Program Analysis and Test Hypotheses Complement

R. M. Hierons and M. Harman

December 3, 2010

Abstract

This paper considers ways in which program analysis and test hypotheses complement, focusing on one particular example: the uniformity hypothesis. Conditioned slicing can be used to either provide confidence in the uniformity hypothesis, identify faults, or suggesting refinements to the hypothesis. The existence of a uniformity hypothesis assists in the production of small conditioned slices which might then be analysed further. keywords: Program verification, test hypotheses, the uniformity hypothesis, program analysis, conditioned program slicing.

1 Introduction

Most approaches to program verification can be categorised as one of dynamic testing and program analysis. Dynamic testing involves exploring the behaviour of the *implementation under test (IUT)* when given particular input values. Within the verification context, program analysis involves studying the source code of the IUT in order to derive information that might either increase the confidence in the correctness of the IUT or detect faults in the IUT.

In general, it is not possible to produce a finite test set that is guaranteed to determine correctness. There are, however, techniques that generate tests that are guaranteed to determine correctness as long as the IUT satisfies certain conditions. These conditions have been called test hypotheses ([11]) and design for test conditions ([19]). Section 2 discusses test hypotheses. Testing might then be seen as a process of choosing an appropriate set of test hypotheses and then generating a corresponding test set.

If the hypotheses do not hold, the corresponding test set may be ineffective and inefficient. Thus, it is important to use test hypotheses that hold.

Previous work has largely focussed on introducing new hypotheses and generating tests in the presence of hypotheses ([12, 8, 31, 32, 9, 27, 5, 11, 17]). This paper instead concentrates upon semi-automated techniques for establishing that such hypotheses do hold. Specifically the relationship between the uniformity hypothesis and program analysis, through the use of conditioned slicing, is described. Slicing shall be briefly reviewed in Section 3.

Sections 4 and 5 will discuss the following ways in which this relationship may be used.

1. An instance of the uniformity hypothesis, which represents expert knowledge about the IUT, might be used to simplify program analysis.

2. Program analysis might be used to either provide confidence in, refute or refine proposed test hypotheses.

It will thus be demonstrated that there exists a symbiotic relationship between program analysis and test hypotheses.

2 Test hypotheses

Suppose I is to be tested against a specification M with input domain D. It is normal to assume that I accepts the same class of inputs as M, though an error may result for some of these input values. Without any further knowledge about I there is, in general, no finite test set that determines correctness.

Fortunately this does not represent the normal scenario in testing. The tester has some expert knowledge about I and I is not, in general, merely a black box: it is often possible to examine the code used to produce I. There is thus further information about I that may be utilised in test generation. This information might be expressed as properties of I called *test hypotheses*.

Suppose that F denotes the set of possible behaviours of the IUT. F is often called a *Fault Model* ([22]). Suppose, further, that the current set of hypotheses is H and F_H denotes the set of behaviours, from F, that are consistent with H. Let $I' \leq M$ denote that I' conforms to M and $I' \leq_T M$ denote that the I' conforms to M on $T \subseteq D$. Naturally, the notion of conformance used depends upon the specification language. The following defines what it means for a test set T to be guaranteed to determine correctness under H.

Definition 1 *Test set T is* complete *with respect to H if and only if $\forall I' \in F_H . I' \leq M \iff I' \leq_T M$.*

Two related notions, of a test being unbiased and valid, have been described ([11]). A test is *valid* if it rejects all faulty implementations that satisfy the test hypotheses. A test is *unbiased* if it cannot reject a conforming implementation that satisfies the test hypotheses. Then a test is complete if and only if it is unbiased and valid.

Clearly, the exhaustive test set D is complete with respect to every hypothesis H. Exhaustive testing is, however, rarely practical. Given M and I, there are the following, inter-related, challenges.

1. To devise some set H of test hypotheses, that I is likely to satisfy, such that there is a corresponding feasible complete test set.
2. To determine whether I satisfies H.
3. To generate a complete test set for I with respect to H.

The development of an appropriate set of hypotheses can proceed via refinement ([11]). Some minimal hypothesis is produced and this is refined through a number of steps. The minimal hypothesis might, for example, be that I is equivalent to some unknown element from fault model F or simply that the input and output domains for I are the same as those for M. Each refinement strengthens the test hypotheses and thus, potentially, allows a smaller complete test set.

Many test generation techniques are based around partitioning the input domain D into a finite set $D^M = \{D_1, \ldots, D_k\}$ of subdomains such that, according to M, all elements in a subdomain should be processed in the same way ([12, 31, 32, 9, 27, 11, 17]). The *uniformity hypothesis* says that if the input of one value in some $D_i \in D^M$ leads to a failure then all values in D_i lead to failures.

It is to be expected that there is some (unknown) partition D^I, of the input domain, such that the behaviour of I is uniform on each subdomain of D^I. The uniformity hypothesis is thus based upon the assumption that D^M and D^I are similar. If the uniformity hypothesis holds it is sufficient to choose one value from each $D_i \in D^M$. However, the test for some $D_i \in D^M$ is normally complemented by tests around the boundaries of D_i ([32, 9]) which are expected to find any small errors in the boundaries.

It has been noted ([29]) that if the partitions D^M and D^I were known, the behaviours of I and M could be compared on each subdomain from $D^{IM} = \{D_i \cap D_j \mid D_i \in D^M, D_j \in D^I\}$. While we do not consider the generation of D^I, this idea from ([29]) provided the inspiration for much of the work contained in this paper.

3 Program slicing and symbolic evaluation

Program slicing is the process of taking a program I and some slicing criterion (V, n) (variable set V and node n) and removing all parts of I that do not affect the value, at node n, of any variable in V. Much work has focussed on the technical problems associated with slicing programs in the presence of procedures [20, 28], pointers [2, 24, 25] and jumps [3, 7, 14, 1]. This paper uses only *end slicing*, in which the end of the program is the point of interest ([23]). Thus, throughout this paper the slicing criterion is simply a set of variables.

Program slicing was initially introduced as a way of assisting debugging ([30, 26]). For this application it is important that the only simplification tool available to slicing algorithms is statement deletion. For a number of other applications, such as mutation testing ([18]) and program comprehension ([13, 15]), this restriction to statement deletion is unhelpful. In such cases *Amorphous Slicing*, which allows the application of any transformations that preserve the semantics of interest, leads to improved simplification ([13, 16, 4]).

In slicing it is possible to place a condition C on the input values. Then any statement that cannot affect the values of the variables in V at n, given that the input satisfies C, may be removed. This is called *conditioned slicing* ([6, 10]). In the amorphous version of conditioned slicing any transformation that preserves the effect of the original program upon the slicing criterion is valid.

Given a program I and condition C, $S_C(I)$ shall denote the (possibly amorphous) conditioned end slice of I (in which all variables are of interest) for condition C. Similarly, given subdomain $D' \subseteq D$, $S_{D'}(I)$ shall denote the conditioned slice in which the input is constrained to D'. Thus $S_{D'}(I)$ denotes $S_C(I)$ where $C(x)$ is the condition $x \in D'$.

Symbolic evaluation is the process of describing the final values of the variables in a program in terms of the initial values of the variables. Since programs normally have control-flow constructs, the result of applying symbolic evaluation to a program will usually lead to a number of symbolic values, each with a precondition.

4 Uniformity can help program analysis

This section describes a way in which the existence of a uniformity hypotheses may assist program analysis. This is achieved through using the information represented by the uniformity hypothesis. It thus introduces the possibility of using standard testing approaches, that generate a uniformity hypothesis, to assist program analysis.

Suppose subdomain D' of D has been chosen and all of the values in D' are processed in the same way by I. Then the conditioned slice of I on the subdomain D' should be relatively simple. Thus, if D^I were known, this would suggest an approach to program analysis: slice on the subdomains of D^I and analyse these slices.

While D^I is not known, it is possible to slice using the partition D^M, forming the set $S(I, D^M) = \{S_{D_i}(I) \mid D_i \in D^M\}$ of conditioned slices. If the uniformity hypothesis holds the slices in $S(I, D^M)$ should be relatively small. This might help solve one of the challenges of conditioned slicing: finding conditions that lead to small but useful slices.

Consider the program analysis problem of producing a proof of correctness. Then, I conforms to M if and only if for all $D_i \in D^M$, I conforms to M on D_i. Thus, in order to prove that I conforms to M it is sufficient to prove that each $S_{D_i}(I)$ conforms to M on the corresponding D_i. It is then sufficient to consider, for each $D_i \in D^M$, I and M restricted to D_i.

The uniformity hypothesis is based on the behaviour of M being relatively simple on each D_i. If the uniformity hypothesis holds, the $S_{D_i}(I)$ should also be relatively simple. Thus, if the uniformity hypothesis holds, the proof of correctness has been broken down into a number of relatively simple proofs. Symbolic evaluation might be applied to each $S_{D_i}(I)$, producing an expression that can more easily be handled by an automated theorem-prover. Naturally, if the partition D^I defined by I were known, slicing would be applied on $D^{IM} = \{D_i \cap D_j \mid D_i \in D^M, D_j \in D^I\}$. The approach outlined in ([29]) might then be used.

Consider now an implementation $I^\triangle$ that is intended to solve the triangle problem. It thus takes three integers x, y and z and should return:

1. 'equilateral' if x=y and y=z;
2. 'isosceles' if two of x, y, and z are the same but the third is different;
3. 'scalene' if x, y, and z are all different.

The tester might analyse this specification and produce the following conditions:

$$C_1(x, y, z) \equiv x = y \wedge y = z$$

$$\begin{gathered} C_2(x, y, z) \equiv \\ ((x = y) \vee (x = z) \vee y = z)) \\ \wedge \neg(x = y \wedge y = z) \end{gathered}$$

$$C_3(x, y, z) \equiv x \neq y \wedge y \neq z \wedge x \neq z$$

Suppose that the computation contained in $I^\triangle$ is the code shown below.

```
if (x==y && y==z)
   r = "equilateral";
if (x==y) r = "isosceles";
if (x==z) r = "isosceles";
if (y==z) r = "isosceles";
if (x!=y && y!=z && x!=z)
   r = "scalene";
printf("The triangle is %s \n",r);
```

Suppose that $I^\triangle$ is sliced on conditions C_1, C_2 and C_3. The initial step in producing a conditioned slice, of $I^\triangle$, for C_3 might give:

```
if (x!=y && y!=z && x!=z)
   r = "scalene";
```

This reduces to:

```
r = "scalene";
```

Similarly, the first step in the process of applying conditioned slicing with C_2 might give:

```
if (x==y) r = "isosceles";
if (x==z) r = "isosceles";
if (y==z) r = "isosceles";
```

This reduces to:

```
r = "isosceles";
```

Suppose conditioned slicing is applied with C_1. Then any effect of the first three lines is killed by the fourth line. Conditioned slicing might initially produce:

```
if (y==z) r = "isosceles";
```

Again, this may be further reduced, giving:

```
r = "isosceles";
```

The behaviour on each subdomain is quite simple. In fact, in each case it is constant. The information provided by the uniformity hypothesis has thus allowed the generation of small conditioned slices. The existence of these conditioned slices allows the production of simple proofs of correctness, for the subdomains where the behaviour is correct, and the identification of counter-examples where the behaviour is not correct. In this case it is clear that the behaviour on C_2 and C_3 is correct but that the behaviour on C_1 is faulty.

It is worth noting that the production of such simple slices has lent weight to the uniformity hypothesis. Thus, if producing a proof of correctness were not feasible for some subdomain, test derived using the hypothesis might be used instead.

5 Program analysis can help when using uniformity hypotheses

This section describes ways in which program analysis assists a tester when considering using the uniformity hypothesis. These approaches are again based upon the conditioned end slices of I, contained in $S(I, D^M)$, produced by slicing I on the subdomains of the partition D^M.

If a slice $I' = S_{D_i}(I)$ is unexpectedly complex, this might indicate that I takes on more than one behaviour on D_i. This might occur either because this subdomain should be split further or because a boundary is wrong. Then we might either further analyse this slice or test more thoroughly in D_i.

Let $Symb(I, D_i)$ denote the result of applying symbolic evaluation to $S_{D_i}(I)$, $D_i \in D^M$. Then $Symb(I, D_i)$ is a set of pairs, each pair (p, f) consisting of a precondition p and a behaviour f. Suppose $Symb(I, D_i)$ has been produced and it contains more than one behaviour with separate preconditions. These preconditions suggest a refinement of D^M: the subdomain should be partitioned into $\{\{x \in D_i \mid p(x)\} \mid \exists f.(p, f) \in Symb(I, D_i)\}$. The conditioned slices on each of these subdomains may now be produced and these should be relatively simple.

Suppose a slice $S_{D_i}(I) \in S(I, D^M)$ is simple and $Symb(I, D_i)$ contains one behaviour only. This provides some initial confidence in the behaviour of I being uniform on D_i. It might also be possible to further analyse the relationship between the behaviour of $S_{D_i}(I)$ or $Symb(I, D_i)$ and that of M on D_i. This analysis might, for example, involve a proof of correctness. Alternatively, it might involve determining the type of function applied. Where the form of the behaviours of M and I on D_i is known, it may be possible to devise a test set that determines correctness on D_i ([21]), thus overcoming the problem of coincidental correctness.

Consider a system designed to return the sale price of a purchase of rice and lentils. Suppose `x` denotes the amount of lentils being purchased and `y` denotes the amount of rice being purchased. The price of rice is `2` and the price of lentils is `1`. There are discounts for bulk purchases: if the amount of lentils being purchases is greater than or equal to 50 there is a five percent discount and if the total price (without discount) is greater than or equal to 1000 there is a ten percent discount. The discounts are cumulative. Suppose program I^p, containing the following code that performs the computation, has been produced.

```
if (x >= 50.0) p1 = 0.95;
   else p1 = 1.0;
if ((2.0*x+y) >1000.0) p2 = 0.9;
   else p2 = 1.0;
c = p1*p2*(2.0*x+y);
```

There are two basic conditions, $x \geq 50$ and $y \geq 1000$, to consider. This leads to the following four conditions.

$$x < 50 \wedge (2x + y) < 1000$$

$$x < 50 \wedge (2x + y) \geq 1000$$

$$x \geq 50 \wedge (2x + y) < 1000$$

$$x \geq 50 \wedge (2x + y) \geq 1000$$

Consider the second condition, $C(x, y) \equiv x < 50 \wedge (2x + y) \geq 1000$. Then the corresponding conditioned slice of I^p is

```
if (x >= 50.0) p1 = 0.95;
   else p1 = 1.0;
if ((2.0*x+y) >1000.0) p2 = 0.9;
   else p2 = 1.0;
c = p1*p2*(2.0*x+y);
```

This can be further reduced to:

```
p1 = 1.0;
if ((2.0*x+y) >1000.0) p2 = 0.9;
   else p2 = 1.0;
c = p1*p2*(2.0*x+y);
```

and then, using amorphous slicing:

```
if ((2.0*x+y) >1000.0) p2 = 0.9;
   else p2 = 1.0;
c = p2*(2.0*x+y);
```

This cannot be simplified any further. Symbolic evaluation may now be applied, leading to the following precondition/function pairs:

$$(((2x + y) > 1000.0), c = 0.9 * (2x + y))$$

$$(((2x + y) \leq 1000.0), c = (2x + y))$$

The second precondition can be simplified to $2x + y = 1000$. This analysis suggests dividing the subdomain, defined by the precondition $C(x, y) \equiv x < 50 \wedge (2x+y) \geq 1000$, into $C_1(x, y) \equiv x < 50 \wedge (2x + y) > 1000$ and $C_2(x, y) \equiv x < 50 \wedge (2a+y) = 1000$. Any test case taken from the second of these subdomains will lead to a failure.

6 Future Work

This paper has considered ways in which program analysis and test generation complement one another. In particular, a relationship between the uniformity hypothesis and conditioned slicing is explored. There is, however, a general principle contained in this work: information contained in test hypotheses may assist when analysing a program and program analysis may assist when using test hypotheses. This general approach may extend to other types of test hypotheses and forms of program analysis.

The potential role of program analysis, when using test hypotheses, suggests the challenge of devising test hypotheses that

1. are likely to hold;
2. lead to feasible tests that are easy to generate;
3. are relatively easy to verify using program analysis.

Other test hypotheses might represent information that can assist in program analysis. This suggests the investigation of information contained in test hypotheses, information that might assist particular forms of program analysis, and any relationships between these types of information.

7 Conclusions

Many test techniques make assumptions, often called test hypotheses, about the implementation under test. These allow stronger statements to be made about the effectiveness of testing if the test hypotheses hold. However, if the hypotheses do not hold then the tests generated may have little value.

Program analysis is capable of providing general information about implementations. Often, however, program complexity limits the use and effectiveness of program analysis.

This paper has considered the relationship between test hypotheses and program analysis. Within this it has concentrated on the uniformity hypothesis. Program analysis may provide confidence in or refute the test hypotheses or may suggest refinements to the hypotheses. The information provided by the existence of the uniformity hypothesis can be used to simplify program analysis through the production of small conditioned slices.

References

[1] H. Agrawal. On slicing programs with jump statements. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 302–312, Orlando, Florida, June 20–24 1994. Proceedings in SIGPLAN Notices, 29(6), June 1994.

[2] H. Agrawal, R. A. DeMillo, and E. H. Spafford. Dynamic slicing in the presence of unconstrained pointers. In 4th *ACM Symposium on Testing, Analysis, and Verification (TAV4)*, pages 60–73, 1991. Appears as Purdue University Technical Report SERC-TR-93-P.

[3] T. Ball and S. Horwitz. Slicing programs with arbitrary control–flow. In Peter Fritzson, editor, 1st *Conference on Automated Algorithmic Debugging*, pages 206–222, Linköping, Sweden, 1993.

Springer. Also available as Uniersity of Wisconsin–Madison, technical report (in extended form), TR-1128, December, 1992.

[4] D. W. Binkley. Computing amorphous program slices using dependence graphs and a data-flow model. In *ACM Symposium on Applied Computing*, pages 519–525, The Menger, San Antonio, Texas, U.S.A., 1999. ACM Press, New York, NY, USA.

[5] G. V. Bochmann, A. Das, R. Dssouli, M. Dubuc, A. Ghedamsi, and G. Luo. Fault models in testing. In *Protocol Test Systems IV*, pages 17–30. Elsevier Science (North-Holland), 1992.

[6] G. Canfora, A. Cimitile, and A. De Lucia. Conditioned program slicing. In Mark Harman and Keith Gallagher, editors, *Information and Software Technology Special Issue on Program Slicing*, volume 40, pages 595–607. Elsevier Science B. V., 1998.

[7] J.–D. Choi and J. Ferrante. Static slicing in the presence of goto statements. *ACM Transactions on Programming Languages and Systems*, 16(4):1097–1113, July 1994.

[8] T. S. Chow. Testing software design modelled by finite state machines. *IEEE Transactions on Software Engineering*, 4:178–187, 1978.

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

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

[11] M. C. Gaudel. Testing can be formal too. In *TAPSOFT'95*, pages 82–96. Springer-Verlag, March 1995.

[12] J. B. Goodenough and S. L. Gerhart. Towards a theory of test data selection. *IEEE Transactions on Software Engineering*, 1:156–173, 1975.

[13] M. Harman and S. Danicic. Amorphous program slicing. In 5^{th} *IEEE International Workshop on Program Comprehesion (IWPC'97)*, pages 70–79, Dearborn, Michigan, USA, May 1997. IEEE Computer Society Press, Los Alamitos, California, USA.

[14] M. Harman and S. Danicic. A new algorithm for slicing unstructured programs. *Journal of Software Maintenance*, 10(6):415–441, 1998.

[15] M. Harman, C. Fox, R. M. Hierons, D. Binkley, and S. Danicic. Program simplification as a means of approximating undecidable propositions. In 7^{th} *IEEE International Workshop on Program Comprehesion (IWPC'99)*, pages 208–217, Pittsburgh, Pennsylvania, USA, May 1999. IEEE Computer Society Press, Los Alamitos, California, USA.

[16] M. Harman, Y. Sivagurunathan, and S. Danicic. Analysis of dynamic memory access using amorphous slicing. In *IEEE International Conference on Software Maintenance (ICSM'98)*, pages 336–345, Bethesda, Maryland, USA, November 1998. IEEE Computer Society Press, Los Alamitos, California, USA.

[17] R. M. Hierons. Testing from a Z specification. *Journal of Software Testing, Verification and Reliability*, 7:19–33, 1997.

[18] R. M. Hierons, M. Harman, and S. Danicic. Using program slicing to assist in the detection of equivalent mutants. *Journal of Software Testing, Verification and Reliability*, 9r:233–262, 1999.

[19] M. Holcombe and F. Ipate. *Correct Systems: Building a Business Process Solution*. Springer-Verlag, 1998.

[20] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–61, 1990.

[21] W. E. Howden. Algebraic program testing. *ACTA Informatica*, 10:55–56, 1978.

[22] ITU-T. *Z.500 Framework on formal methods in conformance testing*. International Telecommunications Union, 1997.

[23] A. Lakhotia. Rule–based approach to computing module cohesion. In *Proceedings of the* 15th *Conference on Software Engineering (ICSE-15)*, pages 34–44, 1993.

[24] P. E. Livadas and A. Rosenstein. Slicing in the presence of pointer variables. Technical Report SERC-TR-74-F, Computer Science and Information Services Department, University of Florida, Gainesville, FL, June 1994.

[25] J. R. Lyle and D. Binkley. Program slicing in the presence of pointers. In *Foundations of Software Engineering*, pages 255–260, Orlando, FL, USA, November 1993.

[26] J. R. Lyle and M. Weiser. Automatic program bug location by program slicing. In 2nd *International Conference on Computers and Applications*, pages 877–882, Peking, 1987. IEEE Computer Society Press, Los Alamitos, California, USA.

[27] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating tests. *Communications of the ACM*, 31:676–686, 1988.

[28] T. W. Reps. Solving demand versions of interprocedural analysis problems. In Peter Fritzon, editor, *Compiler Construction, 5th International Conference*, volume 786 of *Lecture Notes in Computer Science*, pages 389–403, Edinburgh, U.K., 7–9 April 1994. Springer.

[29] D. J. Richardson and L. A. Clarke. Partition analysis: A method combining testing and verification. *IEEE Transactions on Software Engineering*, 14:1477–1490, 1985.

[30] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[31] E. J. Weyuker and T. J. Ostrand. Theories of program testing and the application of revealing subdomains. *IEEE Transactions on Software Engineering*, 6:236–246, 1980.

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