Regression Slicing and Its Use in Regression Testing*

István Forgács, Ákos Hajnal and Éva Takács
Computer and Automation Institute
Hungarian Academy of Sciences
1111 Kende u. 13-17. Budapest, Hungary
{forgacs, ahajnal, eva.takacs}@sztaki.hu

Abstract

In this paper we introduce an improved regression testing method that determines the test cases that must be rerun to fulfil a given testing criterion. The key part of our method is regression slicing. A regression slice contains the statements whose state may be changed in the modified program with respect to the original for a given test case.

1. Introduction

Program maintenance is the most expensive component of the entire cost of the software development. Besides it is the most dangerous as well. Analyzing the world's most expensive program errors the top three disasters were caused by a change to exactly one line of code. One necessary but expensive maintenance task is regression testing performed on modified software to provide confidence that modified programs behave as intended.

To reduce the cost of regression testing, *incremental* methods rerun only test cases for which the modified program may result in different output compared to the original. These test cases are referred to as *modification revealing* tests. If a method selects all the modification revealing tests, then this method is *safe* [11]. Most of the current methods are developed to be safe though some of them are only almost safe. By applying safe methods we can select those test cases that may reveal faults, while the remaining test cases surely would not reveal any fault, except run time errors, speed problem, external environment problem, etc. This type of errors (problems) is considered neither in our paper nor in previous work in the literature. Note that a safe technique is safe only if we assume that the original test set T is correct, i.e., when any $t \in T$ was executed, the original program P halted with a correct output, and if *obsolete* test cases are removed from T.[1] A more detailed discussion of the regression test selection and a deep analysis of current methods is in [11]. We rely on this survey, thus the evaluation of different methods mentioned in our paper is based on this work.

Those test cases that traverse modified code are called *modification traversing*. The selection of all modification traversing test cases is also safe, but selects some non-modification revealing tests.

One type of current incremental methods [2, 3, 8, 10] requires the reexecution of all modification traversing test cases. These methods ignore which testing criterion was used during the development phase. Instead, after executing the selected test cases, new tests for uncovered program components corresponding to the selected criterion are created. Other methods [5, 6] identify those program components that must be retested to satisfy some testing criterion.

Here we introduce *regression slicing* that is a more suitable means of regression testing. Intuitively, a regression slice contains all the statements whose "state"[2] may be different in the original and the modified programs with respect to a given test case. Regression slicing properly expresses the essence of program modifications. Regression slicing uses as much dynamic information (corresponding to the original program and a test case) as possible. In this way we can directly compute modification revealing test cases instead of modification traversing tests.

Former methods such as [2, 3] fail to account all the test cases that should be reexecuted while other methods select superfluous test cases ([8, 10]). Therefore, former methods are either not safe or not precise according to the framework of Rothermel and Harrold ([11]). On the contrary, regression slicing leads to safe and more precise test set that should be reexecuted. Additionally, regression slicing can also be used to eliminate superfluous test cases. It can also be applied to select an optimal execution order of the selected test cases by which other test cases may be omitted.

[0] Research supported by Hungarian National Foundation, grant 023307

[1] Test t is obsolete for program P' if and only if t either specifies an input to P' that, according to the specification, is invalid for P', or t specifies an invalid input-output relation for p'.

[2] See the precise definition in the next section.

However, regression test case minimization is not considered here. In this paper we give only the method how to derive regression slices considering any type of program modifications.

The next section presents the necessary background. In Section 3 we introduce the united dependence graph that contains both static and dynamic dependences of both the original and the modified programs. In Section 4 regression slicing is introduced and we present a method to determine regression slices. Concluding remarks are in the last section.

2. Background

In this paper we restrict our analysis to intraprocedural case, therefore, the words program, procedure and module can be used interchangeably. Any program P can be represented by a control flow graph $G = (s, N, A)$, where s is a start node N contains nodes representing basic blocks and A is the set of edges representing the possible flow of control. A program path is a sequence of nodes that were traversed for a specific input.

Considering a module P, the use of a variable v in an instruction u and a definition for v in an instruction d form a *du pair* if the value of v defined in d can potentially be used in u. For this case we say that u is *directly data dependent* on d. An instruction I_p is *control dependent* on an instruction I_r if (1) I_r has two exists. (2) Following one of the exits from I_r always results in I_p being executed, while taking the other exit may result in I_p not being executed.

Similarly to static dependence we can define dynamic dependence as well. There is dynamic data dependence between a definition d for variable v and a use u of the same variable, if v assigned in d is actually used in u for a given test case t. A node n is dynamically control dependent on a predicate node p, if n is control dependent on p and both n and p are executed for t.

We refer to the set of statements executed for a test case t as *execution slice* of the program with respect to t [2]. The corresponding set of execution slices for T is denoted by H, an element of H is h.

We can relate a *state* to each program statement. The state of a program statement is valid after the statement has been executed. First, a state is a function mapping variable names to their current values [4], i.e., $I_0 = S(I)$, where I is the variable and I_0 is its value. We simple say that the state of I is I_0. The state of an expression is computed from the state of the components of the expression, for example $S(x + y) = S(x) + S(y)$. The state of an assignment statement s includes the states (i.e., the values) of all the variables that are used or defined in s. For example, if the statement s is $x = a + b$, then $S(s) = \{S(x) = S(a) + S(b), S(a), S(b)\}$. The state of an output statement Out(x * y) is S(Out) = {S(x) * S(y)}. Finally, the state of a boolean expression is either $true$ or $false$, for example, $S(a > b)$ is $true$ if $S(a) > S(b)$ and $false$ if $S(a) \leq S(b)$. We assign a special state "non-existing" to a statement that is not executed while running P. This state is different from any other state of executed statements.

Note that the idea of state is necessary and cannot be substituted by simply computing the value of expressions at program statements. In this latter case the state of the following two statements `s: x = a + b` and `s': y = a + b` would be the same. However, if s' contains a faulty modification, then the error may be revealed by a test traversing s'. Since our method is based on possible state modification, therefore we have to differentiate the state of s and s'.

Consider a program P, a program statement $s \in P$ and a test case t. Assume that the corresponding program statement in the modified program P' is s' (s and s' may be identical). The state of a program statement s' in P' *may be modified* wrt P for t if $S(s_i)$ and $S(s'_i)$ may be different for any i, where s_i (s'_i) means the ith occurrence of s (s') in $h(t)$ ($h'(t)$). Note that t is executed only for P.

3. United dependence graph

The *static program dependence graph (PDG)* for module P is a digraph [9]. The nodes of the PDG represent individual program instructions in P; in addition, there is an "enter" node. A control edge in the PDG represents control dependence, while flow edges represent direct data dependences. The source of a control edge is the enter node or a predicate node.

Similarly to the static PDG we can construct the dynamic dependence graph (DDG) that contains only nodes that are in the execution slice for t. The DDG is a subgraph of the PDG. The dynamic dependence graph contains edges representing dynamic control and dynamic data dependences obtained by executing the program for t. Dynamic dependences are determined during the execution of P on input t. Note that this graph is much smaller than the one defined by Agrawal and Horgan [1] and corresponds their Approach 2.

In this section we introduce the *united dependence graph* that is the most suitable representation for our purpose. United dependence graphs (UDG(P, t)) include all the information on the DDG and the PDG. Since the DDG is a subgraph of the PDG we should only label both the nodes and edges that are included in the DDG. Accordingly, the dynamic edges and executed nodes are labeled (we use letter d for dynamic arcs and h for executed nodes). Empty $else$ branches are also considered such that a node related to an empty statement (for the $else$ branch) is inserted into the UDG. The united dependence graph corresponding to procedure `Example` below is shown in Figure 1.

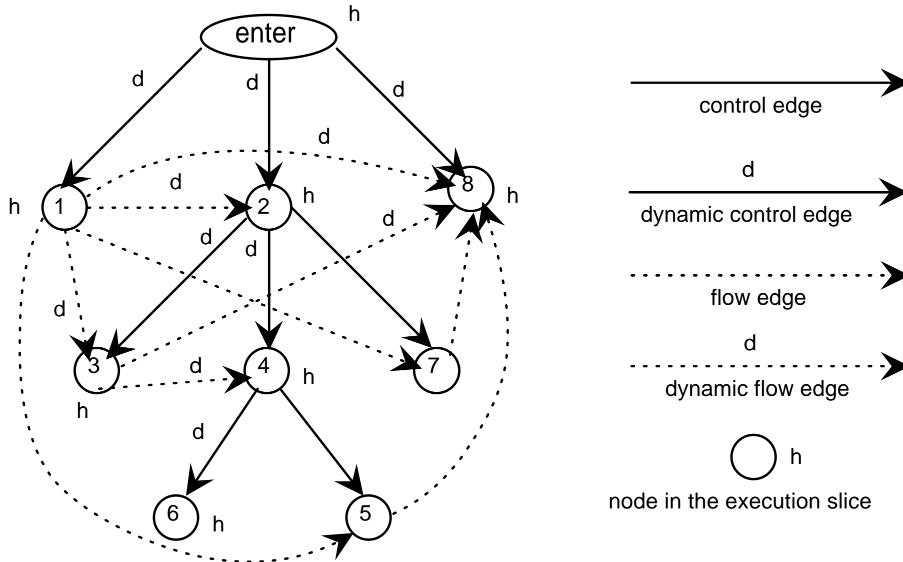

Figure 1. UDG(Example,t)

```
  procedure Example
1   read(x,y)
2   if x > 10 then
3     z = x + y
4     if z < 0 then
5       x = x + 1
6     else
      endif
    else
7     x = x - 1
    endif
8   write(x, z)
  endprocedure
```

t: (x = 11, y = 0); h = $\langle 1, 2, 3, 4, 6, 8 \rangle$

Our method is based on a modified UDG including information on the modified program P'. However, it is not the precise UDG for P', since P' is not executed, thus dynamic information cannot be obtained. Yet, we may have some dynamic information about new statements without any real execution. In fact, if a new statement s is inserted into a branch that has (has not) been executed, and there is no other modification in P', then s is (is not) included in the execution slice.

From here onwards, the UDG for P' without executing t is referred to as UDG(P'). All the nodes corresponding to new and modified statements are marked as "modified". Note that we can compare P and P' by applying "sequence matching" algorithms [7, 12] that identifies the *corresponding* (i.e., identical and modified) statements between P and P'. It also identifies new and deleted statements. The construction of UDG(P') starts with the creation of the PDG for P'. Any label "h" in the UDG(P, t) is changed to label "e" in the UDG(P'). Any edge labeled "d" keeps its label. The set of edges whose label is "e" is denoted by e.

Any new node for which the related statement is inserted into a branch that has been executed in P for t is labeled "e". Other new nodes have no label. Now assume that a new predicate is inserted into P' (the related node is denoted by p). Consider the case when a formerly executed node n (whose label is "h") is (directly or indirectly) control dependent on p. This node n now gets a label "e" though the related node may not be executed in P'. Unlabelled control dependent nodes remain unlabelled by inserting a new predicate p in the same way. If a new block beginning with a predicate (e.g. if ... then ... else .... endif) is inserted into the program such that the predicate would be surely executed for t, then the predicate and all new statements in the block are labeled "e". If we insert the above block in a branch that has not been executed for P and t, then none of the statements have labels.

We can see that a statement for which a related node has a label "e" is not surely in h', while an unlabeled node may correspond to a statement that is in the execution slice for P' and t. The cause is that the meaning of "e" is different from "h", i.e., label "e" for statement $s \in P'$ means that without executing t for P' we know that s would be executed if there were no other changes in the program (except the inclusion of s if it is new). Note that the only exception is a deletion of a predicate, see below.

A control edge is labeled "d" if its both connected nodes have a label "e". Consider a du pair of a variable v. The related flow edge is a dynamic flow edge and has a label "d" if (1) both nodes wrt the definition d and the use u are labeled "e" and (2) there is no definition for v between d and u wrt the execution slice in the UDG(P, t). A newly inserted definition is considered in the same way if and only

if it is surely executed in P' for t. In practice, a dynamic flow edge in the UDG(P, t) is also a dynamic flow edge in the UDG(P'), and only the newly inserted assignment statements have to be analyzed[3].

The modified Example is shown below while the related UDG is depicted in Figure 2.

```
  procedure Example'
1    read(x,y)
2    if x > 10 then
3      z = x + y
4      if z < 0 then
5        x = x + 1
6      else
       endif
     else
7      x = x - 1
7a     z = 1                (* new *)
     endif
8a   if z >= 1 then         (* new *)
8      write(x, z)
8b   else
     endif
  endprocedure
```

$t : (x = 11, y = 0)$; e = $\{1, 2, 3, 4, 6, 8a, 8\}$

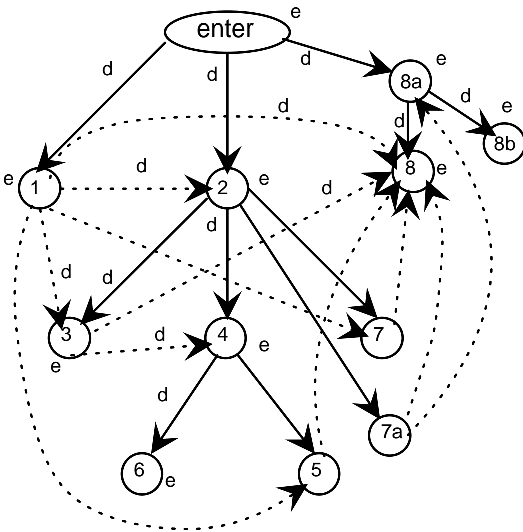

Figure 2. UDG(Example')

Now consider deleted statements. If we delete a predicate p, then let us investigate the nodes that are (directly) control dependent on p and that are not in the execution slice in P. Since in P' these nodes are surely executed their label becomes "e". Though the corresponding statements are not modified, yet we mark them as "modified" since their state was non-existing that has been changed (similarly to new statements).

The effect of deletion of an assignment statement `x = ...` (which is in the execution slice of P) is the same as though the right-hand-side of this statement had been changed to x. Therefore we insert a "virtual" statement `x = x` into P' replacing the deleted statements. The related node in the UDG(P') has a label "e" and marked as "modified". After the regression slice has been determined all virtual statements are deleted from it. Any other type of deleted statements is simple ignored.

Finally, consider the case where an assignment, `x = exp1` has been changed to `y = exp2`. The effect of this modification can be expressed by deletion of `x = exp1` and insertion of `y = exp2`.

4. Regression slicing

In this section we introduce regression slicing by which we can determine a safe regression test set in a simple way. In addition, by applying regression slicing we can divide the original test set into three parts, i.e., the test cases that do not need to be reexecuted; the test cases that should be rerun, and tests that can be removed from the original test set. However, the exact method of this division is out of the scope of this paper.

Definition. Consider the original and the modified programs denoted by P and P', respectively, and a test case $t \in T$. The slicing criterion S_c is a triple, i.e., $S_c = (P, P', t)$. The regression slice with respect to the slicing criterion S_c contains those statements $s \in P'$ whose state may be modified in P' wrt P for t.

Regression slicing involves both static and dynamic elements. It is static since we do not execute t for P', i.e., the execution slice $h'(t)$ is unknown. Therefore, we should determine the statements in the regression slice by considering static dependence information. However, regression slicing has dynamic elements as well. The cause is that t is given, thus we know a subset of execution slices. In addition, we use dynamic dependences where it is possible.

Former dynamic slice-based work on regression testing [2] uses different type of slices. Really, the *relevant* slice contains those statements that, if modified may alter the program output for t. In this way a relevant slice for an unchanged module contains elements, while the regression slice is empty. In addition, a relevant slice is a backward slice with a slicing criterion contains the execution slice, a location and a variable. As a consequence relevant slices may lead to unsafe regression testing as reported in [11].

Now we present a method for regression slicing, i.e., we determine those statements in P' whose state may be different from the related statements in P by executing t (for P). To make our discussion clearer we use a simple program in Figure 3 which determines the different types of a triangles

[3] Though in this way the UDG(P') may contain superfluous dynamic edges, the method described in the next section remains precise. The reason is that a modified use is inserted into the regression slice directly. On the other hand if there is a new redefinition along the execution slice, then the state of the use is modified obtaining the same result as if the superfluous dynamic flow edge was considered.

```
 1  read(a,b,c)                (* assumed the read in an increasing order *)
 2  class = scalene            (* new in Case 1, unmodified in other cases *)
 3  if a + b < c then          (* modified in Case 2 and Case 3 *)
 4    class = non-triangle
 5  if a = b or b = c then
 6    class = isosceles
 7  if a = b then              (* modified in Case 4 *)
 8    if b = c then
 9      class = equilateral
10  if c*c = a*a + b*b then
11    class = right
12  case class of
13    non-triangle:  area = 0
14    equilateral :  area = a*a*sqrt(3)/4
      isosceles,
15    scalene     :  s = (a+b+c)/2
16                   area = sqrt(s * (s-a) * (s-b) * (s-c)
    endcase
17  write(class, area)
```

Figure 3. The sample program to demonstrate regression slicing.

and computes their area.

Let us determine the *regression slice for the triple* (P, P', t), which we simply refer to as regression slice. Firstly, any modified or new statement for which the related node has a label "e" is inserted into the slice. These are the modified statements that are assumed to be executed for t in P'. Because these statements (including the virtual statements) are the source of state modifications of affected statements, we start from them. Then we collect appropriate statements gradually based on the UDG(P').

If a statement s is neither new nor modified, then it is included in the regression slice if it is directly or indirectly influenced by any new or modified statement that is in the regression slice. There are four cases when the state of a statement may be modified. These cases are detailed below.

Case 1

First, consider an assignment statement s_a that is marked as "modified". If there is a direct dynamic data dependence from s_a to another statement s (UDG(P') contains a flow edge from s_a to s), then the state of s may be modified. Hence, s should be inserted into the regression slice. Similar case happens if s_a is not marked as "modified" but it has been formerly inserted into the regression slice. If $s_a \notin e$, and s_a is an assignment statement in the regression slice, then s is inserted into the regression slice if $s \in e$ and there is a flow edge from s_a to s. The reason is that if s_a in the regression slice, then s_a may be executed, therefore, a dynamic dependence between s_a and s can occur. All other cases are investigated in Case 2, Case 3 and Case 4.

Example. Assume that $t : (a = 5, b = 6, c = 7)$, $h = \langle 1, 3, 5, 7, 10, 12, 17 \rangle$, and statement 2 is new. Since 2 is new and surely executed, then it has a label "e" and it is inserted into the regression slice. (Hence $e = \{1, 2, 3, 5, 7, 10, 12, 17\}$). Since by definition there is a dynamic flow edge from 2 to 12 in the modified program, thus statement 12 is also included in the regression slice.

Now consider a predicate p in the regression slice, and classify each statement s that are (directly) control dependent on p as s_e or s_i such that $s_e \in e$ and $s_i \notin e$.

Case 2

Consider a statement s_i. Since the outcome of p may change, s_i is control dependent on p, therefore s_i may be executed in P', i.e., its state may be changed. As a consequence, any s_i is inserted into the regression slice.

Example. Assume, that 3 is modified and the test case is as above, i.e., $t : (a = 5, b = 6, c = 7)$, $h = e = \langle 1, 2, 3, 5, 7, 10, 12, 15, 17 \rangle$. Initially, we insert 3 into the regression slice. Though statement 4 is not in the execution slice for the original program but it may be executed for the modified one, thus, according to Case 2 it should be inserted into the regression slice.

Case 3

Let us investigate non-predicate statements s_e. Since p's outcome may change, s_e may not be executed. In this way its state may be changed to non-existing and thus s_e is inserted into the regression slice. We can see that Case 2 and Case 3 are very similar except that Case 3 ignore predicates.

Example. Assume that statement 3 is modified, thus it is in the regression slice. The test case is $t : (a = 2, b = 3, c = 7)$, therefore, $h = e = \langle 1, 2, 3, 4, 5, 7, 10, 12, 13, 17 \rangle$. Because statement 4 may be excluded for t in P', thus according to Case 3 we insert statement 4 into the regression slice. Since there is direct dynamic dependence between 4 and 12, the latter is also inserted into the slice according to Case 1.

Case 4

If s_e is a predicate, we also insert s_e into the regression slice, since s_e may not be executed in P'. In addition,

s_e is marked as "transparent". The cause is that the state of control dependent statements (on s_e) whose label is "e" may only be modified (to non-existing). Those control dependent statements that are not traversed for t in P are surely not traversed in P' either, thus their state remain non-existing. Similar is the case for new statements which are inserted into non-executed branches. As a consequence, starting from a transparent predicate we insert a control dependent statement s into the regression slice if and only if $s \in e$. If s is a predicate, then s is also marked as transparent. Note that a transparent predicate may become non-transparent if its outcome may be changed. This can occur if an assignment statement that is inserted into the regression slice has an influence on s_e.

Example. Let the test case be $t : (a = 2, b = 2, c = 3)$, for which $h = \langle 1, 2, 3, 5, 6, 7, 8, 10, 12, 15, 17 \rangle$. Assume that now statement 7 is modified. Since it is executed for t, thus it is in the regression slice. Because predicate 8 is control dependent on 7, and 8 is an element of e, hence it is involved in the regression slice and mark as transparent. Though 9 is control dependent on 8, it is not inserted into the slice. The cause is that even if the outcome of 7 may change to false the state of statement 9 remains non-existing. If we select a test case $t : (a = 2, b = 2, c = 2)$ for which $h = e = \langle 1, 2, 3, 5, 6, 7, 8, 9, 10, 11, 12, 14, 17 \rangle$, we have to insert statement 9 into the slice. The cause is that if 7 is modified 9 may not be executed for t which may involve the change of the states of statement 9 (equilateral $\rightarrow$ non-existing), and transitively 12 (equilateral $\rightarrow$ isosceles).

Based on these cases and the UDG(P'), we can construct the regression slices.

5. Conclusion

In this paper we we have introduced a new method called regression slicing. A regression slice contains the statements whose state may be changed in the modified program with respect to the original for a given test. Regression slicing is a method that is perfectly fitted regression testing. With this slice we can select test cases safely. Since regression slice involves both static and dynamic elements we obtain a smaller (but safe) test set than former methods. Comparing to other methods, [3] and [2] are not safe, [8] is less precise than [10] and [10] is less precise than our method. Really, our method considers *potential dependence* due to predicates (see [2]) while the method in [10] does not. However, regression slicing also makes it possible to determine superfluous test cases or an appropriate execution order of test cases obtaining minimum regression testing effort. Therefore, we believe that our new technique can be used for the whole regression testing process. Former methods usually address the regression test selection problem only.

References

[1] H. Agrawal and J.R. Horgan. Dynamic program slicing, *SIGPLAN Notices*, 25(6): 246–256 (1990)

[2] H. Agrawal, J.R. Horgan, E.W. Krauser and S.A. London. Incremental regression testing, *Proc. of the Conf. on Software Maintenance*, 348–357, (Montreal, 1993)

[3] S. Bates and S. Horwitz. Incremental program testing using dependence graphs, *Proc. of 20th ACM Symposium on Principles of Programming Languages*, Jan. 1993

[4] J.D. Gannon, R.G. Hamlet and H.D. Mills. Theory of modules, *IEEE Trans. on Soft. Engineering*, 13(7):820–829, (1987).

[5] R. Gupta, M.J. Harrold and M.L. Soffa. Program slicing-based regression testing techniques, *Software Testing, Verification and Reliability*, 6 83–111 (1996)

[6] M.J. Harrold and M.L. Soffa. An incremental approach to unit testing during maintenance, *Proc. of Conf. on Software Maintenance*, 362–367 (1988)

[7] D.S. Hirsberg A linear space algorithm for computing maximal common subsequences, *Communication of the ACM*, 18(6) 341–343 (1975)

[8] J. Laski and W. Szermer. Identification of program modifications and its applications in software maintenance. *Proc. of the Conf. on Software Maintenance* Nov. 1992

[9] K.J. Ottenstein and L.M. Ottenstein. The program dependence graph in a software development environment, *ACM SIGPLAN Notices*, 19(5):177-184, (1984)

[10] G. Rothermel and M.J. Harrold. Selecting tests and identifying test coverage requirements for modified software, *Proc. of ISSTA'94*, 169–184 (Seattle 1994)

[11] G. Rothermel and M.J. Harrold. Analyzing regression test selection techniques, *IEEE Trans. Software Eng.*, 22(8), 529-551, Aug. 1996.

[12] W. Yang. Identifying syntactic differences between two programs, *Software–Practice and Experience*, 21(7): 739–755, (1991)