Conditioned Slicing Supports Partition Testing

Rob Hierons and Mark Harman
Brunel University
Uxbridge
Middlesex
UB8 3PH
United Kingdom

Chris Fox
King's College
University of London
Strand
London WC2R 2LS
United Kingdom

Lahcen Ouarbya and Mohammed Daoudi
Goldsmiths College
University of London
New Cross
London SE14 6NW
United Kingdom

Abstract

This paper describes the use of conditioned slicing to assist partition testing, illustrating this with a case study. The paper shows how a conditioned slicing tool can be used to provide confidence in the uniformity hypothesis for correct programs, to aid fault detection in incorrect programs and to highlight special cases.
Keywords: **Fault-detection; Conditoned slicing; Partition testing**

1 Introduction

When generating tests from a specification it is common to apply partition analysis: a partition $P = \{D_1, \ldots, D_n\}$ of the input domain D, is produced. This partition has the property that the behaviour of the specification is uniform (and thus relatively simple) on each D_i. Faults may either affect the behaviour within a subdomain (computation faults) or affect the boundaries of the subdomains (domain faults).

Computation faults are detected by choosing one or more test cases from each subdomain. Domain faults are detected by testing around subdomain boundaries [3, 9]. Suppose an implementation under test I is tested on the basis of partition P. If I is *uniform* on each of the subdomains of P, it is likely that faults will be detected by a test set based on P. This form of assumption, that the behaviour is uniform on each D_i, is the 'uniformity hypothesis' of partition testing.

Conditioned slicing [1] is a technique for identifying those statements and predicates which contribute to the computation of a selected set of variables when some chosen condition is satisfied. The technique has previously been used in program comprehension [5, 6] and re-engineering [2]. Further details about conditioned slicing are given in Section 2.

This paper shows how conditioned slicing using the ConSIT slicing tool [4] can be used to assist partition-based testing. Specifically it will be shown how conditioned slicing:

1. Provides confidence in uniformity holding on a subdomain D_i from P.

2. Suggests the existence of faults associated with subdomain $D_i \in P$, providing information that can be used to either refine P(domain faults) or direct effort towards D_i(computation faults).

3. Detects the existence of erroneous special cases.

These three topics are addressed by Sections 3, 4 and 5 respectively.

2 Conditioned Slicing

A static slice of a program P with respect to Weiser's [8] slicing criterion $C(V, n)$ is obtained by deleting, from the original program, all statements which do not contribute to the computation of any variable in V(subset of variables in P) at the program point n for all possible executions. However, there are cases when only a subset of the program executions is of interest. In this case static slices might include statements not involved in the executions of interest.

Conditioned slicing [1] is a technique to compute program slices with respect to a subset of program executions. The conditioned slicing criterion $C(V, n, F)$ is an extension of the static-slicing criterion introduced by Weiser. It allows the programmer to specify not only the variables of interest, but also the initial condition of interest. This is done by using a quantifier-free first order logic formula F, which is a mapping from the set of initial states to the booleans.

```
1-  main() {
2-  int mark;
3-  char grade, *result;
4-  scanf("%d",&mark);

5-  if (mark<35) { result = "fail" ; grade = 'F'; }
6-  else { result = "pass";
7-          if(mark<40) grade = 'E';
8-          else { if(mark<50) grade = 'D';
9-                  else { if(mark<60) grade= 'C';
10-                         else { if(mark<70) grade = 'B';
11-                                else grade = 'A'; }}}}}
```

Figure 1: Student's final year result program P

```
if (mark<35);
else {if (mark<40) grade = 'E';
      else {if(mark<50) grade = 'D';
            else {if(mark<60) grade = 'C';
                  else {if(mark<70) grade = 'B';
                        else grade = 'A'; }}}}
```

Static slice of P with respect to C(`grade`, 11)

```
grade = 'E';
```

Conditioned slice of P with respect to C(`grade`, 11, C_0)

Figure 2: Traditional slice and Conditioned slice of Student's Result Program

In program P in Figure 1 the student's overall `mark` is input and the program determines the `result` (fail, pass) and the `grade`. Suppose that `grade` is the variable of interest and the program execution of interest is defined by the following condition:

$$C_0 \equiv (\ \texttt{mark >= 35} \wedge \texttt{mark < 40});$$

then the static slice with respect to the slicing criterion C(`grade`, 11), shown in the left-hand column of Figure 2, contains unnecessary statements. However adding the condition C_0 to the static slicing criterion, condition slicing eliminates all statements which can not affect the slicing criterion if the input satisfies this condition. The fragment in the right-hand column of Figure 2 yields the conditioned slice of the original program P with respect to slicing criterion C(`grade`, 11, C_0).

```
main() {
int age, blind, widow, married, income;
int personal, tax, t, pc10; char code;

scanf("%d",&age);
scanf("%d",&blind);
scanf("%d",&married);
scanf("%d",&widow);
scanf("%d",&income);

if (age>=75) personal = 5980;
else if (age>=65) personal = 5720;
else personal = 4335;

if ((age>=65) && income>16800) {
  t = personal - ((income-16800)/2);
  if (t>4335) personal = t;
  else personal = 4335;}

if (blind) personal = personal + 1380;

if (married && age>=75) pc10 = 6692;
else if (married && (age>=65)) pc10 = 6625;
else if (married || widow) pc10 = 3470;
else pc10 = 1500;
if (married && age>=65 && income>16800) {
  t = pc10-((income-16800)/2);
  if (t>3470) pc10 = t; else pc10 = 3470;}

if (income<=personal) tax = 0;
else {
  income = income-personal;
  if (income<=pc10) tax = income/10;
  else {
    tax = pc10/10;
    income = income-pc10;
    if (income<=28000)
      tax = ((tax+income)*23)/100;
    else {
      tax = ((tax+28000)*23)/100;
      income = income-28000;
      tax = ((tax+income)*40)/100;}}}

if (!blind && !married && age<65)
  code = 'L';
else if (!blind && age<65 && married)
  code = 'H';
else if (age>=65 && age<75 && !married && !blind)
  code = 'P';
else if (age>=65 && age<75 && married && !blind)
  code = 'V';
else code = 'T';}
```

Figure 3: UK Income Taxation Calculation Program

All examples in Sections 3, 4 and 5 will be constructed with respect to the program in Figure 3, which calculates tax codes and amounts of tax payable, including allowances for a United Kingdom citizen in the tax year April 1998 to April 1999.

3 Confidence Building with Conditioned Slicing

One of the problems associated with partition analysis is that the behaviour of the implementation under test need not be uniform on each element of the partition. Where this assumption fails, the test generated on the basis of P is likely to be insufficient. It would therefore be useful to be able to determine whether the uniformity hypothesis holds. Where it does not hold for some D_i, ideally the tester should either further divide D_i or choose more tests from D_i.

Let C_{D_i} denote the condition that the input lies in D_i. Then, if I is uniform on D_i, the conditioned slice $S(I, C_{D_i})$ is likely to be relatively simple: slicing using condition C_{D_i} should lead to much simplification [7]. Where this is the case, the tester might have greater confidence in the uniformity hypothesis holding for D_i. Consider the tax example of Figure 3 and suppose the tester chooses the subdomain defined by the condition C_1 below:

$$age \geq 75 \ \wedge \ blind \ \wedge \ 0 \leq income \leq 7360$$

For this condition and the variable `tax`, ConSIT produces the following conditioned slice.

```
tax = 0;
```

The simplicity of this conditioned slice suggests that the behaviour is uniform on this subdomain and thus that only a small number of tests are required here. Indeed, in this case, the slice is so simple that the tester can easily determine correctness.

`if (age>75) personal=5980;`
`else if (age>=65)personal=5720;`
`     personal=personal+1380;`
`if (income<=personal) tax=0;`
`else {income=income-personal;`
`      tax=income/10;}`
	`personal=5980;`
	`if (age>=75 && income==1500) personal = 0;`
	`personal = personal+1380;`
	`if (income<=personal) tax=0;`
	`else {income=income-personal;`
	`      tax=income/10;}`

Slice for C_1 Applied to First Faulty Tax Program	Slice for C_1 Applied to Second Faulty Tax Program

Figure 4: Fault-Revealing Conditioned Slices

`tax = 0;`	`personal = 5720;`
	`personal = personal + 1380;`
	`income = income - personal;`
	`tax = income/10;`
		`tax = 0;`

Slice for C_1^1 and variable `tax`	Slice for C_1^2 and variable `tax`	Slice for C_1^3 and variable `tax`

Figure 5: Conditioned Slices for refined Subdomains

4 Fault Detection with Conditioned Slicing

Suppose a fault is introduced by changing `if (age >= 75)` to `if (age > 75)`. ConSIT produces the slice in the left-hand column of Figure 4 for the subdomain defined by C_1. Here there has been far less simplification, suggesting that the behaviour may not be uniform. In particular, the conditioned slice contains `if` statements. In such situations, ConSIT can be of further assistance, by computing the simplest path conditions applicable. In this case it produces: $age = 75 \wedge income <= 7100$, $age = 75 \wedge income > 7100$, and $age > 75$.

This suggests that the subdomain denoted by C_1 should be refined to include each of the three path conditions, yielding:

1. $C_1^1 \equiv (C_1 \wedge age = 75 \wedge income <= 7100)$;
2. $C_1^2 \equiv (C_1 \wedge age = 75 \wedge income > 7100)$;
3. $C_1^3 \equiv (C_1 \wedge age > 75)$.

For these refined domains, ConSIT produces the three slices in Figure 5. Values from the subdomain denoted by C_1^2 will detect the fault.

5 Highlighting Special Cases with Conditioned Slicing

Consider now a second fault, produced by adding the following extra (malicious) code just before the line that starts `if blind`:

```
if (age >= 75 && income == 1500) personal = 0;
```

Slicing using C_1 and variable `tax` yields the fragment in the right-hand column of Figure 4. This appears not to be uniform and thus the tester might either choose to test thoroughly within the corresponding subdomain, or to analyse the slice further. Further analysis of this slice leads to two new conditions:

1. ($income = 1500$);

2. *not* ($income = 1500$).

The fault will be found by refining the subdomain, corresponding to C_1, using these two conditions and then testing from the refined domains.

Interestingly, this second fault is of a type that is usually very difficult to find using specification-based testing because the implementation contains behaviour that is *not* in the specification. Since the specification does not contain this behaviour, and the behaviour lies within the body of a subdomain, traditional specification-based testing is unlikely to find it: there is no information in the specification that indicates that the value 1500 for `income` is significant. Fortunately, conditioned slicing highlights this additional behaviour.

References

[1] Canfora, G., Cimitile, A., and De Lucia, A. Conditioned program slicing. In *Information and Software Technology Special Issue on Program Slicing*, M. Harman and K. Gallagher, Eds., vol. 40. Elsevier Science B. V., 1998, pp. 595–607.

[2] Canfora, G., Lucia, A. D., and Munro, M. An integrated environment for reuse reengineering C code. *Journal of Systems and Software 42* (1998), 153–164.

[3] Clarke, L. A., Hassell, J., and Richardson, D. J. A close look at domain testing. *IEEE Transactions on Software Engineering 8* (1982), 380–390.

[4] Danicic, S., Fox, C., Harman, M., and Hierons, R. M. ConSIT: A conditioned program slicer. In *IEEE International Conference on Software Maintenance (ICSM'00)* (San Jose, California, USA, Oct. 2000), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 216–226.

[5] De Lucia, A., Fasolino, A. R., and Munro, M. Understanding function behaviours through program slicing. In 4^{th} *IEEE International Workshop on Program Comprehension* (Berlin, Germany, Mar. 1996), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 9–18.

[6] Fox, C., Harman, M., Hierons, R. M., and Danicic, S. Backward conditioning: a new program specialisation technique and its application to program comprehension. In 9^{th} *IEEE International Workshop on Program Comprehesion (IWPC'01)* (Toronto, Canada, May 2001), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 89–97.

[7] Hierons, R. M., and Harman, M. Program analysis and test hypotheses complement. In *IEEE ICSE International Workshop on Automated Program Analysis, Testing and Verification* (Limerick, Ireland, June 2000), pp. 32–39.

[8] Weiser, M. *Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method.* PhD thesis, University of Michigan, Ann Arbor, MI, 1979.

[9] White, L. J., and Cohen, E. I. A domain strategy for computer program testing. *IEEE Transactions on Software Engineering 6* (1980), 247–257.