Innovations Syst Softw Eng (2016) 12:153–159
DOI 10.1007/s11334-015-0262-6

ORIGINAL PAPER

Regression test size reduction using improved precision slices

Chhabi Rani Panigrahi[1] · **Rajib Mall**[2]

Received: 23 April 2014 / Accepted: 15 October 2015 / Published online: 4 November 2015

Abstract We propose a technique to select regression test cases that is targeted to reduce the size of test suite by using improved precision slices. For this, we first construct the control flow graph model of an object-oriented program. Subsequently, we identify the infeasible paths and compute the definition-use (*def-use*) pairs only along the feasible paths. Next, we construct the dependency model using this information. This helps us to ignore the dependencies existing along infeasible paths leading to construction of precise slices. Then, we build the dependency model and construct forward slices on the dependency model to determine the affected nodes and the test cases that cover the affected nodes in the dependency graph model are selected for regression testing. The results obtained from our experimental studies indicate that our approach reduces the regression test suite size on an average by 11.25 % as compared to a related approach, without degrading the *fault-revealing* effectiveness.

Keywords Regression testing · Regression test selection · Infeasible paths · Static slice

1 Introduction

Regression testing has been reported to be one of the most expensive maintenance activities [1,2]. A large number of regression test selection (RTS) techniques have been proposed in the literature to reduce the cost of regression testing and were proposed in the context of various programming paradigms [3–9]. An RTS technique selects a subset of test cases from the original test suite and uses it to revalidate the unmodified portions of a program.

Literature study indicates that RTS techniques proposed for object-oriented programs are based on either code analysis [4,7,8] or are based on the analysis of design models [10–12]. The code analysis based RTS techniques were proposed based on either control flow [4,7] or the dependency model [8]. In an approach based on an analysis of dependency model, the set of affected nodes in the dependency model are determined by constructing forward slices and the test cases which cover one or more affected nodes are selected for regression testing. However, during dependency model construction, all program paths are considered to be feasible [13,14] and dependencies are computed along all paths. Hence, the existence of infeasible paths in a program leads to computation of infeasible *def-use* pairs which results in construction of imprecise slices. An *infeasible path* is one, which can not be exercised by any possible input values to the program [15].

It has been reported that on the average 12.5 % of all the program paths in a typical program are infeasible [16]. Although, detecting all infeasible paths is an intractable problem [17], various approaches have been developed to identify some of the infeasible paths in a program. Most of the techniques used for identifying infeasible paths are based on symbolic execution [18–22]. But, there are certain limitations of using symbolic evaluation such as handling pointers, function calls, and arrays and hence by using this approach a small percentage of infeasible paths may be detected. Several heuristics based approaches [23–25] have been reported and more recently Ngo et al. [17] proposed a technique to

✉ Chhabi Rani Panigrahi
panigrahichhabi@gmail.com

Rajib Mall
rajib@cse.iitkgp.ernet.in

[1] C.V. Raman College of Engineering, Bhubaneswar 752054, Odisha, India

[2] Indian institute of Technology Kharagpur, Kharagpur 721302, India

detect infeasible paths based on recognition of code patterns existing in methods of an object-oriented program. In this context, we proposed an improved slicing technique for object-oriented programs by eliminating *def-use* pairs computed along infeasible paths [15]. In [15], we used the approach proposed by Ngo et al. [17] to detect the infeasible paths. We compute *def-use* pairs only along feasible paths and build a dependency model using this information. This helps us to construct smaller and precise slices. Therefore, this constructed dependency model can be used to reduce the number of regression test cases.

The rest of the paper is organized as follows. We discuss some basic concepts that are required to understand our approach in Sect. 2 and the reported work on RTS based on code analysis is described in Sect. 3. Section 4 presents our proposed technique to select regression test cases and in Sect. 5, we discuss our RTS approach based on analysis of the constructed dependency model. We present the results obtained from our experimental studies in Sect. 6 and comparison of our approach with a related approach is presented in Sect. 7. Section 8 concludes the paper.

2 Basic concepts

In this section, we discuss certain background concepts related to RTS that are needed to understand our work. We first present some definitions used in the context of RTS and then discuss a dependency graph model for object-oriented programs.

2.1 Concepts related to regression test selection

In the following, we present a few important terms and concepts relevant to RTS.

2.1.1 Retestable, redundant, and obsolete test cases

Leung et al. [2] classified the initial test suite into *retestable*, *redundant*, and *obsolete*. Test cases that need to be rerun during regression testing are called *Retestable* and these test cases exercise the modified as well as the affected portion of a program. Test cases that exercise only the unaffected portion of a program are called *Redundant*. Although the redundant test cases are valid for the modified program, these can be omitted from the regression test suite. Test cases that are no longer valid for the modified program are called *Obsolete* and these need to be discarded.

2.1.2 Fault-revealing test cases

Let P and P' be the original and modified program respectively. Let T be the original test suite for P. A test case $t \in T$ is called *fault-revealing* for P', iff by executing P' with t results in the incorrect outputs for P' [5].

2.1.3 Regression testing cycle

A regression testing cycle includes activities such as test setup, prioritization, selection, and execution of test cases and preparation of failure report. Each time a modification is made to a software, regression testing is carried out to validate the modified portion of a software. After successful completion of regression testing, a new version of the software is released. The new version also then passes through similar testing cycles and before each release of a software there can be a number of regression testing cycles.

2.2 Dependency graph model of object-oriented programs

The *system dependence graph* (SDG) was first introduced by Horwitz et al. [13] and was used to model procedural programs. Later on, Larsen et al. extended the SDG to model object-oriented programs [26] and they named their proposed model as *extended system dependency graph* (ESDG). In an ESDG model, each class is represented by a class dependence graph (ClDG). The ESDG model is described in more detail in [26] and is used in our RTS approach.

3 Related work

A number of RTS approaches have been proposed in the literature for object-oriented programs [4,7,8,10–12,27–30]. Here, we restrict our discussion to only those that are based on code analysis [4,7,8].

Rothermel and Harrold [8] proposed an RTS approach for object-oriented programs based on a dependency graph model. They performed RTS for application programs as well as for classes. For RTS of an application program, they used *inter-procedural program dependence graph* (IPDG) to model the original as well as the modified program. An IPDG models a program as having a single entry point. However, a class has an entry point corresponding to each method. Therefore, it is difficult to use an IPDG for RTS of nontrivial classes. For RTS of modified classes, they used ClDG of both the original (G) and modified classes (G'). A *representative driver node* (RDN) serves as a root of the graph that summarizes the set of drivers for test cases. Each public method in a class is made as a child of the root and are connected with RDN by driver edges. The test coverage information is used to associate the predicate and statement nodes of the ClDG model with each test case. Then, a depth-first traversal of G and G' is performed starting from the root nodes of both the graphs. If a pair of nodes n and n' is found in G and

G' such that they are not identical or their control dependent successors are not identical, the test coverage information associated with the node n is used to select regression test cases.

Rothermel et al. [7] proposed an RTS technique for C++ programs and they used control flow graph model as an intermediate representation of programs. Their technique can be applied to both classes as well as to the application programs. For RTS, they used inter class control flow graph (ICCFG) to model an application program and a Class Control Flow Graph (CCFG) to model a class. They computed execution trace of a test case as the set of branches covered by that test case in the control flow graph model. When a class is modified, two CCFGs that is G and G' are constructed corresponding to the original and modified classes. As in [8], a depth-first traversal is performed starting with the entry nodes of G and G', looking for a pair of nodes n and n' whose associated statements are not lexicographically equivalent. Whenever the targets of identically labeled edges in both G and G' differ, the edge is identified as an affected edge. When the algorithm finds such a pair, it uses execution trace of test cases to select the test cases that cover one or more affected edges.

Harrold et al. proposed an RTS technique for Java programs [4] by extending their earlier work on RTS [7]. Their technique performs RTS in three steps: constructing intermediate representations for both the original and modified programs, analyzing the graph models and determining the set of affected edges, and selection of test cases. Their technique used Java Interclass Graph (JIG) as an intermediate representation. JIG represents various Java language features such as inheritance, polymorphism, and exception handling. The JIGs constructed for both P and P' are simultaneously traversed and depth-first search is used to identify the affected edges. The affected edges are determined by using the algorithm proposed by Rothermel et al. [7]. Subsequently, based on the test coverage information previously obtained through code instrumentation, they selected the test cases that cover the affected edges.

4 Our proposed approach

We named our approach to select regression test cases as improved slicing-based regression test Selection (I-RTS). In our RTS approach, we first construct the control flow graph model of an object-oriented program. Next, we identify infeasible paths using an existing approach proposed by Ngo et al. [17] and compute def-use pairs only along feasible paths. We build the dependency model using this information and for more details please refer to [15]. Subsequently, we construct slices on the dependency model using Horwitz et al.'s algorithm [13]. We select those test cases for regression testing that cover the affected nodes in the dependency model.

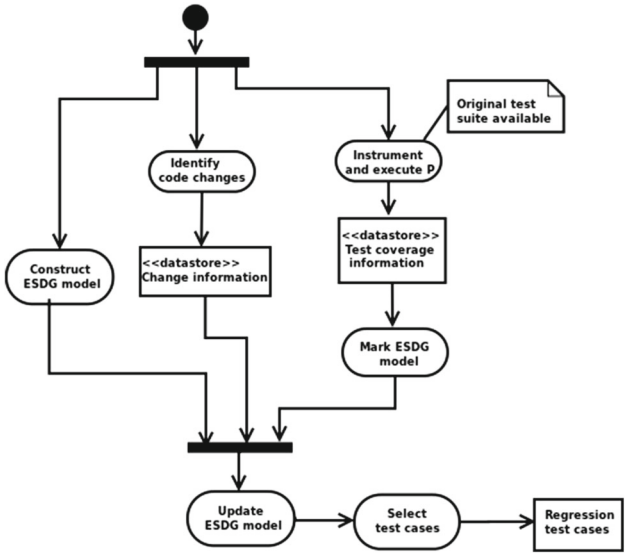

Fig. 1 Activity diagram representation of I-RTS

The important steps of I-RTS have been shown in Fig. 1 using an activity diagram representation. We have used an activity diagram to represent the parallel execution and synchronization aspects of activities involved in our RTS approach. The important activities in I-RTS as shown in Fig. 1 include constructing the ESDG model for an object-oriented program, collecting coverage information of test cases, and updating the ESDG model. Some activities such as collecting test coverage information and marking the coverage information in ESDG model are performed in the first regression test cycle as mentioned in Sect. 2.1.3 and are not repeated in later regression testing cycles. The different activities that are carried out during the first regression testing cycle are described as follows.

- *Construct ESDG model* This activity is involved with the construction of the ESDG model for P from its control flow graph model that is ICCFG and is done by computing *def-use* pairs only along the feasible paths as in [15].
- *Identify changes* In this step, statement-level changes between P and P' are identified. The identified changes are stored in a file named *diff*. Each entry in the *diff* file contains the changed statements in P and P'. This information is represented as *Change information* as shown in Fig. 1.
- *Instrument and execute the program* This activity is involved with the instrumentation of P and is done at the basic block level. The instrumented code is executed with T to generate test coverage information. The test coverage information is logged in a file named *Coverage* and is saved for later usage.

– *Mark ESDG model* In this step, the coverage information from *Coverage* is marked on ESDG model. That is, the specific program statement(s) that are covered by the test cases are recorded in the corresponding node(s) in the ESDG model.
– *Update ESDG model* In this step, the modifications made to P are updated on the constructed ESDG model M so that it corresponds to P' and the updated ESDG model is denoted by M_u. The nodes which are marked due to changes are tagged in M_u and are denoted by *Tagged*. The details of updating of ESDG model for different types of changes can be found in [31].
– *Select test cases* In this step, test cases which cover affected nodes in the ESDG model are selected. The detail of this step is discussed in Sect. 5.

Algorithm 1: Pseudo code to select regression test cases by computing ESDG model slice.

```
Input: M_u, Tagged
Output: T_SEL
1  procedure ESDGMSlice(M_u, Tagged, T_SEL )
2  AffectedNodes ← NULL
3  T_SEL ← NULL
4  for each node n in Tagged do
5      Determine DepNodes for n
6      AffectedNodes = AffectedNodes ⋃ DepNodes
7  end
8  while AffectedNodes ≠ ϕ do
9      for each node p ∈ AffectedNodes do
10         Add all test cases that execute p to T_SEL
11     end
12 end
```

5 Regression test selection using ESDG model

For selecting regression test cases, we first construct the ESDG model. Next, we construct the forward slices on M_u to compute the set of nodes that are affected due to modification in ESDG model. For this, we used the two-pass graph reachability algorithm [13] and in this algorithm, each marked node in the ESDG model that are tagged during *Update ESDG model* step is taken as the slicing criterion. The union of the computed slices is taken to identify the set of nodes that are affected by modifications. The steps to compute the affected nodes is similar to that in our earlier work on regression test case prioritization [32].

Algorithm 1 shows the pseudo code to select regression test cases by constructing slices on ESDG model and we named this algorithm as *ESDGMSlice*. *ESDGMSlice* takes M_u and *Tagged* as input and produces the selected set of regression test cases denoted by T_{SEL} as the output. For each node n in *Tagged*, *ESDGMSlice* determines the set of dependent nodes denoted by $DepNodes$. The dependency may be due to data and control dependencies or due to object relations such as inheritance, association or aggregation and is computed by graph reachability analysis. *ESDGMSlice* computes the set of all affected nodes as the union of all $DepNodes$ for each node n in *Tagged* and is denoted by *AffectedNodes*. In Algorithm 1, the steps to compute the set of all affected nodes is given in lines 2 to 6.

In our approach, after the identification of all affected nodes, we select the test cases which cover one or more of these affected nodes. This is done by traversing the ESDG model and visiting each node in *AffectedNodes*.

6 Experimental study

We developed a prototype tool to implement I-RTS methodology and is named as `I-ReTEST`(Improved-Regression TEST case selector). We used I-ReTEST to measure the effectiveness of our proposed RTS technique. In the next section, we first briefly discuss our tool implementation. Next, we explain the experimental setup and subsequently, we discuss the results obtained using `I-ReTEST`.

We compared I-RTS with a traditional dependency model ESDG based approach and we have named it as ESDG-RTS [26]. For RTS using ESDG-RTS, we compute def-use pairs along all paths in the control flow model of an object-oriented program and then construct ESDG model. We select the regression test cases using I-RTS as well as using ESDG-RTS in our experimental study. In our experimentation, we make a simplifying assumption that when a program is modified no new infeasible paths are introduced due to the changes.

6.1 I-ReTEST: a prototype implementation of I-RTS

`I-ReTEST` has been developed on a Microsoft Windows XP environment. Next, we describe the open source software packages that have been used for implementing I-RTS.

Open source software packages used: to develop `I-ReTEST`, we used eclipse [33] as an IDE and ANTLR [34] as the parser generator. We adapted ANTLR grammar file for Java language [35] and Graphviz [36] is used to graphically visualize the constructed ESDG model. We have used Java Swing to develop the user interface part of `I-ReTEST`.

Components of `I-ReTEST`: Fig. 2 shows a schematic representation of `I-ReTEST`. The components of `I-ReTEST` are *ESDG Model Constructor*, *Test Coverage Generator*, *Model Marker*, *Model Updater*, and *Test Case Selector*. In Fig. 2, a rectangle represents a component and an ellipse represents the data of `I-ReTEST`. In `I-ReTEST`, the output of one component is used as an input to an another component. The role of different components of `I-ReTEST` are described as follows.

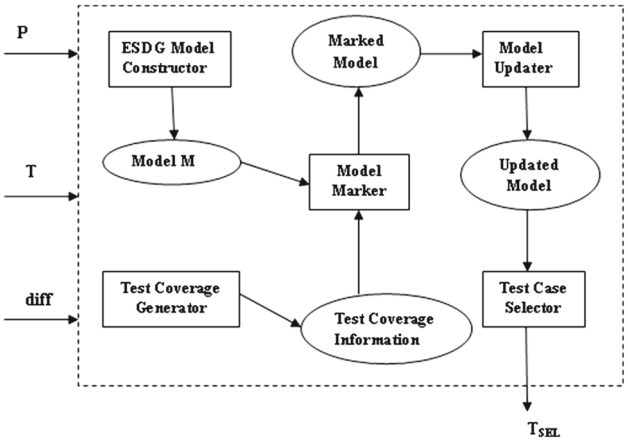

Fig. 2 Schematic representation of I-ReTEST

– *ESDG Model Constructor* In [15], we developed a tool named I-Slicer to construct slices on ESDG Model. I-Slicer has different components such as *ICCFG Model Constructor*, *Infeasible Path Detector*, *Def-Use Analyzer*, *ESDG Model Constructor*, and *Slicer*.
The *ICCFG Model Constructor* component of I-Slicer takes the Java program as input and construct the ICCFG model of the program. This component of `I-ReTEST` is developed in Java using ANTLR tool [34]. The component *Infeasible Path Detector* detects the infeasible paths in the ICCFG model of a program using the approach as proposed by Ngo et al. [17]. The *Def-Use Analyzer* component performs def-use analysis on ICCFG model. Then to compute the data flow information on ICCFG model, we use an existing data flow information algorithm as in [37]. For this, we first compute the reaching definitions of each node along all paths to that node and with the given set of reaching definitions, we determine the set of def-use pairs along all feasible paths of the ICCFG model.
The *ESDG model constructor* constructs the ESDG model by using the def-use pairs computed by the *Def-Use Analyzer* and by performing control dependence analysis on the ICCFG model. The *Slicer* component of I-Slicer constructs slices using ESDG model based on specific slicing criterion using two-pass graph reachability algorithm as proposed by Horwitz et al. [13] and results in constructing precise slices as the def-use pairs are considered only along feasible paths of the ICCFG model. The ESDG Model Constructor of I-ReTEST is same as that of I-Slicer [15].
– *Test Coverage Generator* The *Test Coverage Generator* component helps in generating test coverage information by executing the input program P with all tests from T. The test coverage information generated is stored in a file denoted by *Coverage*.
– *Model Marker* This component of `I-ReTEST` stores the test coverage information from *Coverage* in the ESDG model. For this, ESDG model is traversed to find out the corresponding node(s) modeling each program statement s in P and *Model Marker* stores all the test cases that execute the nodes corresponding to s.
– *Model Updater* This component takes *diff* file and updates the ESDG model. The updated model includes all changes that are made to P.
– *Test Case Selector* This component of `I-ReTEST` determines the affected nodes in the ESDG model and then selects all the test cases that cover one or more affected nodes in the ESDG model using Algorithm 1.

6.2 Experimental setup

We discuss the specific experimental setup made and the experimentation carried out by us using `I-ReTEST` in this section. We considered seven Java programs in our experimentation. Among these seven programs, five programs were developed by students and two were open source programs. The student programs were developed by undergraduate students of IIT as a part of their laboratory assignment in software engineering course and were developed by a group of two students. The programs were Super market Automation Software (*SAS*), Time Management Software (TMS), Book shop Automation System (BAS), Restaurant Automation System (RAS), and Word Processsing Software (WPS). The two open source software were JlibSys and JHotDraw. These two systems were downloaded from [38]. The summary of program features used in our experimentation is given in Table 1. In Table 1, columns 1 and 2 represent the program names and sources of the programs respectively and the total number of classes in each program is given in column 3. Column 4 represents the total number of methods in each program and the total number of mutants in each program is given in column 5.

To determine the bug detection effectiveness in each program, we injected faults. For this, we used a fault injec-

Table 1 Summary of program features used in experimentation

Program name	Source	Number of classes	Number of methods	Number of mutants
SAS	Student	9	57	43
TMS	Student	7	16	68
BAS	Student	8	34	82
RAS	Student	11	86	76
WPS	Student	17	91	128
JlibSys	Open source	28	229	89
JHotDraw	Open source	190	2087	163

Table 2 Summary of RTS results

Study subjects		% Tests selected		% Reduc.	% Detected faults	
Name	Test cases	ESDG-RTS	I-RTS		ESDG-RTS	I-RTS
SAS	39	61.5	53.8	12.5	100	100
TMS	28	64.2	57.1	11	100	100
BAS	42	42.8	40.4	5.6	100	100
RAS	38	63.1	55.2	12.5	100	100
WPS	56	55.3	50	9.5	100	100
JlibSys	116	40.5	35.3	12.8	100	100
JHotDraw	724	52.7	44.8	14.9	100	100

tion tool MuClipse [39]. MuClipse is an Eclipse plug-in of MuJava mutation system [40]. MuClipse provides an API to generate mutants. The types of mutation operators provided by MuClipse include both the class level as well as the traditional mutation operators.

6.3 Results and discussion

We provide a discussion on the results obtained during our experimental studies in this section. Our experimental results is summarized in Table 2. In Table 2, the programs and their test cases are represented in columns 1 and 2 respectively. Column 3 and 4 show the test cases selected in percentage by using ESDG-RTS and I-RTS respectively and the percentage reduction in the size of test suite using I-RTS as compared to ESDG-RTS is given in column 5. In Table 2, column 6 and 7 represent the percentage of faults detected out of the injected faults for ESDG-RTS and I-RTS, respectively.

It can be seen from the results presented in Table 2 that percentage of regression tests selected by I-RTS is lower as compared to ESDG-RTS for all the considered programs. The reason for this may be that I-RTS helps in reducing the statements which do not have dependencies on the slicing criterion during construction of slices unlike constructing slicing using ESDG-RTS. It can also be observed that for the program BAS the percentage reduction of test cases is only 5.6 %, where as for JHotDraw it is 14.9 %. This indicates that the percentage reduction in the test suite size depends on the slice size. It can also be observed from the experimental results that in all cases, both ESDG-RTS and I-RTS detected 100 % of the seeded faults.

It can be computed from Table 2 that I-RTS results in reduction in the test suite size on an average by 11.25 % as compared to ESDG-RTS, without affecting the fault-revealing effectiveness.

7 Comparison with related work

RTS techniques based on dependency models construct slices to identify the affected nodes and select test cases which cover one or more affected nodes [41]. However, during the dependency model construction, def-use pairs are computed along all possible paths in its control flow graph model and thus leads to inclusion of def-use pairs computed along infeasible paths. Hence, constructing static slicing on dependency model to select regression test cases may produce imprecise results [41]. But our regression test selection approach overcomes this problem by using slices with increased precision.

8 Conclusion

We propose an RTS technique using an improved slicing technique on a more accurate dependency model of object-oriented programs. The accurate dependency model is constructed by computing dependencies existing only along feasible paths. In our approach, after a modification is made to a program, we first determine the affected portion of a program. This is computed by considering data and control dependencies as well as dependencies arising due to various object relations. For this, we construct forward slices on our constructed dependency model ESDG and then select test cases which cover affected nodes. The effectiveness of our proposed RTS technique is verified using seven Java programs. The experimental studies show that on an average I-RTS selects 11.25 % less test cases as compared to a related approach, while not sacrificing the quality of testing. Our constructed accurate dependency model can also be used for prioritization of regression test cases.

References

1. Kapfhammer G (2004) The Computer Science Handbook, Chapter Software testing. CRC Press, Boca Raton
2. Leung H, White L (1989) Insights into regression testing. In: Proc. of the conference on software maintenance, pp 60–69
3. Ball T (1998) On the limit of control flow analysis for regression test selection. In: Proc. of the acm international symposium on software testing and analysis, pp. 134–142
4. Harrold M, Jones J, Li T, Liang D, Orso A (2001) Regression test selection for Java software. In: Proc. of the 16th ACM SIGPLAN

Conference on object-oriented programming, systems, languages and applications, pp 312–326
5. Rothermel G, Harrold M (1996) Analyzing regression test selection techniques. IEEE Trans Softw Eng 22(8):529–551
6. Rothermel G, Harrold M (1997) A safe, efficient regression test selection technique. ACM Trans Softw Eng Methodol 6(2):173–210
7. Rothermel G, Harrold M, Dedhia J (2000) Regression test selection for C++ software. J Softw Test Verif Reliab 10(6):77–109
8. Rothermel G, Harrold MJ (1994) Selecting regression tests for object-oriented software. In: Proc. of the Intl. Conf. on software maintenance, Victoria, pp 14–25
9. Zheng J, Robinson B, Williams L, Smiley K (2006) Applying regression test selection for cots-based applications. In: Proc. of the 28th international conference on software engineering (ICSE'06), pp 512–522
10. Farooq Q, Zohaib M, Iqbal Z, Nadeem A, Malik ZI (2007) An approach for selective state machine based regression testing. In: Proc. of the ACM 3rd international workshop on advances in model-based testing, pp 44–52
11. Briand L, Labiche Y, He S (2009) Automating regression test selection based on uml designs. J Inf Softw Technol 16–30
12. Naslavsky L, Ziv H, Richardson DJ (2009) A model-based regression test selection technique. In: Proc. of 25th IEEE international conference on software maintenance, Edmonton
13. Horwitz S, Reps T, Binkley D (1990) Interprocedural slicing using dependence graphs. Trans Program Lang Syst 12(1):26–60
14. Tip F (1994) A survey of program slicing techniques. Technical Report: CSR9438, CWI (Centre for Mathematics and Computer Science), Amsterdam
15. Panigrahi C, Mall R (2013) Slicing of object-oriented programs with improved precision. Technical Report, Department of Computer Science and Engineering, IIT Kharagpur
16. Hedley D, Hennell MA (1985) The causes and effects of infeasible paths in computer program. In: Proc. of 8th international conference on software engineering (Cat. No.85CH2139-4), London, pp 28–30
17. Ngo MN, Tan HBK (2007) Detecting large number of infeasible paths through recognizing their patterns. In: Proc. of ESEC/FSE 07, Cavtat near Dubrovnik, Croatia, pp 102–112
18. Young M, Taylor RN (1988) Combining static concurrency analysis with symbolic execution. IEEE Trans Softw Eng 14:1499–1511
19. Coward PD (1991) Symbolic execution and testing. Inf Softw Technol 33(1):53–64
20. Goldberg A, Wang TC, Zimmerman D (1994) Applications of feasible path analysis to program testing. In: Proc. of ISSTA, Seattle, pp 17–19
21. Bodik R, Gupta R, Soffa ML (1997) Refining data flow information using infeasible paths. In: Proc. of 6th European software engineering conference held jointly with the 5th ACM SIGSOFT symposium on the foundations of software engineering (ESEC/FSE '97), vol 22, pp 361–377
22. Zhang J, Wang X (2001) A constraint solver and its application to path feasibility analysis. Int J Softw Eng Knowl Eng 11:139–156
23. Malevris N, Yates DF, Veevers A (1990) Predictive metric for likely feasibility of program paths. Inf Softw Technol 32:115–118
24. Forgacs I, Bertolino A (1997) Feasible test path selection by principal slicing. In: Proc. of ESEC/FSE '97, Zurich
25. Bueno PMS, Jino M (2000) Identification of potentially infeasible program paths by monitoring the search for test data. In: Proc. of ASE, Grenoble, France, pp 11–15
26. Larsen L, Harrold M (1996) Slicing object-oriented software. In: Proc. of 18th IEEE international conference on software engineering
27. White L, Abdullah K (1997) A firewall approach for regression testing of object-oriented software. In: Proc. of 10th annual software quality week
28. Kung D, Gao J, Hsia P, Toyoshima Y, Chen C (1997) Firewall regression testing and software maintenance of object-oriented systems. J Object-Oriented Program
29. Jang YK, Munro M, Kwon YR (2001) An improved method of selecting regression tests for c++ programs. J Softw Maint Res Pract 13:331–350
30. Martins E, Vieira VG (2005) Regression test selection for testable classes. Lect Notes Comput Sci 34(63):453–470
31. Panigrahi C, Mall R (2013) An approach to prioritize the regression test cases of object-oriented programs. J CSI Trans ICT 1(2):159–173
32. Panigrahi C, Mall R (2013) A Heuristic-based regression test case prioritization approach for object-oriented programs. J Innov Syst Softw Eng (ISSE) NASA J 10(3):155–163
33. Eclipse (last visited March 21, 2014). Available http://www.eclipse.org/
34. Antlr parser generator. (last visited March 21, 2014). Available: http://www.antlr.org/
35. Wisniewski S, Antlr java grammar. (last visited March 21, 2014). Available: http://www.antlr.org/grammar/list
36. Graphviz (last visited March 21, 2014). Available: http://www.graphviz.org/
37. Pande H, Landi W, Ryder BG (1994) Interprocedural def-use associations in C programs. IEEE Trans Softw Eng 20(5):385–403
38. Open source website. (last visited October 30, 2015). Available: http://sourceforge.net/
39. An open source mutation testing plug-in for eclipse. (last visited March 21, 2014). Available: http://muclipse.sourceforge.net/
40. Ma Y-S, Offutt J, Kwon Y-R (2005) Mujava: An automated class mutation system. Journal of Software Testing, Verification and Reliability 15(2):97–133
41. Bates S, Horwitz S (1993) Incremental program testing using program dependence graphs. In: Conf. Record of the 20th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press, Charleston, South California, pp. 384–396