

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

Mohammed Daoudi and Lahcen Ouarbya
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.

1

1 Introduction

When generating tests from a specification it is common to apply partition analysis: a partition $P = \{D_1, \dots, D_n\}$ 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 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 [Clarke et al., 1982, White and Cohen, 1980]. 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 [Canfora et al., 1998a] 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 [De Lucia et al., 1996, Fox et al., 2001] and re-engineering [Canfora et al., 1998b].

This paper shows how conditioned slicing using the ConSIT slicing tool [Danicic et al., 2000] 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 .

¹This is a preliminary version of the following paper: R.M. Hierons, M. Harman, C.J. Fox, M. Daoudi, and L. Ouarbya, 2002, Conditioned slicing supports partition testing, *The Journal of Software Testing, Verification and Reliability*, **12** 1, pp. 23-28.

2. Suggests the existence of faults associated with subdomain $D_i \in P$, providing information that can be used to either refine P or direct effort towards D_i .
3. Detects the existence of erroneous special cases.

These three topics are addressed by sections 2, 3 and 4 respectively. All examples will be constructed with respect to the program in Figure 1, 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.

<pre> 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; </pre>	<pre> 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';} </pre>
---	---

Figure 1: UK Income Taxation Calculation Program

2 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 [Hierons and Harman, 2000]. Where this is the case, the tester might have greater

<pre> 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;} </pre>	<pre> personal=5980; if (age>=75 && income==1500) personal=personal-1000; personal=personal+1380; if (income<=personal) tax=0; else {income=income-personal; tax=income/10;} </pre>
Slice for C_1 Applied to First Faulty Tax Program	Slice for C_1 Applied to Second Faulty Tax Program

Figure 2: Fault-Revealing Conditioned Slices

confidence in the uniformity hypothesis holding for D_i . Consider the tax example of Figure 1 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.

3 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 2 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: $age = 75 \wedge income \leq 7100$, $age = 75 \wedge income > 7100$, 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. $C_1^1 \equiv (C_1 \wedge age = 75 \wedge income \leq 7100)$;
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 3. Values from the subdomain denoted by C_1^2 will detect the fault.

4 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`:

<code>tax = 0;</code>	<code>personal = 5720;</code> <code>personal = personal + 1380;</code> <code>income = income - personal;</code> <code>tax = income/10;</code>	<code>tax = 0;</code>
Slice for Condition C_1^1	Slice for Condition C_1^2	Slice for Condition C_1^3

Figure 3: Conditioned Slices for refined Subdomains

```
if (age >= 75 && income == 1500) personal = personal - 1000;
```

Slicing using C_1 yields the fragment in the right-hand column of Figure 2. 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. Symbolically evaluating 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

- [Canfora et al., 1998a] Canfora, G., Cimitile, A., and De Lucia, A. (1998a). Conditioned program slicing. In Harman, M. and Gallagher, K., editors, *Information and Software Technology Special Issue on Program Slicing*, volume 40, pages 595–607. Elsevier Science B. V.
- [Canfora et al., 1998b] Canfora, G., De Lucia, A., and Munro, M. (1998b). An integrated environment for reuse reengineering C code. *Journal of Systems and Software*, 42:153–164.
- [Clarke et al., 1982] Clarke, L. A., Hassell, J., and Richardson, D. J. (1982). A close look at domain testing. *IEEE Transactions on Software Engineering*, 8:380–390.
- [Danicic et al., 2000] Danicic, S., Fox, C., Harman, M., and Hierons, R. M. (2000). ConSIT: A conditioned program slicer. In *IEEE International Conference on Software Maintenance (ICSM'00)*, pages 216–226, San Jose, California, USA. IEEE Computer Society Press, Los Alamitos, California, USA.
- [De Lucia et al., 1996] De Lucia, A., Fasolino, A. R., and Munro, M. (1996). Understanding function behaviours through program slicing. In *4th IEEE Workshop on Program Comprehension*, pages 9–18, Berlin, Germany. IEEE Computer Society Press, Los Alamitos, California, USA.

- [Fox et al., 2001] Fox, C., Harman, M., Hierons, R. M., and Danicic, S. (2001). Backward conditioning: a new program specialisation technique and its application to program comprehension. In *9th IEEE International Workshop on Program Comprehension (IWPC'01)*, pages 89–97, Toronto, Canada. IEEE Computer Society Press, Los Alamitos, California, USA.
- [Hierons and Harman, 2000] Hierons, R. M. and Harman, M. (2000). Program analysis and test hypotheses complement. In *IEEE ICSE International Workshop on Automated Program Analysis, Testing and Verification*, pages 32–39, Limerick, Ireland.
- [White and Cohen, 1980] White, L. J. and Cohen, E. I. (1980). A domain strategy for computer program testing. *IEEE Transactions on Software Engineering*, 6:247–257.