Using Program Slicing to Assist in the Detection of Equivalent Mutants

Rob Hierons and Mark Harman,
Department of Mathematical and Computing Sciences,
Goldsmiths College, University of London,
Lewisham Way, New Cross, London SE14 6NW, UK.
Tel: +44 (0)171 919 7850. Fax: +44 (0)171 919 7853.
{M.Harman,R.Hierons}@gold.ac.uk

and

Sebastian Danicic,
School of Informatics, University of North London,
Eden Grove, London, N7 8DB.
Tel: +44 (0)171 607 2789. Fax: +44 (0)171 753 7009.
s.danicic@unl.ac.uk

Keywords: mutation testing, program slicing, equivalent mutants, stubborn mutants

Abstract

While mutation testing has proved to be an effective way of finding software faults, currently it is only applied to relatively small programs. One of the main reasons for this is the human analysis required in detecting equivalent mutants. Here program slicing is used to simplify this problem. Program slicing is also used to reduce the number of equivalent mutants produced.

1 Introduction

Mutation testing is based on the notion that any test set that is capable of distinguishing a program p from programs syntactically similar to p is likely to be good at detecting faults in p ([Hamlet, 1977], [DeMillo et al., 1978], [Woodward, 1993], [Budd, 1981]). In mutation testing the program p is mutated, or changed, to form a set M of mutants. Both p and the programs in M are then tested.

An input value σ *kills* a mutant p' if it distinguishes between p and p' in some well defined manner. In the original form of mutation testing, now called strong mutation testing, input σ kills p' if p and p' produce different output when executed on σ. Several alternative definitions of what it means for input to distinguish between two programs have since been developed: these will be described in section 5. Naturally, a mutant p' of a program p may be indistinguishable from p; p' is an *equivalent mutant* if it cannot be killed by any test case.

Empirical studies have shown mutation testing to be highly effective. For example, two studies have compared mutation testing to all-uses data flow testing ([Frankl et al., 1997], [Offutt et al., 1996b]). These studies concern relatively small programs, but the teams involved and the mutation operators and programming languages used were different in each case. The general finding appears to be that mutation testing is more effective than data flow testing (though not markedly so) and that a test set which is adequate according to mutation testing criteria is highly likely to be adequate according to data flow testing criteria; more likely, than a data flow adequate test set is to be mutation adequate.

However, there is a significant drawback in mutation testing which inhibits its wider take-up: it is possible to make a syntactic change to a program, without affecting its semantics. Clearly, in considering adequacy criteria, only non-equivalent mutants should be used. However, the problem of detecting equivalent mutants is undecidable, and so removing equivalent mutants is non-trivial.

Although empirical studies all point to the effectiveness of mutation testing, they also all point to its *cost*. In their comparison of all-uses data flow testing and mutation testing, Frankl et al. comment

> "Even for these small programs the human effort needed to check a large number of mutants for equivalence was almost prohibitive." ([Frankl et al., 1997].

Even a small program such as the triangle categorization program, with a mere 28 executable statements, has 951 mutants (using the 22 Mothra ([King and Offutt, 1991]) mutation operators), of which 109 are equivalent ([Offutt and Pan, 1997]).

Unfortunately, as the problem of detecting equivalent mutants is undecidable, there will always remain a set of equivalent mutants which go undetected by *any* automated system. Whilst automation can diminish human effort, it can never eradicate the need for it.

The main contribution of this paper is a technique for assisting the human analysis of mutants. Instead of attempting to answer the question of equivalence, the approach presented here uses a program simplification process (program slicing), attempting to create the simplest program which denotes the answer to the equivalence question. In cases where no amount of automation can fully answer the question, this approach allows automation to be exploited in *partially* answering the question. The simplified program is an approximate answer; the greater the level of simplification, the closer the approximation. While this does not answer the question for the human analyst, as will be shown, it can reduce the effort involved in detecting equivalence. Where the mutant is not equivalent, it can help the tester find input that kills the mutant.

Throughout this paper it will be assumed that any program considered is written in an imperative programming language with side effect free expressions. The work generalises to languages that allow side effects in expressions but this complicates the exposition. All examples will be written using a notation that is consistent with C. The features of C used are not semantically dissimilar to those available in FORTRAN and thus the Mothra mutation set is applicable.

The rest of the paper is organized as follows. Mutation testing and program slicing are described in sections 2 and 3. In section 4 a new notation is introduced: this is used in, section 5, in order to formalize and compare the different forms of mutation testing. This notation is later used to describe the relationship between mutation testing and slicing. Approaches, that use program slicing in order to assist in mutation testing, are given in sections 6 and 7. Finally, conclusions are drawn.

2 Mutation Testing

This section introduces mutation testing, initially outlining mutant generation. Two types of mutant and different definitions of what it means for a mutant to be killed are then described.

2.1 Generating mutants

Mutants are generated by applying one or more operators from a set of *mutation operators*. A *mutation* is the application of a mutation operator to a program. Consider, for example, the program (based[1] on the Kernighan and Ritchie `itoa` library routine [Kernighan and Ritchie, 1988]) given in Figure 1. This program, which shall be denoted p_0 throughout the paper, inputs an integer and outputs a string of characters that represents the number. Line numbers have been added in order to relate syntactic components of the program to corresponding nodes in the Control Flow Graph (CFG).

There are many possible mutations of p_0, such as changing line 15 from `if (sign<0)` to `if (sign<=0)` or changing line 8 from `s[0] = n%10+'0'` to `s[0] = n/10+'0'`. Interestingly, replacing line 5, `sign=n`, by `sign=2*n` will not affect the input/output behaviour; this is an example of a mutation which gives rise to an equivalent mutant.

In order to reduce the set of mutants considered, it is useful to say what it means for p' to be close to p. The mutant p' is a *first order mutant* if p' can be generated from p by a single application of a mutation operator. In general, p' is an *nth order mutant* if it can be produced from p by a sequence of n mutations.

The *competent programmer hypothesis* states that most faults are syntactically small, and thus that low order mutants represent most real faults ([DeMillo et al., 1978]). It has also been argued that complex faults are *coupled* to simple faults and thus that tests that are good at finding low order mutants are likely to be good at finding more complex differences ([DeMillo et al., 1978], [Offutt, 1992]). These findings suggest that it is sufficient to consider only low order mutants. In practice, usually only first order mutants are used; only first order mutants will be considered in this paper.

Most empirical studies use the Mothra tool, that produces mutants from FORTRAN 77 code using the 22 mutation operators listed in Figure 2 ([King and Offutt, 1991]). It has been found that a subset of these, *E-selective* $= \{ABS, AOR, LCR, ROR, UOI\}$, is effective: in an empirical study, test sets that killed all mutants produced using the E-selective operators also killed over 98% of mutants produced by the complete set of Mothra operators ([Offutt et al., 1996a]). The number of mutants produced using the full set of Mothra mutation operators is quadratic in the number of lines of code and the number of variable references. Fortunately, the number of mutants generated using the E-selective operators is linear in the number of lines of code and the number of variable references ([Offutt et al., 1996a]). Thus, E-selective forms a scalable yet effective set of mutation operators.

[1] In Figure 1 the digits of the number are reversed in the translation process, but these details are not of concern here.

```
main()
{
char s[MAX] ;
int n, i, sign ;
scanf("%d",n) ;
sign=n ;
if (n<0)
   n = -n ;
s[0]=n%10+'0' ;
n=n/10 ;
i=1 ;
 while (n>0)
{
   s[i] = n%10 + '0' ;
   n=n/10 ;
   i=i+1 ;
 }
if (sign <0)
{
   s[i] = '-' ;
   i=i+1 ;
}
s[i] = '\0' ;
printf("%s",s) ;
}
```

Figure 1: The program p_0

Name	Description
AAR	array reference for array reference
ABS	absolute value insertion
ACR	array reference for constant replacement
AOR	arithmetic operator replacement
ASR	array reference for scalar variable replacement
CAR	constant for array reference replacement
CNR	comparable array name replacement
CRP	constant replacement
CSR	constant for scalar replacement
DER	DO statement end replacement
DSA	DATA statement alterations
GLR	GOTO label replacement
LCR	logical connector replacement
ROR	relational operator replacement
RSR	RETURN statement replacement
SAN	replace statement by TRAP
SAR	scalar variable for array reference replacement
SCR	scalar for constant replacement
SDL	statement deletion
SRC	source constant replacement
SVR	scalar variable replacement
UOI	unary operator insertion

Figure 2: The Mothra Operators

2.2 Classes of Mutations

There are two main classes of mutation: *syntactic mutation* and *data state mutation*. In syntactic mutation, the syntax of the program is altered. In data state mutation, the internal state is altered. There are three ways of defining what it means for a mutant, of either type, to be killed: *strong mutation*, *firm mutation*, and *weak mutation*. In general, a mutant p' of a program p is killed by a test case t if during testing with t the internal states of p and p' differ at some point. The different types of mutation testing differ in the point they consider. Weak mutation considers the point after the mutated statement. In strong mutation the difference must propagate to different output while in firm mutation the tester may choose the end of any section of the program that contains the mutated statement. These different forms of mutation testing will be described in section 5. Mothra produces syntactic mutations.

Most syntactic mutations affect either a predicate, the right hand side (RHS) of an assignment or the left hand side (LHS) of an assignment. Only these types of mutation will be considered here. Of Mothra's 22 mutation operators, 16 fall into this category. Further, *all* those from the E-selective operator set ([Offutt et al., 1996a]) are from these types.

The syntactic mutant p' of p, that is formed by replacing some $\mathtt{e}$ by $\mathtt{e}'$ at node n, shall be denoted $m(p, n, \mathtt{e} \rightarrow \mathtt{e}')$. Here $\mathtt{e}$ may be the whole content of the node n in p or some part of this. A data state mutation of a program p is defined by a node n and a function μ that takes a data state and returns a data state. The mutant produced by applying μ after n shall be denoted $md(p, n, \mu)$. If μ is allowed to affect the program counter it is possible to produce a wide range of mutants.

Unless otherwise stated, p' shall denote a mutant that has been generated by mutating p at node n.

3 Slicing

Program slicing ([Weiser, 1984]) is a technique for isolating the components of a program which are concerned with the computation of a single variable or set of variables at some point in the program. Slices are constructed with respect to a slicing criterion, a pair (V, n), for some set of variables V and node n in the program's control flow graph. Nodes of the graph of a program p, which cannot affect the value of any variable in V when the next statement to be executed is at node n, are removed to form a slice of p with respect to (V, n).

Consider the program in the left hand section of Figure 3 and two of its slices. The original program computes the value of the biggest and smallest element of the array `A`, storing the result in `b` and `s` respectively. The slices represent the parts of the original concerned with each of these computations. Line numbering is used to identify the nodes of the CFGs of the program and its slices. Simultaneous slicing on both slicing criteria would have had the effect of taking the union of the two individual slices.

This form of slicing is known as static backward slicing ([Venkatesh, 1991]), to distinguish it from dynamic slicing ([Korel and Rilling, 1998]), conditioned slicing ([Canfora et al., 1998]) and forward slicing ([Horwitz et al., 1990]). This paper will make use of static backward slicing only and so hereinafter the term 'slice' should be regarded as a reference to this form of slice.

The issue of reconstructing the syntax of the program from the reduced CFG is interesting ([Weiser, 1979]), but beyond the scope of this paper; it will be assumed that there is an obvious way to construct the syntax of a slice from the reduced CFG produced by the slicing process.

Slicing has been the subject of an intensive research programme from its inception in the late seventies. Much work has been concerned with tackling slicing in the presence of a variety of language features such as procedural abstraction ([Horwitz et al., 1990]), pointers ([Agrawal et al., 1991],

1 b=A[0] ;
2 s=A[0] ;
3 for(i=1;i<=MAX;i++)
{
4 if (b<A[i])
5 b=A[i];
6 if (s>A[i])
7 s=A[i];
}
8 printf("%d",b);
9 printf("%d",s);
	b=A[0] ;
	
	for(i=1;i<=MAX;i++)
	{
	if (b<A[i])
	b=A[i];
	
	}
		
		s=A[0] ;
		for(i=1;i<=MAX;i++)
		{
		
		if (s>A[i])
		s=A[i];
		}

Original Program	Slice w.r.t. ({b}, 8)	Slice w.r.t. ({s}, 8)

Figure 3: A program fragment and two of its slices

```
for(i=0,sum=a[0],biggest=sum;i<20;sum=sum+a[++i])
if (a[i] > biggest) biggest = a[i] ;
average=sum/20 ;
```

Figure 4: A Program to be sliced

[Livadas and Rosenstein, 1994], [Lyle and Binkley, 1993], [Ernst, 1994]), object oriented programming constructs ([Larsen and Harrold, 1996], [Tip et al., 1996]), unstructured control flow in the dynamic ([Korel, 1995]) and static ([Choi and Ferrante, 1994], [Harman and Danicic, 1998], [Agrawal, 1994], [Ball and Horwitz, 1993]) paradigms and concurrent execution ([Krinke, 1998], [Cheng, 1993]). Excellent surveys of the theory and practice of slicing can be found in survey papers by ([Tip, 1994]) and ([Binkley and Gallagher, 1996]).

Conventional program slicing is limited in its ability to simplify the program to be sliced, because of the restriction to command deletion. This restriction is necessary for some applications, for example debugging, where it is essential to maintain a syntactic connection between the slice and the program from which it is constructed.

For applications like program comprehension, where the simplicity of the slice is paramount, and syntax preservation is relatively unimportant, amorphous slicing [Harman and Danicic, 1997] maybe more appropriate. An amorphous slice maintains the semantic requirement of slicing (that the program and slice have identical behaviour with respect to the slicing criterion), but drops the syntactic requirement (that a slice is constructed by deleting nodes from the original program's CFG). An amorphous slice is therefore always as thin as, and often considerably thinner than a conventional slice constructed for an identical slicing criterion.

Consider the program in Figure 4. The conventional static slice constructed with respect to the final value of the variable `biggest` contains all but the final assignment to `average`. By contrast a corresponding amorphous slice is depicted in Figure 5.

4 A Formal Notation

A new notation, that will allow mutation testing to be formally defined, will be introduced in this section. This will allow the property of mutant equivalence to be explored.

```
biggest=a[0];
for(i=0;i<20;i++) if (a[i] > biggest) biggest = a[i];
```

.

Figure 5: Amorphous Slice for `biggest`

At times it will be necessary to consider the internal state of program p both before and after the execution of a node n during some execution profile of p. At any time during the execution of p the internal state will be seen as a function E that gives the current values of the variables: one of these is the program counter η. Where the value of a variable $\mathtt{x}$ is undefined, $E(\mathtt{x}) = \perp$. Let *States* be the set of all such state functions.

The term $\overrightarrow{R}_{pn\sigma i\mathtt{x}}$ will denote the value of the variable $\mathtt{x}$ directly after the node n has been executed for the ith time when the program p has been run on input σ. Thus, if the state is E after the ith execution of n when p is executed on input σ, then $\overrightarrow{R}_{pn\sigma i\mathtt{x}} = E(\mathtt{x})$. If $\mathtt{x}$ is not defined at this point or the node n is not executed i times or more, the result is undefined: $\overrightarrow{R}_{pn\sigma i\mathtt{x}} = \perp$. Similarly, $\overleftarrow{R}_{pn\sigma i\mathtt{x}}$ will denote the value of the variable $\mathtt{x}$ directly before n has been executed for the ith time when the program p has been run on input σ.

Sometimes it will be necessary to consider the output of a program. In order to do this, within the notation described, a stream based approach will be adopted in which output will be seen as updating a special variable ϕ ([Horwitz et al., 1990]). Initially ϕ is the empty string. Each output statement adds to the end of ϕ. Thus, at the end of the execution of a program p, ϕ contains a list of all output from p.

It is possible to consider $\overrightarrow{R}$ and $\overleftarrow{R}$ to be functions that take a number of parameters. It will be assumed that $\overrightarrow{R}$ and $\overleftarrow{R}$ can be partially applied. Thus, for example, $\overrightarrow{R}_{pn\sigma i}$ is a function that takes a variable name and returns a value.

Given a program p and node n, at times it will be necessary to add a statement s to p, at a new node $pred(n)$ before n, to form a program $Add_b(p, n, s)$. If necessary, suitable variable type declarations are also added. Control flow passes through n in p if and only if it passes through $pred(n)$ and then n in $Add_b(p, n, s)$. Similarly, given a node n of p and a statement s it is possible to create a program $Add_a(p, n, s)$ by placing s in a new node $succ(n)$ that follows n: control flow passes through n in p if and only if it passes through n and then $succ(n)$ in $Add_a(p, n, s)$.

Given expression $\mathtt{e}$, $REF(\mathtt{e})$ is the set of variables referenced by $\mathtt{e}$ ([Aho et al., 1986]). Given a program p and set of state variables X, the static slice at n of p for X shall be denoted $SS(p, n, X)$. Where $X = \{\mathtt{x}\}$, for some variable $\mathtt{x}$, this shall (with a slight abuse of notation) be denoted $SS(p, n, \mathtt{x})$. Where slicing occurs at the end of the program, these will be shortened to $SS(p, X)$ and $SS(p, \mathtt{x})$ and the slicing criterion will simply be a set of variables.

5 Killing mutants

In this section a number of different notions, of what it means to kill a mutant, will be defined. Suppose p and p' are run using input σ. It is possible to compare p and p' by considering either their internal states or their output. If the internal state is being considered, there are choices as to where to compare the internal states. ¿From these differences it is possible to derive a number of definitions of distinguishing mutants; some of these will be considered in this section.

5.1 Strong mutation

Traditionally, under strong mutation, p' is killed by input σ if p and p' produce different output when executed using σ. It has been argued that the output should not be considered separately: that the output is simply part of the internal state. Instead it is possible to consider the final state of the program ([Woodward and Halewood, 1988]).

It is thus possible to consider two forms of strong mutation. The traditional form, which considers only the program output, shall be called strong *output* mutation. The alternative, which considers the final state, shall be called strong *state* mutation. Under strong state mutation it is possible to restrict comparison to some subset X of the variable space.

Let ω denote the exit node. Under strong output mutation, input σ kills p' if and only if the values of ϕ at the end of the execution of p and p' are different. As ω is only executed once, this is equivalent to

$$\overleftarrow{R}_{p\omega\sigma 1\phi} \neq \overleftarrow{R}_{p'\omega\sigma 1\phi}.$$

Equivalently, under strong output mutation, p' is an equivalent mutant if and only if

$$\forall \sigma \in States \bullet \overleftarrow{R}_{p\omega\sigma 1\phi} = \overleftarrow{R}_{p'\omega\sigma 1\phi}.$$

Suppose strong state mutation and variable space X are being considered. Then σ kills p' if and only if there is some variable $\mathtt{x} \in X$ such that the values of $\mathtt{x}$ at the end of the execution of p and p', on σ, are different. This is the case if and only if

$$\exists \mathtt{x} \in X \bullet \overleftarrow{R}_{p\omega\sigma 1\mathtt{x}} \neq \overleftarrow{R}_{p'\omega\sigma 1\mathtt{x}}.$$

The function $\overleftarrow{R}_{p\omega\sigma}$ takes parameters i and $\mathtt{x}$ and returns: $\perp$ if $i \neq 1$; the final value of $\mathtt{x}$ if $i = 1$. This is because ω only executes once. Thus, this is equivalent to

$$\exists \mathtt{x} \in X, i \bullet \overleftarrow{R}_{p\omega\sigma i\mathtt{x}} \neq \overleftarrow{R}_{p'\omega\sigma i\mathtt{x}}.$$

Let $\rhd$ denote a form of domain restriction defined by

$$(R_{pn} \rhd X)(\sigma, i, \mathtt{x}) = \begin{cases} R_{pn\sigma i\mathtt{x}} \text{ if } \mathtt{x} \in X \\ \perp \text{ otherwise} \end{cases}.$$

The function $\overleftarrow{R}_{p\omega\sigma}$ takes a variable $\mathtt{x}$ and an index i and returns the value of $\mathtt{x}$ before the ith execution of ω. If $i \neq 1$ this value is $\perp$. Thus, the functions $(\overleftarrow{R}_{p\omega} \rhd X)\sigma$ and $(\overleftarrow{R}_{p'\omega} \rhd X)\sigma$ differ if and only if there exist i and $\mathtt{x} \in X$ such that $\overleftarrow{R}_{p\omega\sigma i\mathtt{x}} \neq \overleftarrow{R}_{p'\omega\sigma i\mathtt{x}}$. Thus, σ kills p' if and only if

$$(\overleftarrow{R}_{p\omega} \rhd X)\sigma \neq (\overleftarrow{R}_{p'\omega} \rhd X)\sigma.$$

Similarly, p' is an equivalent mutant if and only if

$$\overleftarrow{R}_{p\omega} \rhd X = \overleftarrow{R}_{p'\omega} \rhd X.$$

Under strong output mutation, input σ kills p' if and only if the values of ϕ at the end of the execution of p and p' are different. This is equivalent to

$$(\overleftarrow{R}_{p\omega} \rhd \{\phi\})\sigma \neq (\overleftarrow{R}_{p'\omega} \rhd \{\phi\})\sigma.$$

Similarly, under strong output mutation, p' is an equivalent mutant if and only if

$$\overleftarrow{R}_{p\omega} \rhd \{\phi\} = \overleftarrow{R}_{p'\omega} \rhd \{\phi\}.$$

5.2 Weak mutation

Under weak mutation ([Howden, 1982]), p' is killed by input σ on variable space X if the execution of p and p' on σ leads to different values of some $\mathtt{x} \in X$ immediately after *some* execution of node n. This form of mutation is weaker than strong mutation, as the difference after n need not be propagated to ω. Throughout this section it will be assumed that weak mutation, using variable space X, is being applied.

When considering input σ and a set M_n of mutants of p, that differ from p at node n, it is possible to execute p with σ and record the internal state just before n. This allows the tester to determine which mutants would have been killed by σ, without actually executing the mutants with this input. If the execution passes through n more than once the values should be recorded each time n is met. Naturally, this greatly reduces computational effort, and this is one of the principal advantages of weak mutation over other forms of mutation testing.

Then as $\overrightarrow{R}$ can be partially evaluated, p' is killed by input σ if and only if

$$(\overrightarrow{R}_{pn} \rhd X)\sigma \neq (\overrightarrow{R}_{p'n} \rhd X)\sigma.$$

The only difference between the expressions for weak mutation and strong state mutation is the node considered: for weak mutation it is n while for strong state mutation it is ω. Similarly, p' is an equivalent mutant if and only if

$$\overrightarrow{R}_{pn} \rhd X = \overrightarrow{R}_{p'n} \rhd X.$$

Suppose the program counter η is in X. A special case is where n contains a *predicate* $\mathtt{e}$, which is mutated to form $p' = m(p, n, \mathtt{e} \to \mathtt{e}')$. If either the values of the variables or the program counter differ, after n, for p and p' then p' is killed. Consider the first time, during execution, that the variable spaces of p and p' differ. This must be immediately after n and, as the predicates are side-effect free, at this point the spaces only differ in their values for η. It is thus sufficient to consider only η: to set $X = \{\eta\}$.

Thus p' is killed by input σ if and only if

$$(\overrightarrow{R}_{pn} \rhd \{\eta\})\sigma \neq (\overrightarrow{R}_{p'n} \rhd \{\eta\})\sigma.$$

It follows that p' is an equivalent mutant if and only if

$$\overrightarrow{R}_{pn} \rhd \{\eta\} = \overrightarrow{R}_{p'n} \rhd \{\eta\}.$$

5.3 Firm mutation

Firm mutation ([Woodward and Halewood, 1988]) may be seen as a process in which p is mutated at some point n to form p' and p and p' are compared on some variable subset X at some node n' ([Voas and McGraw, 1998]). Both weak mutation and strong state mutation are special cases of firm mutation: in weak mutation $n' = n$ and in strong state mutation $n' = \omega$. Firm mutation testing is thus a generalization of these approaches. Throughout this section it will be assumed that firm mutation, using variable space X and considering node n', is being applied.

Firm mutation can be seen as a way of studying the effect of mutation on a wide range of structures within the program: strong mutation considers the whole program while weak mutation looks at statements or expressions. Firm mutation allows structures, such as loops, procedure calls, and program fragments to be considered. It is possible to do this by mutating some part of the structure and choosing the final node of that structure to be the point at which p and the mutant are compared.

Given input σ, the function $\overrightarrow{R}_{pn'} \rhd X$ gives the values of the variables in X at n' when p is executed on σ. Thus, p' is killed by input σ if and only if

$$(\overrightarrow{R}_{pn'} \rhd X)\sigma \neq (\overrightarrow{R}_{p'n'} \rhd X)\sigma.$$

Similarly, p' is an equivalent mutant if and only if

$$\overrightarrow{R}_{pn'} \rhd X = \overrightarrow{R}_{p'n'} \rhd X.$$

6 Weak Mutation Testing and Slicing

It is common, during mutation testing, to reach a situation in which there are a few mutants that have not been killed: these are then analysed manually ([Voas and McGraw, 1998]). Where the programs are complex, this analysis can be non-trivial. It is thus useful to reduce the size of the problem.

For weak mutation, a new variable will be introduced. This variable will represent whether the program execution would have killed the mutant. It is then possible to slice on this variable and analyse the slice returned rather than the initial program. Initially expression mutations will be examined: this will be followed by a consideration of state mutations and finally the mutation of the LHS of an assignment.

Throughout this section it will be assumed that weak mutation, using variable space X, is being applied.

6.1 Expression mutations

Suppose $\mathtt{z}$ is not part of the variable space of p, n is either a predicate $\mathtt{e}$ or an assignment whose RHS is $\mathtt{e}$, and $p' = m(p, n, \mathtt{e} \rightarrow \mathtt{e}')$. A new boolean variable $\mathtt{z}$, initially set to *true*, will be added. Every time n is met, the value of $\mathtt{z}$ is updated: if the mutant is killed $\mathtt{z}$ becomes false, if it is not already, and the value of $\mathtt{z}$ is not allowed to change again. The assignment $\mathtt{z} = \mathtt{z}\ \&\&\ (\mathtt{e} == \mathtt{e}')$, introduced as the new node $pred(n)$, suffices.

The addition of this statement to p gives $Add_b(p, n, \mathtt{z} = \mathtt{z}\ \&\&\ (\mathtt{e} == \mathtt{e}'))$. Under weak mutation, input σ kills $m(p, n, \mathtt{e} \rightarrow \mathtt{e}')$ if and only if, at some time during the execution of $Add_b(p, n, \mathtt{z} = \mathtt{z}\ \&\&\ (\mathtt{e} == \mathtt{e}'))$ with input σ, the value of $\mathtt{z}$ is $false$.

The problem of killing a mutant $m(p, n, \mathtt{e} \rightarrow \mathtt{e}')$ has thus been phrased in terms of the value of a variable $\mathtt{z}$ at the node $pred(n)$. It is natural to slice $Add_b(p, n, \mathtt{z} = \mathtt{z}\ \&\&\ (\mathtt{e} == \mathtt{e}'))$ at $pred(n)$ for

$\mathtt{z}$ to produce $SS(Add_b(p,n,\mathtt{z}=\mathtt{z}\ \&\&\ (\mathtt{e}==\mathtt{e}')),pred(n),\mathtt{z})$. Then the execution of $SS(Add_b(p,n,$ $\mathtt{z}=\mathtt{z}\ \&\&\ (\mathtt{e}==\mathtt{e}')),pred(n),\mathtt{z})$ on σ determines whether σ kills $m(p,n,\mathtt{e}\rightarrow\mathtt{e}')$.

Input σ kills $m(p,n,\mathtt{e}\rightarrow\mathtt{e}')$ if and only if

$$\exists i \bullet \overrightarrow{R}_{SS(Add_b(p,n,\mathtt{z}=\mathtt{z}\ \&\&\ (\mathtt{e}==\mathtt{e}')),pred(n),\mathtt{z})pred(n)\sigma i\mathtt{z}} = false.$$

A mutant $m(p,n,\mathtt{e}\rightarrow\mathtt{e}')$ is equivalent to p if and only if, for all σ, $\mathtt{z}$ cannot take on the value $false$ at $pred(n)$ during the execution of $SS(Add_b(p,n,\mathtt{z}=\mathtt{z}\ \&\&\ (\mathtt{e}=\mathtt{e}')),pred(n),\mathtt{z})$ with σ. More formally, $m(p,n,\mathtt{e}\rightarrow\mathtt{e}')$ is an equivalent mutant if and only if

$$\forall\sigma,i \bullet \overrightarrow{R}_{SS(Add_b(p,n,\mathtt{z}=\mathtt{z}\ \&\&\ (\mathtt{e}==\mathtt{e}')),pred(n),\mathtt{z})pred(n)\sigma i\mathtt{z}} = true.$$

Analysis of the static slice, rather than the original program, offers two advantages:

1. the size of the program being considered is reduced;
2. the questions of whether a mutant is equivalent and whether a test case kills a mutant are phrased in terms of the value of a single boolean variable.

Only the effect of the slice on the value of $\mathtt{z}$ is of interest: the syntax is irrelevant. It is therefore possible to exploit the superior simplification afforded by amorphous slicing. Consider, for example, the mutation

```
if (sign<0)
     →
if (sign<=0)
```

of line 15 of p_0. This line may be proceeded by the statement

```
z=z && ((sign<0)==(sign<=0));
```

producing a program p_0'. Figure 6 gives p_0' and Figure 7 gives the program generated by conventional static slicing p_0' for $\mathtt{z}$. The program produced by using amorphous static slicing is given in Figure 8. From this it is clear that the mutant is non-equivalent and is killed by input 0. In this case slicing helps the tester to identify a test case that kills the mutant.

6.2 Data state mutants

In data state mutation at node n, some state transformation function μ is applied after n. Suppose, during the execution of p with input σ, the internal state is E. Then, under weak mutation, the data state mutant is killed if and only if $\mathtt{E}\neq\mu(\mathtt{E})$. It is thus possible to add a new variable $\mathtt{z}$, whose initial value is $true$, and introduce a new node $\mathtt{z}=\mathtt{z}\ \&\&\ (\mathtt{E}==\mu(\mathtt{E}))$ at $succ(n)$ to form the program $Add_a(p,n,\mathtt{z}=\mathtt{z}\ \&\&\ (\mathtt{E}==\mu(\mathtt{E})))$.

Again, it is natural to slice $Add_a(p,n,\mathtt{z}=\mathtt{z}\ \&\&\ (\mathtt{E}==\mu(\mathtt{E})))$ at $succ(n)$ for $\mathtt{z}$. The program produced determines whether an input value kills the mutant $md(p,n,\mu)$.

Thus, input σ kills p' if and only if

$$\exists i \bullet \overrightarrow{R}_{SS(Add_a(p,n,\mathtt{z}=\mathtt{z}\&\&(\mathtt{E}==\mu(\mathtt{E})))succ(n)\sigma i\mathtt{z}} = false.$$

Similarly, p' is an equivalent mutant if and only if

$$\forall\sigma,i \bullet \overrightarrow{R}_{SS(Add_a(p,n,\mathtt{z}=\mathtt{z}\&\&(\mathtt{E}==\mu(\mathtt{E})))succ(n)\sigma i\mathtt{z}} = true.$$

```
1    main()
     {
2    char s[MAX] ;
3    int n, i, sign ;
4    scanf("%d",n) ;
4a   z = 1 ;
5    sign=n ;
6    if (n<0)
7       n = -n ;
8    s[0]=n%10+'0' ;
9    n=n/10 ;
10   i=1 ;
11    while (n>0)
     {
12      s[i] = n%10 + '0' ;
13      n=n/10 ;
14      i=i+1 ;
      }
15b   z=z && ((sign<0)==(sign<=0)) ;
15   if (sign <0)
     {
16      s[i] = '-' ;
17      i=i+1 ;
     }
18   s[i] = '\0' ;
19 printf("%s",s) ;
     }
```

Figure 6: The program p'_0

```
1    main()
     {
2    char s[MAX] ;
3    int n, i, sign ;
4    scanf("%d",n) ;
4a   z = 1 ;
5    sign=n ;
16b  z=z && ((sign <0)==(sign <=0)) ;
     }
```

Figure 7: The conventional slice of p'_0

```
main()
{
char s[MAX] ;
int n ;
scanf("%d",n) ;
z=((n <0)==(n <=0)) ;
}
```

Figure 8: An amorphous slice of p'_0

6.3 Mutating the LHS of an assignment

The approaches described above can be extended to mutants that affect the LHS of an assignment. Suppose a program p has been mutated, to form p', by changing the assignment $\mathtt{y} = \mathtt{e}$ at node n to $\mathtt{x} = \mathtt{e}$. Thus $p' = m(p, n, \mathtt{y} = \mathtt{e} \rightarrow \mathtt{x} = \mathtt{e})$. Under weak mutation, p' is killed by input σ if, at some time immediately before control flow passes to node n during the execution of p' with σ, the statement $\mathtt{x} = \mathtt{e}$ leads to a different data state than the statement $\mathtt{y} = \mathtt{e}$. This happens if n is reached with either $\mathtt{x}$ or $\mathtt{y}$ not having the value $\mathtt{e}$.

Again, it is possible to add a variable $\mathtt{z}$, whose initial value is $true$, that determines whether the mutant is killed: the program $Add_b(p, n, \mathtt{z} = \mathtt{z}\ \&\&\ (\mathtt{x} == \mathtt{e})\ \&\&\ (\mathtt{y} == \mathtt{e}))$ is generated. Under weak mutation, p' is killed by input σ if and only if, at some point the variable $\mathtt{z}$ takes on the value $false$. It is possible to slice on $\mathtt{z}$ at $pred(n)$ to get $SS(Add_b(p, n, \mathtt{z} = \mathtt{z}\ \&\&\ (\mathtt{x} == \mathtt{e})\ \&\&\ (\mathtt{y} == \mathtt{e})), pred(n), \mathtt{z})$. Then under weak mutation, the mutant is killed by input σ if and only if

$$\exists i \bullet \overrightarrow{R}_{SS(Add_b(p,n,\mathtt{z}=\mathtt{z}\ \&\&\ (\mathtt{x}==\mathtt{e})\ \&\&\ (\mathtt{y}==\mathtt{e})),pred(n),\mathtt{z})pred(n)\sigma i\mathtt{z}} = false.$$

The mutant is equivalent if and only if

$$\forall \sigma, i \bullet \overrightarrow{R}_{SS(Add_b(p,n,\mathtt{z}=\mathtt{z}\ \&\&\ (\mathtt{x}==\mathtt{e})\ \&\&\ (\mathtt{y}==\mathtt{e})),pred(n),\mathtt{z})pred(n)\sigma i\mathtt{z}} = true.$$

Again, amorphous slicing may be applied.

6.4 Amorphous slicing

Amorphous slicing allows the application of a larger set of simplification transformations and thus may lead to smaller slices. Consider the function substring, given in Figure 9, which sets `target` to the substring between the indices `from` and `to` (exclusive) in `source`.

Using Mothra's SVR (Scalar Variable Replacement), one of the mutants obtained is the equivalent mutant in Figure 10. The point of mutation is marked Δ. The program to be sliced in order to attempt to detect equivalence (under weak mutation) for this mutant is that given in Figure 11.

The conventional static slice of this mutant (constructed with respect to `z`) is the program given in Figure 12. Conventional static slicing is helpful, because the size of the mutant is reduced, which presumably increases the speed of human recognition. However, the size of the slice is far larger than that of the corresponding amorphous slice. That is, the *theoretical* minimal amorphous slice of *any* equivalent mutant will be

```
substring(int from, int to,
char target[], char source[])
{
from++;
start=from;
if (from < to)
  { while (from != to)
      { target[from-start] = source[from];
        from++;
      }
    target[to-start] = '\0';
  }
  else target[0] = '\0';
}
```

Figure 9: A program to be mutated

```
substring(int from, int to,
char target[], char source[])
{
from++;
start=from;
if (from < to)
  { while (from != to)
      { target[from-start] = source[from];
        from++;
      }
    target[from-start]= '\0'; (Δ)
  }
  else target[0] = '\0';
}
```

Figure 10: The mutant of Figure 9

```
substring(int from, int to,
char target[], char source[])
{
z=TRUE;
from++;
start=from;
if (from < to)
  { while (from != to)
      { target[from-start] = source[from];
        from++;
      }
     z=z && from == to;
     target[from-start] = '\0';
  }
  else target[0] = '\0';
}
```

Figure 11: The mutant to be sliced

```
substring(int from, int to,
char target[], char source[])
{
z=TRUE;
from++;
if (from < to)
  { while (from != to)
      from++;
  z=z && from == to;
  }
}
```

Figure 12: The mutant after conventional slicing

```
(a)
z=TRUE;
from++;
if (from<to)
{ while (from != to) from++;
z=z && TRUE;
}
```

```
(b)
z=TRUE;
from++;
if (from<to)
{ while (from != to) from++;
z=TRUE && TRUE;
}
```

```
(c)
z=TRUE;
from++;
if (from<to)
{ while (from != to) from++;
z= TRUE;
}
```

```
(d)
z=TRUE;
from++;
if (from<to)
z=TRUE ;
```

Figure 13: Applying transformations

```
z = TRUE
```

If there were to be an ideal amorphous slicing algorithm (which always delivered minimal slices) then it would yield this slice for all equivalent mutants, and the equivalent mutant problem would disappear. Unfortunately, in general, minimal amorphous slicing, like minimal conventional slicing is not computable. Indeed, the foregoing discussion is effectively an informal proof of this. However, in many cases, with appropriate domain-specific choice of transformation set, it appears that relatively powerful amorphous slicing algorithms can be defined [Harman et al., 1998]. This example is a case in point.

Path condition analysis of the kind used in conditioned slicing [Canfora et al., 1998] allows the body of the program to be transformed to that given in Figure 13a. Constant propagation allows this to be re-written to that given in Figure 13b. Constant evaluation yields the code given in Figure 13c. Conventional slicing produces the fragment given in Figure 13d. Finally, case analysis could be expected to produce the program whose body is simply `z=TRUE`.

Clearly, choosing equally effective transformation sequences for other examples is non-trivial, but as amorphous slices are guaranteed to be no thicker than conventional slices, and syntax preservation is unimportant (even a hindrance), there appears to be no reason *not* to choose amorphous slicing for this application.

7 Firm Mutation and Slicing

Firm mutation is a generalization of weak and strong state mutation. Suppose a program has been mutated at n and it and its mutant. Are being compared at node n'. In order to introduce a variable whose value represents whether the mutant is killed it is necessary to consider each path from n to n'. In general this is infeasible. Throughout this section it will be assumed that firm mutation, using variable space X and node n', is being applied.

Where there are only a few control flow paths from n to n', it may be feasible to symbolically evaluate these subpaths for p and p' and thus include a statement that represents the differences between the states of p and p' at n'. It is, however, generally easier to produce the mutant. Slicing can still, however, reduce the size of any program being considered.

7.1 Applying slicing

During execution, only the values of the variables from X, after the execution of n', are of interest. Thus, it is possible to slice p and p' with respect to (X, n'). These slices, $SS(p, n', X)$ and $SS(p', n', X)$, preserve the effect of the programs on X at n' and are, generally, simpler than the original programs.

Input σ kills p' at n' if and only if

$$(\overrightarrow{R}_{SS(p,n',X)n'} \rhd X)\sigma \neq (\overrightarrow{R}_{SS(p',n',X)n'} \rhd X)\sigma.$$

The mutant p' is equivalent if and only if

$$\overrightarrow{R}_{SS(p,n',X)n'} \rhd X = \overrightarrow{R}_{SS(p',n',X)n'} \rhd X.$$

Suppose $X_1, \ldots, X_r$ cover X ($\cup_i X_i = X$). It is possible to produce, for each $i \in \{1, \ldots, r\}$, the slices $SS(p, n'X_i)$ and $SS(p', n', X_i)$. With an appropriate choice of X_i, these slices may be much simpler than $SS(p, n', X)$ and $SS(p', n', X)$. Then input σ kills p' at n' if and only if

$$\exists j \bullet (\overrightarrow{R}_{SS(p,n',X_j)n'} \rhd X_j)\sigma \neq (\overrightarrow{R}_{SS(p',n',X_j)n'} \rhd X_j)\sigma.$$

Similarly, the mutant is equivalent if and only if

$$\forall j \bullet \overrightarrow{R}_{SS(p,n',X_j)} \rhd X_j = \overrightarrow{R}_{SS(p',n',X_j)} \rhd X_j.$$

Where the slices produced using the X_i are significantly simpler then the slice produced using X, this may help the tester determine whether p' is equivalent and, if it is not equivalent, kill p'.

Consider the mutation `s[0]=n%10+'0'`$\rightarrow$`s[0]=n/10+'0'` applied to line 8 of p_0 to produce the program p_0''. Firm mutation, after line 19, will be applied. Rather than slice on the whole variable space at line 19, the set $X = \{$`s[0]`$\}$ will be used.

A program produced by slicing p_0'' for `s[0]` is shown in Figure 14. From this slice it is clear that the mutant is killed by any input value n with the property that $n/10 \neq n\%10$. Thus, for example, the input value 7 will suffice.

Interestingly, most standard slicing algorithms would not produce this slice: they would not notice that the while loop cannot affect the value of `s[0]`. Constraint based slicing ([Krinke and Snelting, 1998]) offers one possible approach to extending slicing to deal with these types of cases.

The above process requires both p and p' to be sliced at n'. Thus, if m mutants are being considered, slicing must be performed $m + 1$ times. It would be preferable to slice p at n' and then mutate this slice. This can be done if conventional slicing is used and mutating and slicing commute: if slicing and then mutating is the same as mutating and then slicing. Conditions under which this is the case, and ways in which p can be augmented in order to ensure that this is the case, will be discussed in the following sections. It is important to note that as amorphous slicing is not syntax preserving it is often meaningless to talk about amorphous slicing and mutation commuting.

```
1    main()
     {
2    char s[MAX] ;
3    int n, i, sign ;
4    scanf("%d",n) ;
6    if (n<0)
7       n = -n ;
8    s[0] = n/10 + '0' ;
     }
```

Figure 14: The slice of p_0''

7.2 Commuting slicing and mutation

Under certain conditions, slicing and mutation commute. Where this is the case, it is possible to slice p at n' and mutate the resultant program.

A mutation operator shall be said to be *reference preserving* if it affects either a predicate or the RHS of an assignment and does not change the referenced variable set. The following holds for all current conventional static slicing algorithms.

Lemma 1 *Suppose p has been mutated by using a reference preserving operator that replaces the statement s at node n by a statement s'. Then $SS(m(p,n,s \rightarrow s'),n',X) = m(SS(p,n',X),n,s \rightarrow s')$.*

This follows from the fact that slicing *algorithms* ([Danicic et al., 1995, Horwitz et al., 1990]) work on an abstraction of the program which does not distinguish between expressions with identical sets of referenced variables.

□

It should be noted that this result only holds for algorithms which behave identically for expressions with the same referenced variable set. While this is true for published approaches to conventional static slicing ([Danicic et al., 1995, Horwitz et al., 1990, Weiser, 1984]), it is not true for the *definition* of conventional slicing nor is it generally true for amorphous slicing algorithms. For example, the slice on the final value of `x` for the code fragment `x = x+1`, is the whole program. However, if this program is replaced by the mutant `x = x+0`, the referenced variables remain invariant, while the slice becomes the empty program.

Each of the five operators in E-selective is actually a set of operators. Thus, for example, AOR contains mutation operators such as replacing the arithmetic operator `+` by the arithmetic operator `-`. Most operators in the E-selective set are reference preserving. Thus, if selective mutation using E-selective is applied and slices are constructed using the standard algorithms, mutation and slicing usually commute.

The following shows how this property can be used to reduce the number of equivalent mutants produced.

Corollary 1 *Suppose the node n is not contained in $SS(p,n',X)$. When using firm mutation, considering the values of the variables in X at n', the application of a reference preserving mutation operator to n is guaranteed to generate an equivalent mutant.*

Thus, it is possible to determine in advance that certain mutants are equivalent. This allows the tester to reduce the number of equivalent mutants produced.

Voas and McGraw [Voas and McGraw, 1998] suggest that slicing could be used to identify statements for which it would be pointless to apply mutation. That is, if the slice for some point n in the program does not contain a statement s, then this statement could be replaced with any mutation without affecting the point n. Thus, any mutation is guaranteed to be equivalent. Interestingly, this argument rests on a similar assumption to the 'referenced-variable preserving mutation' assumption. Strictly speaking, such mutations are only pointless if they only serve to affect referenced variables and do not affect defined variables. Those which affect defined variables, by contrast, may alter the slice, making Voas and McGraw's observation invalid. For example, consider the fragment below

```
1   x = y+1;
2   z = z*2;
3   p = z-4;
```

Suppose that line 3 is to be probed for the effect of a mutation at line one. That is, the test for equivalence will be:

> "does the value of `p` after line 3 change when we mutate line 1?"

The slice on `p` at the end of the fragment is $\{2, 3\}$, indicating that line one does not affect the value of `p` at line 3. If line one is mutated in such a way as to replace the expression on the right hand side of line 1, the slice remains unaffected and the mutant is equivalent. However if statement one is replaced by `z = y+1` (one of the SVR mutations) then the slice changes to $\{1, 2, 3\}$ and the mutant is not equivalent

Voas and McGraw's observation is, however, extremely useful in the context of defining suitable sets of mutants for selective mutation, where the particular selection mechanism is tailored to suit the program under analysis.

7.3 Augmenting p so that mutating and slicing commute

Where mutation operators that are not reference preserving are being used, mutation may change the set of statements that are in the slice. This is because of the change in the variables mentioned at n. It is possible, however, to augment p and p' in order to avoid this problem.

Suppose p' has been formed by mutating p at n. Suppose also that firm mutation, at node n' with variable set X, is being used and that p' is live. In order to ensure that slicing and mutation commute it is sufficient to

1. alter both p and p' by adding a new node $pred(n)$ and a new variable `z`
2. at $pred(n)$, in p and p', assign to `z` a tuple of variables that constitutes the difference between the referenced variable sets at n for p and p'.
3. add `z` to the slice criterion at n'

Suppose the mutation affects the RHS of an assignment or a predicate and involves replacing $\mathtt{e}$ by $\mathtt{e}'$. Then, in p and p', a tuple containing the variables in the symmetric difference of $ref(\mathtt{e})$ and $ref(\mathtt{e}')$ should be assigned to `z` at $pred(n)$. If the mutation involves replacing the variable `y` on the LHS of an assignment at n by `x` then the assignment $\mathtt{z} = (\mathtt{x}, \mathtt{y})$ should be added to p and p' at $pred(n)$.

Thus, when considering mutations that are not reference preserving, it is possible to augment the program in order to guarantee that Lemma 1 and Corollary 1 follow. This allows analysis that is capable of demonstrating that certain mutations, that are not reference preserving, are guaranteed to produce equivalent mutants.

7.4 Strong Mutation

The approaches outlined for firm mutation can also be applied to the two types of strong mutation. This is because both may be seen as special cases of firm mutation. Strong state mutation is firm mutation with $n = \omega$. Strong output mutation is firm mutation with $n = \omega$ and $X = \{\phi\}$.

8 Related Work

Voas and McGraw were the first authors to suggest the application of slicing to mutation testing [Voas and McGraw, 1998]. They observed that (conventional) slicing can identify points unaffected by some statement and that these make poor candidates[2] for fault injection.

Until recently, detection of equivalent mutants has been carried out 'by hand', as no automated tools exist to assist with the process. Naturally, such a labour-intensive task is prohibitively expensive. This has meant that, despite the efficacy demonstrated in empirical studies, mutation testing has remained solely a research topic.

Fortunately, recent work has lead to a partial automation of the process of detecting equivalent mutants. Although the problem is not generally decidable, there are many cases where an automated tool *is* capable of detecting equivalence. If this threshold can be sufficiently raised, with only a small number of mutants remaining for human checking, then this may open up the way for mutation to move from research labs into industry.

In 1994 Offutt and Craft ([Offutt and Craft, 1994]) implemented and evaluated Baldwin and Sayward's ([Baldwin and Sayward, 1979]) heuristics for detecting equivalent mutants. The idea is that compiler optimisation techniques create equivalent programs using small 'mutations', and so these optimising transformations can be thought of as 'idioms' which can be detected. Baldwin and Sayward suggested a set of six suitable optimising transformations. In Offutt and Craft's evaluation (once again on relatively small programs) the technique was capable of detecting approximately 10% of equivalent mutants.

Between 1994 and 1996 Offutt and Pan ([Offutt and Pan, 1996, Pan, 1994]) introduced a much improved equivalence detection approach, based on constraint solving. Empirical evaluation of a 'proof-of-concept' prototype shows that the new technique pushes the equivalence detection rate up to 48%, and there is cause to believe that the approach is capable of achieving much higher detection rates than this. The idea is to represent the problem of whether a mutation at n is equivalent as an inequality at the point of mutation. This point n is reached according to a path condition, which describes the condition, on the initial state, for the path leading to s to be executed. By conjoining the path condition to the inequality and using symbolic execution to re-write the resulting formula in terms of input variables, it is possible to represent mutant equivalence as a problem in constraint satisfaction; the mutant is equivalent if and only if the input constraint is unsatisfiable. Furthermore, if the input constraint is satisfiable then its satisfaction suggests suitable input values which will kill the mutant; namely, those which satisfy the constraint. Constraint solving is compared with slicing in the next section. The rest of this section briefly explains how constraint solving can be applied to the equivalent mutant detection problem.

For example, consider the fragment and SVR mutant ([Offutt et al., 1996a]) in figure 15. In order to execute the mutant the initial state must satisfy the condition (called a 'path condition'), `p>0`. Should this path be followed then it will be necessary for the values of `x` and `y` to differ at the mutation point, that is, in the initial state `x != y`. This is termed the 'necessary constraint' by Offutt

[2] As explained in section 7.2, this is slightly inaccurate, but the essential motivation behind this observation was the inspiration which lead to the work reported here.

`if (p>0)` `    i=x;`	`if (p>0)` `    i=y;`
(a) Program	(b) Mutant

Figure 15: A program and non-equivalent SVR mutant

`k=x;`
`if (p>0)`
`    i=k;`
	`k=x;`
	`if (p>0)`
	`    i=x;`

(a) Program	(b) Mutant

Figure 16: A program and an equivalent SVR mutant

et al. ([Offutt and Pan, 1996, Offutt and Pan, 1997]). Therefore, the (initial) condition under which the mutant will be killed is

$$\texttt{p} > \texttt{0} \;\&\&\; \texttt{x}! = \texttt{y}$$

In addition to being a boolean expression in the programming language under consideration (`C` in this case), the constraint can also be thought of as an existentially quantified first order predicate logic formula, namely:

$$\exists p, x, y \bullet p > 0 \;\wedge\; x \neq y$$

or as a characterisation of the set[3] of states in which the mutant will be killed:

$$\{\sigma \mid \sigma p > 0 \;\wedge\; \sigma x \neq \sigma y\}$$

In this case the mutant is not equivalent, there are many values of `p`, `x` and `y` which satisfy the formula. Moreover, any one of these solutions is a test case which kills the mutant, so in cases where computability issues do not prevent a constraint system from answering, the system will either respond with a test case which is guaranteed to kill the mutant or with a statement that the mutant is guaranteed to be equivalent.

Equivalent mutants produce constraints which are unsatisfiable. For example, consider the program fragment and another simple SVR mutation in figure 16. For this mutant, the path condition is `p>0`, while the necessary constraint at the mutation point is `x != k`, which gives `p>0 && x != x` in the initial state.

Offutt and Pan evaluated the technique using a constraint satisfaction algorithm, which employed three very simple rules for detecting unsatisfiable constraints. On a suite of eleven programs the system was able to detect 48% of equivalent mutants (on seven of the eleven the detection rate was over 60%). These results are extremely encouraging, as the system used so little theorem proving ability to decide upon the question of constraint satisfaction.

[3]If this set is empty, then the mutant cannot be killed — the constraint cannot be satisfied and so the mutant is equivalent.

More recently Offutt et al ([Offutt et al., 1999]) introduced a technique for generating test data based on the computation of set expressions in place of constraints. In this approach a set expression is used to represent the possible values of a variable at some point in the computation. The approach has been used to generate test data to kill mutants, but it remains to be seen whether it will prove helpful in determining mutant equivalence.

The insertion of the necessary constraint is similar to our insertion of a variable `z` to capture the answer to the question as to whether or not the mutant is equivalent to the original. However, whereas the constraint propagation algorithm has only three possible outputs: equivalent, non-equivalent and 'do not know', the approach given in this paper has many outputs, each of which is a program which captures the answer to the question 'is the mutant equivalent'. Where a constraint propagation system would be powerful enough to produce the definite outputs 'equivalent' and 'non-equivalent' it would be reasonable to expect an equally powerful amorphous slicing algorithm to produce correspondingly definite programs `z = TRUE` and `z = FALSE`. However, in the cases 'in between' where a constraint system produces the answer 'may be' the slicing algorithm will produce a simplified program which answers the question and which is guaranteed to be at least as simple as (and probably a good deal more simple than) the original program.

The approach presented here thus provides a way to soften the impact of computability issues in analysing problems of this nature; where no precise answer can be provided the programming notation is used to provide an approximate answer. In the next section the way in which the constraint-based and slicing-based approach may complement is described.

Currently, implementations of amorphous slicing are at the level of 'proof-of-concept' systems, such as those described in ([Harman and Danicic, 1995]), ([Binkley, 1999]) and ([Harman et al., 1998]). These algorithms produce good amorphous slices in specialised domains. However, their quadratic complexity means that they cannot be expected to scale up to the problem of equivalent mutants for large programs[4]. However, more recently work has begun on an implementation of amorphous slicing using the Program Dependence Graph [Binkley, 1998]. The aim of this work is to produce realistic, efficient, scalable algorithms for amorphous slicing.

9 A Comparison of Constraint– and Amorphous Slicing–Based Approaches

This section explores the relationship between the two approaches of amorphous slicing and constraint solving in detection of equivalent mutants.

It is demonstrated that whenever the constraint-based approach is capable of determining whether a mutant is equivalent the slicing approach is also capable of doing this. This result follows immediately from the definition of amorphous slicing. It is then argued that the use of slicing offers some advantages over a purely constraint-based approach.

1. It is capable of demonstrating that the application of certain mutation operators, at particular points of the program, is *guaranteed* to generate equivalent mutants[5].

2. Where neither approach determines whether a mutant is equivalent, the slicing based approach provides the tester with a simplified program, reducing the remaining human effort in determining whether or not the mutant is equivalent.

[4]Although the algorithms are not without value, as mutation testing is a unit level testing technique and is therefore likely to be applied to small code units, requiring, in many cases, only small-scale intraprocedural slicing.

[5]An observation due to Voas and McGraw [Voas and McGraw, 1998].

It is also shown that amorphous slicing and constraint-based testing are closely related and that a sensible equivalent mutant prevention nd detection strategy would incorporate both.

9.1 The Amorphous Slicing Approach Subsumes the Constraint Based Approach

Amorphous slicing and constraint solving are of equal power in detecting equivalent mutants. The difference between the two approaches lies in the way each handles cases where it is not possible to decide whether the mutant is equivalent, where amorphous slicing may offer additional assistance over that available through the constraint based approach.

9.1.1 Equivalence of Detection Power

To see that the two approaches are of equivalent power with respect to equivalent mutant detection, consider the case where a constraint-based system has constructed the test data generation constraint C from the mutant p' of a program p. Suppose that the constraint solver is capable of proving that C is unsatisfiable. This means that the system is capable of detecting the equivalence of p' and p, and an approach to the equivalent mutant problem based on constraint solving would report this.

Given the same constraint solver an amorphous slicing system would be capable of using the provable unsatisfiability of the constraint C to transform p' into

$$\mathtt{z} = \mathtt{TRUE};$$

That is, an amorphous slice may be constructed using *any transformation* which preserves the effect of the original program (p') on the slicing criterion ($\{\mathtt{z}\}$) [Harman and Danicic, 1997].

9.2 Constraint Solving and Amorphous Slicing Use Similar Technologies

Both constraint solving and slicing use similar technologies. Consider the program fragment below:

```
x=y;
s=x+1;
k=k*2;
if(s>0)
  k=x+2
```

Suppose that this is mutated (using the SVR mutation operator) to

```
x=y;
s=x+1;
k=k*2;
if(s>0)
  k=y+2
```

The constraint for this mutant consists of a path condition $\mathtt{x} + 1 > 0$, which must be satisfied for the mutant to be reached in some execution and a 'necessary condition', $\mathtt{y} + 2 \neq \mathtt{y} + 2$ which must be satisfied in order for the mutant to be killed by some execution under weak mutation testing. The constraint to be tested is thus

$$\mathtt{x} + 1 > 0 \wedge \mathtt{y} + 2 \neq \mathtt{y} + 2$$

Clearly this constraint is unsatisfiable and a system using the approach suggested by Offutt and Pan[Offutt and Pan, 1996, Offutt and Pan, 1997] would be able to detect this.

Using the amorphous slicing approach advocated in this paper, a sequence of transformations would take place which would involve testing a similar constraint during the process of the transformation. The mutant would be instrumented with additional code to capture the question of mutation equivalence (inserting the assignments to z), giving

```
z = TRUE;
x=y;
s=x+1;
k=k*2;
if(s>0)
{ z = z && y+2 == x+2;
  k=y+2; }
```

Conventional (non-amorphous) end-slicing on {z} gives:

```
z = TRUE;
x=y;
s=x+1;
if(s>0)
  z = z && y+2 == x+2;
```

Constant propagation using Binkley's algorithm [Binkley, 1999] gives

```
z = TRUE;
if(s+1>0)
  z = TRUE && y+2 == y+2;
```

At this point boolean simplification is required [Harman et al., 1999, Harman et al., 1998]. This boolean simplification could be viewed as 'trivial theorem proving' — demonstrating the truth of a simple theorem. The theorem in question is essentially the constraint with which the constraint solver was faced, namely

$$\texttt{TRUE} \wedge \texttt{y} + 2 == \texttt{y} + 2$$

Having simplified this boolean expression to TRUE, the amorphous slice becomes

```
z = TRUE;
if(s+1>0)
  z = TRUE;
```

From this slice, a simple rule of transformation can be used to give

```
z = TRUE;
```

In this way the amorphous slicing approach has achieved the same result as constraint solving using similar technology. The only 'hard part' of the transformation (that which may work for some examples but not for others) is the boolean simplification. This requires the same level of sophistication as the constraint solver.

Now consider the situation where the constraint solver is unable to detect an equivalent mutant.

9.3 Using Constraint Solving and Slicing Together

Section 9.1 showed that constraint solvers could be used to construct amorphous slices. However, it would be unwise to use constraint solving simply in order to construct slices which offer the same power as a constraint testing solution. A better approach would use both slicing and constraint solving in concert as each offers complementary information.

Given the current state of the art in test data generation and equivalent mutant detection, a hybrid approach appears to be most suitable.

Initially, conventional syntax-preserving slicing might be used in order to avoid the generation of some equivalent mutants. A constraint solver or Dynamic Domain Reduction ([Offutt et al., 1999]) can be used to generate test data to kill some mutants and to prove some other set of mutants equivalent. The initial application of the constraint solver therefore partitions the set of mutants M into three subsets:

- M_k: Mutants killed by some test case
- M_e: Mutants determined to be equivalent
- M_s: Mutants which remain alive and which cannot be shown to be equivalent

Attention will now focus on the set M_s, which are termed "stubborn mutants" in this paper. Of these, some will most likely be equivalent and some will not, but which is which? Using amorphous slicing at this point will be unlikely to return the simplest possible program for any of the remaining mutants which are equivalent. If it were able to do so, then a similar argument to the one above, would lead to the conclusion that a constraint solver could have shown the associated constraint system to be unsatisfiable.

However, slicing will be able to advance the position of the tester in three ways:

- Some simplification will be performed on the mutants in M_s.
- The equivalence question will be phrased in the form of a program, rather than in the form of a complex logical formula.
- The program generated by amorphous slicing may be significantly smaller than the logical formula generated by the constraint-based approach.

This hybrid approach to mutation testing is depicted in Figure 17. There are three kinds of mutants.

1. Dead mutants (M_k), killed by a test case generated using either constraint solving or Dynamic Domain Reduction;
2. Equivalent Mutants (M_e) determined to be equivalent using constraint solving;
3. Stubborn Mutants (M_s), which may or may not be equivalent and require human analysis. These are sliced using Amorphous Slicing to reduce the human effort involved in determining whether or not they are equivalent.

Clearly it would be desirable to squash the stubborn mutant category as much as possible. However, the undecidability of the equivalent mutant problem means that the 'stubborn' category cannot always be empty.

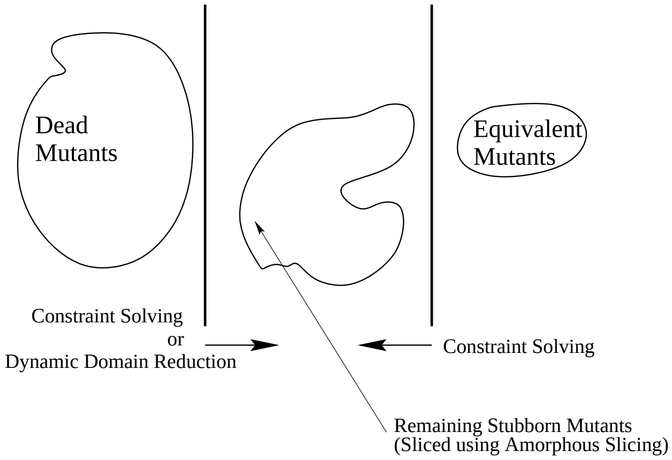

Figure 17: A Hybrid Approach to Mutation Testing

9.3.1 Using Slicing after Constraint Solving Fails

Where constraint solving is able neither to detect the equivalence of a mutant nor to generate test data to kill it, amorphous slicing may be helpful in simplifying the remaining task for the tester.

Consider the program fragment below

```
x = f(s);
y = g(s);
s=s+1;
if (s>0)
   p=x;
```

and an SVR mutant of this program

```
x = f(s);
y = g(s);
s=s+1;
if (s>0)
   p=y;
```

The necessity constraint to kill this mutant is $\exists s. f(s) \neq g(s)$. The path condition is $s + 1 > 0$. The constraint is[6]:

$$\exists s.(f(s) \neq g(s) \wedge s + 1 > 0)$$

If this constraint cannot be satisfied then the mutant is equivalent. Not surprisingly, no constraint solver could decide this question in general. If it could, then it would be capable of deciding whether two arbitrary functions were equivalent. That is, this constraint is a constraint schema in `f` and `g` which is not generally decidable.

Using amorphous slicing gives:

[6]This examples illustrates the way in which the path condition and necessity condition may not always be separable — they are bound by the same quantifier.

```
z = TRUE;
if (s+1>0)
   z = (f(s) === g(s));
```

The amorphous slice is simply a program, which the tester could analyse using any techniques available for analysis and testing of programs. The constraint is a logic formula which the tester could analyse using any techniques available for analysis of formulæ. Here the only question that remains is whether the program is easier to analyse or whether the formula is better.

9.3.2 Combinatorial Constraint Explosion

Suppose p has been mutated to form p' and automated analysis fails to determine whether p' is an equivalent mutant. In generating the constraint C the constraint-based method may have to analyse a number of the paths of the program. This is because the different paths may lead to different conditions on the variables of interest. Consider, for example, a program p represented below and a mutant p' produced by replacing `x` by `y` in the final statement.

```
 if P1 then S1 else S1';
 if P2 then S2 else S2';
...
 if Pn then Sn else Sn';
a=x+1;
```

Then the constraint-based method produces a constraint C while the slicing-based method produces a program p''. If C is unsatisfiable then p and p' are equivalent. Similarly, if p'' always ends with the variable `z` taking the value `TRUE`, p and p' are equivalent.

Suppose that the predicates `P` , ... , `Pn` mention variable `x` and each statement `S1`, `S1'` , ... , `Sn` , `Sn'` both uses and mentions `x`. The mutant p' is equivalent if and only if the values of `x` and `y` are always the same at the end of this section of code. But each path may represent a different computation, leading to different symbolic representations of `x` and `y` at this point. Where this is the case, in order to generate the constraints that determine whether the mutant is equivalent it may be necessary to generate a separate constraint for *each* path, generating at least 2^n constraints. Clearly, as n grows this becomes impractical. Even worse, where there are loops there may be an infinite number of paths.

Where a combinatorial explosion occurs, the constraint-based approach may not help. The slicing-based approach does not, however, suffer from this combinatorial explosion. Slicing may still produce a simplified program and thus assist the tester.

10 Conclusions

Mutation testing is a powerful, but expensive, test technique. Much of the expense is associated with the human analysis of mutants that are equivalent or difficult to kill. It has been shown that program slicing can reduce the effort involved in determining whether mutants are equivalent. In cases where a mutant is not equivalent, the process simplifies the mutant and thus helps the tester find test cases that kill it.

Where either firm or strong mutation is being used, program slicing may also be used in order to determine, in advance, that certain mutants are equivalent. This allows the tester to reduce the number of equivalent mutants generated.

The process of analysing a mutant has been phrased in terms of comparing the effect of a program and its mutant on certain variables. Here only the semantics of the program and the mutant are important: the syntax is irrelevant. This suggests the application of amorphous, rather than conventional, slicing as amorphous slicing often leads to slices that are considerably thinner than those produced by conventional slicing.

References

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

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

[Aho et al., 1986] Aho, A. V., Sethi, R., and Ullman, J. D. (1986). *Compilers: Principles, techniques and tools.* Addison Wesley.

[Baldwin and Sayward, 1979] Baldwin, D. and Sayward, F. (1979). Heuristics for determining equivalence of program mutations. *Research Report 276, Department of Computer Science, Yale University.*

[Ball and Horwitz, 1993] Ball, T. and Horwitz, S. (1993). Slicing programs with arbitrary control–flow. In Fritzson, P., editor, 1^{st} *Conference on Automated Algorithmic Debugging*, pages 206–222. Springer. Also available as Uniersity of Wisconsin–Madison, technical report (in extended form), TR-1128, December, 1992.

[Binkley, 1998] Binkley, D. (1998). Private communication.

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

[Binkley and Gallagher, 1996] Binkley, D. W. and Gallagher, K. B. (1996). Program slicing. In Zelkowitz, M., editor, *Advances of Computing, Volume 43.* Academic Press.

[Budd, 1981] Budd, T. A. (1981). Mutation analysis: Ideas, examples, problems and prospects. In *Proceedings of the Summer School on Computer Program Testing*, pages 129–148, Sogesta.

[Canfora et al., 1998] Canfora, G., Cimitile, A., and De Lucia, A. (to appear 1998). Conditioned program slicing. In Harman, M. and Gallagher, K., editors, *Journal of Information and Software Technology Special Issue on Program Slicing*, volume 40. Elsevier.

[Cheng, 1993] Cheng, J. (1993). Slicing concurrent programs – a graph–theoretical approach. In Fritzson, P., editor, 1^{st} *Automatic Algorithmic Debugging Conference (AADEGUB'93)*, pages 223–240. Appears as Springer Lecture Notes in Computer Science vol 749.

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

[Danicic et al., 1995] Danicic, S., Harman, M., and Sivagurunathan, Y. (1995). A parallel algorithm for static program slicing. *Information Processing Letters*, 56(6):307–313.

[DeMillo et al., 1978] DeMillo, R. A., Lipton, R. J., and Sayward, F. G. (1978). Hints on test data selection: Help for the practical programmer. *IEEE Computer*, 11:31–41.

[Ernst, 1994] Ernst, M. D. (1994). Practical fine–grained static slicing of optimised code. Technical Report MSR-TR-94-14, Microsoft research, Redmond, WA.

[Frankl et al., 1997] Frankl, P. G., Weiss, S. N., and Hu, C. (1997). All-uses vs mutation testing: An experimental comparison of effectiveness. *Journal of Systems Software*, 38:235–253.

[Hamlet, 1977] Hamlet, R. G. (1977). Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, 3:279–290.

[Harman and Danicic, 1995] Harman, M. and Danicic, S. (1995). Using program slicing to simplify testing. *Journal of Software Testing, Verification and Reliability*, 5:143–162.

[Harman and Danicic, 1997] Harman, M. and Danicic, S. (1997). Amorphous program slicing. In 5^{th} *IEEE Internation Workshop on Program Comprehesion (IWPC'97)*, pages 70–79, Dearborn, Michigan, USA.

[Harman and Danicic, 1998] Harman, M. and Danicic, S. (1998). A new algorithm for slicing unstructured programs. *Journal of Software Maintenance.*

[Harman et al., 1998] Harman, M., Danicic, S., and Sivagurunathan, Y. (1998). Analysis of dynamic memory access using amorphous slicing. In *IEEE International Conference on Software Maintenance (ICSM'98)*, Washington DC, USA. To appear.

[Harman et al., 1999] Harman, M., Fox, C., Hierons, R., Binkley, D., and Danicic, S. (1999). Program simplification as a means of approximating undecidable propositions. In 7^{th} *IEEE Internation Workshop on Program Comprehesion (IWPC'99)*, page to appear, Pittsburgh, PA, USA. IEEE Computer Society Press, Los Alamitos, California, USA.

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

[Howden, 1982] Howden, W. E. (1982). Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, 8:371–379.

[Kernighan and Ritchie, 1988] Kernighan, B. W. and Ritchie, D. M. (1988). *The C programming language.* Prentice Hall. Second Edition, (ANSI C).

[King and Offutt, 1991] King, K. N. and Offutt, A. J. (1991). A FORTRAN language system for mutation-based software testing. *Software Practice and Experience*, 21:686–718.

[Korel, 1995] Korel, B. (1995). Computation of dynamic slices for programs with arbitrary control flow. In Ducassé, M., editor, 2^{nd} *International Workshop on Automated Algorithmic Debugging (AADEBUG'95)*, Saint–Malo, France.

[Korel and Rilling, 1998] Korel, B. and Rilling, J. (to appear 1998). Dynamic program slicing methods. In Harman, M. and Gallagher, K., editors, *Journal of Information and Software Technology Special Issue on Program Slicing*, volume 40.

[Krinke, 1998] Krinke, J. (1998). Static slicing of threaded programs. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'98)*, pages 35–42, Montreal, Canada.

[Krinke and Snelting, 1998] Krinke, J. and Snelting, G. (to appear 1998). Validation of measurement software as an application of slicing and constraint solving. In Harman, M. and Gallagher, K., editors, *Journal of Information and Software Technology Special Issue on Program Slicing*, volume 40.

[Larsen and Harrold, 1996] Larsen, L. D. and Harrold, M. J. (1996). Slicing object-oriented software. In *Proceedings of the 18th International Conference on Software Engineering*, pages 495–505, Berlin.

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

[Lyle and Binkley, 1993] Lyle, J. R. and Binkley, D. (1993). Program slicing in the presence of pointers. In *Foundations of Software Engineering.*

[Offutt, 1992] Offutt, A. J. (1992). Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering Methodology*, 1:3–18.

[Offutt and Craft, 1994] Offutt, A. J. and Craft, W. M. (1994). Using compiler optimization techniques to detect equivalent mutants. *The Journal of Software Testing, Verification, and Reliability*, 4:131–154.

[Offutt et al., 1996a] Offutt, A. J., Lee, A., Rothermel, G., Untch, R. H., and Zapf, C. (1996a). An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering Methodology*, 5:99–118.

[Offutt and Pan, 1996] Offutt, A. J. and Pan, J. (1996). Detecting equivalent mutants and the feasible path problem. In *Annual Conference on Computer Assurance (COMPASS 96), IEEE Computer Society Press*, pages 224–236, Gaithersburg, MD.

[Offutt et al., 1996b] Offutt, A. J., Pan, J., Tewary, K., and Zhang, T. (1996b). An experimental evalutation of data flow and mutation testing. *Software - Practice and Experience*, 26:165–176.

[Offutt et al., 1999] Offutt, J., Jin, Z., and Pan, J. (1999). The dynamic domain reduction approach to test data generation. *Software – Practice and Experience.* to appear.

[Offutt and Pan, 1997] Offutt, J. and Pan, J. (1997). Automatically detecting equivalent mutants and infeasible paths. *Software Testing, Verification, and Reliability*, 7(3):165–192.

[Pan, 1994] Pan, J. (1994). Using constraints to detect equivalant mutants. Master's thesis, George Mason University.

[Tip, 1994] Tip, F. (1994). A survey of program slicing techniques. Technical Report CS-R9438, Centrum voor Wiskunde en Informatica, Amsterdam.

[Tip et al., 1996] Tip, F., Choi, J.-D., Field, J., and Ramalingham, G. (1996). Slicing class hierarchies in C++. In *Proceedings of the* 11^{th} *conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'96)*, San Jose.

[Venkatesh, 1991] Venkatesh, G. A. (1991). The semantic approach to program slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 26–28, Toronto, Canada. Proceedings in *SIGPLAN Notices*, 26(6), pp.107–119, 1991.

[Voas and McGraw, 1998] Voas, J. and McGraw, G. (1998). *Software Fault Injection.* Wiley.

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

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

[Woodward, 1993] Woodward, M. R. (1993). Mutation testing - its origin and evolution. *Information and Software Technology*, 35:163–169.

[Woodward and Halewood, 1988] Woodward, M. R. and Halewood, K. (1988). From weak to strong, dead or alive? an analysis of some mutation testing issues. In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, Banff, Canada.