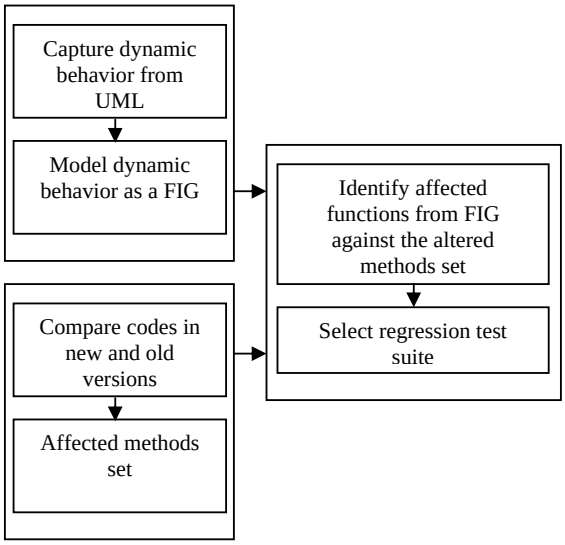

Fig. 1 Block diagram of our approach

A. Capturing Dynamic Behavior of the Application

Dynamic behavior of software is a set of interactions among system components along with their invoked classes/functions across all application processes. We captured dynamic behavior of the system from UML Class diagram and Sequence diagram. The captured behavior is modeled into Interclass Relation Graph (IRG) and Functional Interaction Graph (FIG). The Interclass Relation Graph (IRG) for a program is a triple {*N, IE,UE*}:

- *N* is the set of nodes, one for each class.
- *IE* is the set of inheritance edges. An inheritance edge between a node for class *C1* and a node for class *C2* indicates that *C1* is a direct sub-class of *C2*.
- *UE* is the set of use edges. A use edge between a node for class *C1* and a node for class *C2* indicates that *C1* contains an explicit reference to *C2*

Program P

```
public class SuperA {
public void F6 ( ) {
System.out.println("aa") ;
}
}
public class A extends SuperA {
public void F4( ) { F6(); }
public void F5( ) {…. }
}
public class SubA extends A {}
public class B {
A a = new A();
public void F1 ( ) {a.F4();}
public void F2 ( ) {a.F4();}
public void F3 ( ) {a.F5();}
}
public class SubB extends B {}
public class C {
public static void main ( ) {
B b=new B();
b.F1() ;
b.F2();
b.F3();
}
}
```

Fig. 2 Example program *P*

Fig. 3 shows the algorithm for building an IRG, *buildIRG*. For simplicity, in defining the algorithms, we use the following syntax: n_e indicates the node for a type *e* (class or interface); G_N, G_{IE}, and G_{UE} indicate the set of nodes *N*, inheritance edges *IE*, and use edges *UE* for a graph *G*, respectively. Algorithm *buildIRG* first creates a node for each type in the program (lines 2–5). Then, for each type *e*, the algorithm connects *ne* to the node of its direct super-type through an inheritance edge (lines 7–8), and (2) creates a use edge from each n_c to n_e, such that *c* contains a reference to *e* (lines 9–11).

```
Algorithm buildIRG
Input: program P
Output: IRG G for P
Begin buildIRG
1: create empty IRG G
2: for each class and interface in P do
3: create node n_e
4: G_N = G_N ∪ {n}
5: end for
6: for each class and interface in P do
7: get direct super-type of e, s
8: G_IE = G_IE ∪ {( n_e , n_s )}
9: for each class c in P that e references do
10: G_UE = G_UE ∪ {( n_c , n_e}
11: end for
12: end for
13: return G
End buildIRG
```

Fig. 3 Algorithm for building an IRG

Fig. 4 shows the IRG for program *P* in Fig. 2. The IRG represents the six classes in *P* and their inheritance and use relationships. The figure has three inheritance relationships and two use relationships. The class *A* inherits from class *SuperA*, class *SubA* inherit from class *A* and class *SubB* inherit from class *B*. So if class *A* is changes then we need to test class *A*, class *SuperA* and class *SubA*.

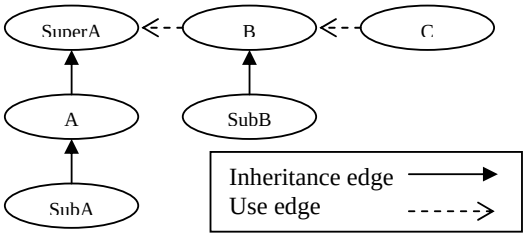

Fig. 4 IRG for program P of Fig. 2

After we draw the IRG we can draw the sequence diagram of the original program to detect the relations between functions as shown in Fig. 5.

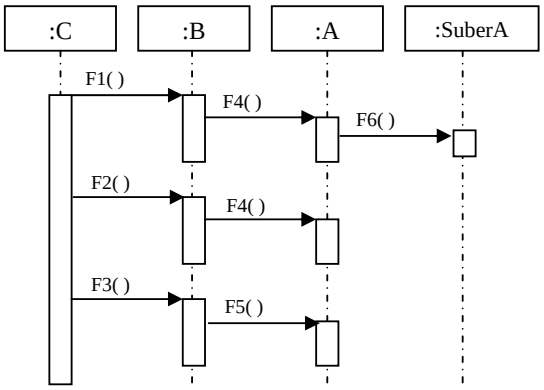

Fig. 5 Sequence diagram for Program *P* in Fig. 2

Fig. 6 shows the FIG for program *P* in Fig. 2 where we use UML sequence diagram in Fig. 5 to capture the relationship between functions.

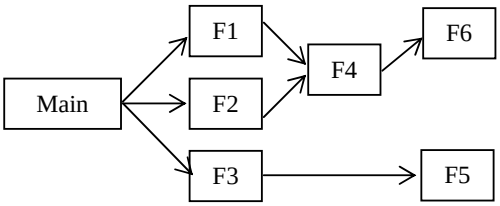

Fig. 6 FIG for Program *P* of Fig. 1

B. Identify Affected Methods

Affected methods are identified by comparing the original program and modified program. Changes to code occur at the syntactic and semantic levels. Code changes due to a change in syntax refer to the textual differences between corresponding line statements of code versions of a program. A syntactic difference may not necessarily cause a change in the semantics of the program. For example, consider that *int sum = a + b + c*; statement is replaced with two statements (1) *int sum = a + b*; and (2) *sum = sum + c*; in the new version of the software. There is a change syntactically between corresponding lines of code. However, semantically the final value assigned to variable *sum* is the added value of variables *a*, *b*, and *c* in both the cases, hence, it is considered as no change. To resolve such problems, data flow analysis techniques, based on program slicing, have been devised [13][14]. Although slicing of program statements is a safe and precise method, it is overly complex and necessitates heavy usage of memory and processing time. Thus, scaling slicing techniques to large programs would be difficult and too costly in terms of performance.

The semantic change involves identifying indirectly affected methods which might get invoked due to polymorphism, dynamic binding and exceptions features.

Dynamic binding, Because of dynamic binding, an apparently harmless modification of a program may affect call statements in a different part of the program with respect to the change point. For example, class-hierarchy changes may affect calls to methods in any of the classes in the hierarchy, and adding a method to a class may affect calls to the methods with the same signature in its superclasses and subclasses. As shown in Fig. 7, Class B inherits from Class A and a virtual method *calc()* is implemented in both the classes. The method *calc()* in class A has been changed in new version of the code. This changed method will affect the execution of method *func1()* of class D as the argument passed to that method can also be an object of type A due to inheritance property (parent is a sub-type of a child). Therefore, we marks both methods *A.calc()* and *D.func1()* as changed.

Old Program *P*	New Program *P′*

Class A **:** System.Array{
int sum;
virtual int calc() {
return sum;
}}
Class B**:**A{
int calc() { return sum/3;
}}
Class D {
void int func1(B obj)
{ return obj.calc(); }
void func2() {
try{…}
catch(e1){…}
catch(e2){…}
}
void func3()
{
try{…}
catch(e1){return;}
catch(e2){…}
}
	Class A{
	int sum;
	virtual int calc() {
	return sum*sum;
	}}
	Class B**:**A{
	int calc() { return sum/3;
	}}
	Class D{
	void int func1(B obj)
	{ return obj.calc(); }
	void func2() {
	try{…}
	catch(e1){…}
	
	}
	void func3()
	{
	try{…}
	catch(e1){func3();}
	catch(e2){…}
	}

Fig. 7 Original program *P* and its modified *P′*

Changes to an inheritance tree also affect the execution of methods in that tree. For instance, consider the new version of Class A shown in Fig. 7. The new version Class *A* does not inherit from System.Array. This change will influence the execution of all the methods in Class *A*. Hence, all methods in Class *A* are marked as changed. This change also influences the execution of Class *B* since it inherits the changed Class *A*. Therefore, all the methods of Class *B* are also marked as changed. The changes to an inheritance tree are identified by simple comparison of two inheritance tree objects of both old and new versions of a component. The methods (of both Classes *A* and *B*) are identified from the inheritance tree object and are marked as changed.

Changes to exceptions can occur at two levels. One during the handling of the exceptions and another is at the definition of exceptions. For instance consider the changes made to exceptions handling as shown in Fig. 7. In the new version, method *func2*() doesn't handle exception *e2* whereas method *func3*() handle both exceptions but has changed its implementation while handling exception of type e1. In such cases both methods *func2*() and *func3*() are marked as changed methods.

To make automation of change impact analysis complete, both syntactic and semantic changes to a program should be considered. Our technique identifies methods affected due to both syntactic and semantic changes made to software written using any programming language.

C. Selecting Smaller Regression Test Suite

On identifying the affected methods, we can find the impact of these changed methods by analyzing the FIG and IRG. For example, in Fig. 6, if method *F4* is marked as changed in the new version of the software, then *F4*, *F1* and *F2* are marked as changed so any test case pass in these functions are selected.

IV. Case Studies

In this section we apply our approach on two case studies. The first one is software that written in Java language called AlarmClock, and the second is software that written in C++ language called Schedule.

In the first case study the system test suite consists of 90 test cases. The case study has been conducted on three upgrades released during application regression testing cycle. These upgraded consists of mainly bug fixes, like change to source code statements, deletion of methods, adding new methods. The original program has 6 classes and 20 methods and when we apply our approach on this case study we get the resulted that are tabulated in Table I.

TABLE I
RESULTS OF THE FIRST CASE STUDY

# Version	# Test cases selected	% of test effort saved
V1	16	81%
V2	42	53%
V3	27	70%

In the second case study the system test suite consists of 150 test cases. The case study has been conducted on four upgrades released during application regression testing cycle and we get the resulted that are tabulated in Table II.

TABLE II
RESULTS OF THE SECOND CASE STUDY

# Version	# Test cases selected	% of test effort saved
V1	76	49%
V2	84	44%
V3	65	65%
V4	72	52%

V. Conclusion and Future Work

In this paper we present a new approach that is based on combining the code based technique and model based technique together to generate a safe and general regression test selection technique. Where we capture the dynamic behaviors of the software applications from UML diagrams. Then identify the impact of changes made to software code that written in any programming language, and based on these changes we select test cases to be re-executed. These test cases are fewer in number when compared to the complete system test suite.

Software maintenance also includes addition and deletion of user functionality. These modifications could be classified as major changes. Often these changes require new test cases to be added/deleted or modify existing test cases. Our future research would focus on investigation of techniques that automatically identify major changes made to code and generate test cases that validate these changes.

References

[1] G. Wikstrand, R. Feldt, J.K Gorantla, Zhe Wang, C. White, "Dynamic regression test selection based on a file cache- an industrial evaluation", International Conference on Software Testing Verification and Validation, 2009, pp 299-302.

[2] Anjaneyulu Pasala, Yannick LH, Fady A, Appala Raju G and Ravi P Gorthi, "Selection of regression test suite to validate software applications upon deployment of upgrades", 19th Australian Software Engineering conference, 25-28 March 2008, pp 130-138.

[3] Ravi P Gorthi, Anjaneyulu Pasala, Kailash KP and Benny Leong, "Specification-based approach to select regression test suite to validate change software", 15th Asia-Pacific Software Engineering conference, 2008, pp 153-160.

[4] L.C. Briand, Y. Labiche and S. He, "Automating regression test selection based on UML designs" , 2008, pp 16-30

[5] Apiwattanapong, T., Orso, A., and Harrold, M.J., "JDiff: A Differencing Technique and Tool for Object--Oriented Programs", Journal of Automated Software Engineering, Vol 14, No. 1, March 2007, pp 3-36.

[6] A. Orso, N. Shi and M.J. Harrold, "Scaling regression testing to large software systems", Proceeding of the 12th ACMSIGSOFT International Symposium on Foundation of Software Engineering, 2004, pp 241-251.

[7] T. Koju, S. Takada, N. Doi, "Regression test selection based on intermediate code for virtual machines", Proceeding of International Conference on Software Maintenance (ICSM 03), 2003 , pp 1-10.

[8] Orest P, Hunay U, and Andrews A, "Regression Testing UML Designs", Proceedings of 22nd IEEE International Conference on Software Maintenance (ICSM), Philadelphia, Pennsylvania, September 24-27, 2006, pp254-264.

[9] A. Ali, A. Nadeem, M.Z. Iqbal, M. Usman, "Regression testing based on UML design models", 13th IEEE International Symposium on Pacific Rim Dependable Computing, 2007, pp 85-88.

[10] Xiaoxia R, Barbara G R, Maximilian S and Frank T, “Chianti: A prototype change impact analysis tool for Java”, Proceedings of 27th international conference on Software engineering (ICSE), St. Louis, USA, May 15-21, 2005, pp 664-665.
[11] M.J. Harrold, J.A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S.A. Spoon, "Regression test selection for java software", in: Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA’01), 2001, pp 312-326.
[12] A. Orso, N. Shi and M.J. Harrold, "Scaling regression testing to large software systems", Proceeding of the 12th ACMSIGSOFT International Symposium on Foundation of Software Engineering, 2004, pp 241-251.
[13] Pócza K, Biczó M, and Porkoláb Z, “Cross language Program Slicing in the .NET Framework” Proceedings of 3rd International Conference on .NET Technologies, Plzen, Czech Republic, May 2005.
[14] Zhang X and Gupta R, “Cost Effective Dynamic Program Slicing”, Proceedings of ACM SIGPLAN Conference on Programming language design and implementation, June 2004, pp 94 – 106.
[15] Y. Chen, R.L. Probert, D.P. Sims, Specification based Regression test selection with risk analysis, in: Proceedings of Conference of the Center for Advance Studies on Collaborative Research, 2002.
[16] Anjaneyulu P, Yannick LH Lew, and Ravi Prakash G, “How to Select Regression Tests to Validate Applications upon Deployment of Upgrades”, Vol. 6, No. 1, 2008, pp 55 – 62.
[17] G. Rothermel, M.J. Harrold, "Analysing regression test selection techniques", IEEE Transactions on Software Engineering , 1996, pp 529–551.
[18] E. Martins and V.G. Vieira, "Regression test selection for testable classes", ENCS 2005, pp 453-470.
[19] H. Agrawal, J.R. Horgan, E.W. Krauser and S.A. London, "Incremental Regression Testing", Proceedings of IEEE Conference on software Maintenance,1993, pp 348-357.
[20] T. Apiwattanapong, A. Orso, M.J. Harrold, "A differencing algorithm for object-oriented programs", Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE 2004), 2004, pp 2–13.
[21] G. Rothermel, M.J. Harrold, J. Debhia, Regression test selection for C++ software, Journal of Software Testing, Verification, and Reliability, 2000 pp 77–109.