The Relationship Between Program Dependence and Mutation Analysis

Mark Harman,
Rob Hierons,
Brunel University,
Uxbridge, Middlesex,
UB8 3PH, UK.
Tel: +44 (0)1895 274 000
Fax: +44 (0)1895 251 686
`Mark.Harman@brunel.ac.uk`
`Rob.Hierons@brunel.ac.uk`

Sebastian Danicic,
Goldsmiths College,
University of London,
New Cross,
London SE14 6NW, UK.
Tel: +44 (0)20 7919 7856
Fax: +44 (0)20 7919 7853
`Sebastian@mcs.gold.ac.uk`

Keywords: Slicing, Dependence Analysis, Test Data Generation, Equivalent Mutant Detection

Abstract

This paper presents some connections between dependence analysis and mutation testing. Specifically, dependence analysis can be applied to two problems in mutation testing, captured by the questions:

1. How do we avoid the creation of equivalent mutants?
2. How do we generate test data that kills non-equivalent mutants?

The theoretical connections described here suggest ways in which a dependence analysis tool might be used, in combination with existing tools for mutation testing, for test-data generation and equivalent-mutant detection.

In this paper the variable orientated, fine grained dependence framework of Jackson and Rollins is used to achieve these two goals. This framework of dependence analysis appears to be better suited to mutation testing than the more traditional, Program Dependence Graph (PDG) approach, used in slicing and other forms of program analysis.

The relationship between dependence analysis and mutation testing is used to defined an augmented mutation testing process, with starts and ends with dependence analysis phases. The pre-analysis removes a class of equivalent mutants from further analysis, while the post-analysis phase is used to simplify the human effort required to study the few mutants that evade the automated phases of the process.

1 Introduction

In mutation testing, a program is mutated to create a mutant by a small syntactic change. These changes are likely to make the mutant behave differently when compared to the original. Such behavioural differences can be found by running the program with an input which reveals the differing behaviour. An execution which does this 'kills' the mutant in the nomenclature of mutation testing[1]. Running a mutant is 'testing' the original program in the sense that bugs in the original program are simulated by the mutations. Broadly speaking, a test set which kills many mutants is considered to be better at finding bugs than one which does not. In this way mutation testing can be used to assess the effectiveness of a test set or to help in the construction of an effective test set.

Mutation testing has been shown to be highly effective in empirical studies [6, 23]. It is also theoretically appealing because it is tailored to the program under test and because it can be used to emulate the effect of other coverage based test adequacy criteria. However, there is a downside. Typically, many mutants can be created from even the simplest original program under test and some of these may be equivalent to the original program semantically (though they differ syntactically). The problem of finding a set of test data which kills all non-equivalent mutants is also far from trivial.

Two currently important problems for mutation testing are thus summed up as two questions. The first question is 'How do we avoid the creation of equivalent mutants?' The second question is 'How do we (automatically) generate test data that kills all non-equivalent mutants?'.

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. The automatic generation of test data which satisfies some test data adequacy criterion is known to be a hard problem and the mutation-inspired adequacy criteria

[1]Mutants are always executed on test cases for which this original program behaves correctly. If a test case reveals an error, the original program must be corrected and the mutation analysis restarted *ab initio*.

are no exception.

This paper shows how dependency analysis can be used to attack these two problems. The rest of the paper is organised as follows. Sections 2 and 3 introduces the forms of mutation testing and dependence considered in this paper. Section 4 establishes the theoretical connection between these forms of mutation testing and program dependence analysis, showing how it can be used to support test data generation and equivalent mutant detection. Section 5 provides examples which illustrate the claims in the preceding section. Section 6 presents related work on compiler-optimisation and constraint-based approaches to test data generation and equivalent mutation detection. This suggests a combined mutation testing process presented in Section 7. The process combines dependence analysis and constraint based approaches to mutation testing. Section 8 concludes with directions for future work.

2 Forms of Mutation Testing

2.1 Strong, Weak and Firm Mutation

Mutants are always created in the same way: a single simple syntactic alternation is made to a node, n, of the original program. Mutants are 'killed' when a test case reveals mutant behaviour which differs from that of the original program. In order to kill a mutant, its behaviour must therefore be inspected in some way. There are three approaches to the way in which a mutant is inspected, known as 'strong', 'firm' and 'weak' mutation.

In the original form of mutation testing, now called *strong mutation testing*, input σ kills a mutant p' of a program p if p and p' produce different output when executed on σ.

Woodward and Halewood [32] suggested using the final state of the program in place of the output. This is a helpful generalisation because the output sequence can be considered to be denoted by a variable. Their definition of this 'state-based' approach to mutation subsumes the traditional 'output based' approach. It also simplifies the formal exposition presented later. Hereinafter, we shall, without loss of generality, assume that testing a mutant consists of inspecting some set of variables, which we call the *inspection set*.

Strong mutants are theoretically hard to kill because the effect of the mutation can be lost before the program reaches the final state.

Under *weak mutation testing* [15], p' is killed by input σ on inspection set I if the execution of p and p' on σ leads to different values of some $\mathtt{x} \in I$ immediately after *some* execution of the mutation point at node n.

Woodward and Halewood [32] introduce *firm mutation testing* as a compromise between strong and weak mutation. In firm mutation, p is mutated at some point n to form p'. The original program, p and the mutant, p' are compared on some inspection set I at some node i. Both weak mutation and strong mutation are special cases of firm mutation: in weak mutation $i = n$ and in strong mutation i is the exit node of the program. Firm mutation testing therefore subsumes strong and weak mutation. Hereinafter it will be assumed, without loss of generality, that firm mutation is being used. The point i will be termed the *probe point*.

The view of mutation testing adopted in this paper is summarised in Definition 2.1 below.

Definition 2.1 (Mutant)
In this paper, a *mutant* will be considered to be a program p', constructed from a program p by a single syntactic change affecting node n of p. The mutant p' will be tested by executing p and p' on input σ and checking the values of the variables in some inspection set I at some probe point i. If the value of any variable in I at i is different for the execution of p and p' then the mutant p' is killed.

Definition 2.2 (Equivalence)
Equivalence of a mutant p' is dependent upon the choice of probe set and probe point, as well as the particular mutation applied to the original program. If p' is not killed by any possible input for a inspection set I at a point i, we shall say that p' is *equivalent* to p with respect to I and i. If, for all inspection sets, p' is not killed by any possible input we shall say that p' is equivalent at i. If for all inspection sets I and for all mutation points i, a mutant p' is equivalent with respect to I and i, then we shall simply say that p' is *equivalent*.

2.2 Reference–Preserving Mutation Operators

In considering the connections between dependence analysis and mutation testing, the set of reference-preserving mutation operators are an important class. A mutation operator is reference-preserving if it does not change the set of referenced variables of any node of any program it mutates. Referenced variables are those mentioned in expressions, either on the RHS of an assignment node or as variables mentioned in the boolean expression of a predicate node.

The concept of reference preserving can be illustrated by considering the Mothra system [18]. Mothra contains 22 meta mutation operators. We call the Mothra operators 'meta operators', because each defines a family of individual primitive mutation operators. The Mothra operators are well known and will not be described in detail here. However, we shall consider a few operators to illustrate reference preservation.

`ABS` composes an expression with the absolute value function, while `CRP` replaces a source constant with a different constant. Neither of these can add or remove a variable reference and they are thus reference preserving. `SVR` replaces a scalar variable with an alternative and so is clearly not reference preserving when applied to the right-hand side of an assignment or to the boolean expression of a predicate. `SDL` deletes a statement and so will only be reference preserving in the few rare cases where the deleted statement references no variables.

`AOR` and `ROR` stand for Arithmetic and Relational Operator Replacement respectively. Each includes primitive mutation operators `LEFTOP` and `RIGHTOP`, which delete the left and right operand of the binary expression they are applied to. While the bona fide operator replacement primitive mutation operators in `AOR` and `ROR` meta mutation sets are reference preserving, `LEFTOP` and `RIGHTOP` will not generally be reference preserving.

In dependence analysis, programs which are identical up to referenced variable sets contain identical dependencies under the approximation used in most dependence analysis algorithms [4]. This means that when reference-preserving mutations are considered, it will not be necessary to re-compute dependence information from the original program. As many mutation operators are reference preserving, this makes the combination of dependence analysis and mutation testing more efficient than it might otherwise be. Where dependence information needs to be re-computed, an incremental approach [28, 29, 33, 34] will be highly applicable since, by definition, mutation involves only very minor changes.

3 Dependence Analysis

Jackson and Rollins [16] introduce a form of program dependence, which we shall call 'JR–Dependence'. As we will show, JR–dependence is a useful aid in tackling various problems in mutation testing, including reducing the input set in the search for mutant killing states.

JR–dependence is 'finer grained' than the Program Dependence Graph (PDG) [5, 13, 14] often used in program analysis, and particularly in slicing [7, 10, 31]. It allows us to relate (variable, node) pairs rather than simply to relate nodes. This allows us to say that the definition of a variable `x` at node n_1 can influence the value of a variable `y` at node n_2, rather than merely saying that node n_1 can influence node n_2.

JR–dependence is better suited to the analysis of mutants, because we shall want to know which sets of variables are important at certain points in the program, not merely which nodes are important. Specifically, we shall want to know the set of variables which can be used to kill a mutant and which set of variables store values which cannot be used to kill a mutant.

In the JR–approach, dependence relations are constructed from two basic dependence relations, du and ucd, which we now explain.

Every atomic program statement (node of the CFG) is represented as a relation, du, between pairs. For example, an assignment statement at node 5:-

```
5:   x:=y+z;
```

has

$$du = \{((x, 5), (y, 5)), ((x, 5), (z, 5))\}$$

Node 5 defines $\{$`x`$\}$ and uses $\{$`y`, `z`$\}$. The relation du is thus *internal* in the sense that it relates variables of the *same* node. In the above example, the value of `x` defined at node 5 depends upon the value of `y` referenced at node 5. Hence there is a relationship from $(x, 5)$ to $(y, 5)$ in du. Similarly, the value of `x` at node 5 depends upon the value of `z` at node 5, so there is also a relationship from $(x, 5)$ to $(z, 5)$ in du.

A du relation will always mention the same node in all relationships. That is

$$((v_1, n_1), (v_2, n_2)) \in du \quad \Rightarrow n_1 = n_2$$

Thus relationships in du are 'intranode' but 'intervariable'. It is because the relationship is intranode that we call it an 'internal relation'.

The external relationship between nodes is expressed by a relation, ucd, which captures the control and data dependence relations of the PDG (augmented to allow variables to be identified in addition to nodes). For data dependence, ucd expresses the connection between the use of a variable at node n_i with the definition of the same variable at a different node n_j. The ucd relation is also used to express control dependence between (variable, node) pairs. A control dependence[2] exists from a node x to a node y if y chooses whether or not node x gets executed. In such a control dependence, the node y will always be a predicate.

Control dependence is achieved in ucd using a coding, in which an assigned variable depends (in du) upon a special variable ε, the 'execution variable'. The ε variable of node n is dependent (in ucd) upon a variable τ, which is defined in all nodes which control n. In this way control dependence is captured in ucd as if it were a data dependence upon special variables τ and ε.

By composing du and ucd in various ways, and restricting the resulting domains and ranges, we can use JR–notation to express, and hence to compute[3] the dependencies which will be of interest in program analysis and, in particular, in mutation analysis.

Jackson and Rollins define several compositions. Here only one will be required: the UD relation.

$$UD = ucd \circ (du \circ ucd)^*$$

Here, R^* is the reflexive transitive closure of R and relational composition is from left to right i.e.

$$r_1 \circ r_2 = \{(x, y) | \exists z \text{ such that } (x, z) \in r_1 \text{ and } (z, y) \in r_2\}$$

By closing the relation $du \circ ucd$, the propagation of dependence information is achieved. The composition of the closure with ucd is merely required to ensure that the resulting relation, UD is of an appropriate type. That is, one from uses of variables to the definitions upon which they depend.

[2]More precisely, a control dependence exists from node x to node y if x is always executed when one branch of node y is executed, but there is also a path from y to the exit which does not contain x[5].

[3]The JR–notation is written in a functional style. This provides both a relatively high level 'specification orientated' style of notation, and a relatively simple way of prototyping implementations in a functional language such as `ML`.

4 The Relationship Between JR–Dependence and Mutation Testing

This section establishes the relationship between JR–Dependence and Mutation Testing. It is used as the theoretical basis for the augmented mutation testing process proposed in section 7.

Let p be the original program under test and let p' be a mutant of p created by mutating node n of p. Let the probe point be i and the inspection set be I and let Var be the set of all variables. Let $start$ be the entry node of the CFG.

To kill p' we are looking for a state σ which *distinguishes* between p and p' with respect to (i, I). Some initial program variables will be interesting and some will not. Those which are uninteresting can be varied arbitrarily without killing the mutant p'. That is, the value of these variables cannot be used to produce different behaviour in the mutant and the original program. Those which are interesting may potentially affect the outcome of execution of the mutant in a way that will allow us to determine that it is different from the original.

Clearly, such a set of variables will depend both upon the mutant and the program from which it is created. It will also depend upon the probe point and inspection set, as these are the lens through which the mutant and original are inspected. If a mutant is hard to kill it may be because it is equivalent. This may occur either because the mutant is so similar to the original that no inspection can detect it, or simply because the (inspection-set, probe-point) pair are inappropriate (the lens is too weak).

Using JR–notation, we shall define the set T of 'interesting variables' which is such that any variable outside this set need not vary in our test set. To generate test data to kill a mutant, we should be sure to consider only variables in the set T. If T is empty then the mutant is equivalent, and should not be analysed further (or the probe point and/or inspection set should be varied).

Let $\mathbf{ran}(\mathrm{R})$ be the range of a relation R. With a slight abuse of notation we shall refer to variables in this range. Strictly these are the variables in the first element of each pair in the range.

Using the JR–dependence notation [16], consider the relation

$$R = (I \times \{i\}) \lhd UD_p \rhd (Var \times \{\text{start}\})$$

Any variable outside $X = \mathbf{ran}(\mathrm{R})$ cannot possibly affect I at i. i.e. Any two initial states differing only on variables outside $\mathbf{ran}(\mathrm{R})$ will always behave the same at (i, I).

Similarly,

$$R' = (I \times \{i\}) \lhd UD_{p'} \rhd (Var \times \{\text{start}\})$$

is such that any variable outside $X' = \mathbf{ran}(\mathrm{R})'$ cannot possibly affect I at i.

Therefore if we are looking for states which can kill the mutant, we only need consider states which differ on variables in $X \cup X'$.

$$\text{i.e. } T = X \cup X'$$

We can improve on this, however. A 'killing state' must also affect the mutant at node n, otherwise it cannot possibly distinguish between p and p'.

Now consider the relation

$$S = (Var \times \{n\}) \lhd UD_p \rhd (Var \times \{\text{start}\})$$

For program p, any two initial states differing only on variables outside $\mathbf{ran}(\mathrm{S})$ will always behave the same at n.

Similarly, let

$$S' = (Var \times \{n\}) \lhd UD_{p'} \rhd (Var \times \{\text{start}\})$$

For program p', any two initial states differing only on variables outside $\mathbf{ran}(\mathrm{S})'$ will always behave the same at n.

Let $Z = \mathbf{ran}(\mathrm{S})$ and $Z' = \mathbf{ran}(\mathrm{S})'$. Any two states which only differ outside $Z \cup Z'$ must behave the same *everywhere* with respect to p and p'.

So the space where we look for killing states can be limited further to states differing only on $(X \cup X') \cap (Z \cup Z')$.

$$\text{i.e. } T = (X \cup X') \cap (Z \cup Z')$$

Observe that choosing a variable z, which *is* in T does not guarantee that the mutant will be killed. This is not merely because the right value has to be chosen for the 'influencing' variable in order to kill the mutant. This lack of a guarantee also arises because it is sadly possible for dependence analysis to produce 'false positives'. All dependence analysis must be safe: it will always include a dependency if one exists. However, for the usual, well-known, decidability reasons, such analyses will not always fail to include a dependency where none exists.

Therefore, T is sufficient in the sense that any variable outside T cannot have a bearing on whether the mutant is killed. Therefore, although there is some imprecision inherent in dependence analysis, it is nonetheless possible to make the following two definite statements about T:

- In selecting test data, only the initial values of variables in T should be varied.
  All other variable values are definitely irrelevant.

- If no variable in T can be varied in the initial state, then the mutant is definitely equivalent.
  The most obvious way in which this can happen is for T to be empty. However, in certain domain-specific problems, there may be non-empty sets, T, for which no variable in T can be varied. For instance, the only variables in T may have values which depend upon the input to some physical-environment sensor, which cannot be easily set to an arbitrary value by the tester.

5 Examples

In this section we present a few simple motivating examples which illustrate the way in which the foregoing formal analysis can be used to help focus attention upon interesting variables

```
x = y+z;    /* mutation point */
⋮
x = 0;   /* re-initialise */
⋮
printf("%d",x);    /* probe point */
```

Figure 1: A set of Equivalent Mutants

when generating test data and how we can avoid the creation of certain classes of equivalent mutants.

5.1 Avoiding the Creation of Equivalent Mutants

Mutants which fail to propagate ‘corrupted data’ to the inspection set at the probe point will be equivalent and should be avoided.

For example, consider the program in Figure 1. In this program, the value of x at the mutation point fails to propagate to the probe point because it is killed on all paths from the mutation point to the probe point (in this simple example, there is only one such path).

In this case, equivalence arises because of the control flow structure of the original program. The re-initialisation of the variable x effectively destroys the previous value, and so any mutation analysis which places the mutation point on one side of the assignment and the probe point on the other, without affecting the control flow, will fail to detect any mutation which retains x as the defined variable. This situation (in which variables are re-initialised) is common in embedded systems, where the number of variables is not large. Paucity promotes re-use, leading to just this sort of re-initialisation.

In general, it is not easy to determine which variable values can reach a particular program point, and so it will not be easy to spot these equivalent mutants manually. Fortunately, dependence analysis is designed to automate the production of answers to just these sort of questions.

In the previous example, the mutant was equivalent because the value of a variable was set during computation. This form of program is likely to cause the creation of equivalent mutants in strong mutation testing, but less so in weak mutation testing where the value of the mutated variable does not have to propagate to the probe point. Mutants may also be equivalent, because the inspection set is poorly chosen. In such situations, either the mutation needs to be reconsidered or the (inspection set, probe point) pair should be altered.

For example, consider the program fragment in Figure 2. In this program fragment, the value of x defined at the first line, will not reach the printf statement at the last line because of the intervening initialisation. In this respect the example is similar to the previous one. However, in this example there is a difference. The value of x can reach the probe point through the variables y and z, which store a value dependent upon x and which are not

```
x = 42;   /* mutation point */
⋮    /* does not define or reference x */
if (z==2)
   y = x+1;
else z = x*x;
⋮
x = 0;     /* re-initialise */
⋮   /* does not define y or z */
printf("%d",x);     /* probe point */
```

Figure 2: Poor Choice of Inspection Set

re-initialised on the way to the probe point. Therefore, a better choice of inspection set would be $\{\texttt{y}, \texttt{z}\}$ rather than $\{\texttt{x}\}$.

5.2 Generating Test Data to Kill Non-Equivalent Mutants

Consider the CFG in Figure 3. In this figure, code blocks indicated by question marks contain arbitrary code, but do not reference or define the variables $\{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}, \texttt{x}, \texttt{y}, \texttt{z}\}$.

Suppose the predicate p4 is mutated using a reference-preserving mutation. Clearly, this mutation can only be killed if a suitable set of values is chosen for the input variables to the program. The inputs in this case, are, a, b, c, d, x, y and z. If each variable is a 16-bit integer then the input is therefore $16 \times 7 = 112$ bits long. There are thus $2^{112} \approx 10^{33}$ potential values in the space to be searched. However, it turns out that *only* the variables a and b can affect the values of b and c at the predicate node p4 and therefore, only these two variables need be considered when attempting to find an input to kill the mutant. This observation reduces the search space to $2^{32} \approx 10^{9}$, a reduction of order 22 in the number of possible test cases to consider.

Determining that the values of b and c at node p4 are affected by only the variables a and b in the initial state is precisely the role of dependence analysis. This analysis does not determine the input values needed to kill the mutant. That (harder) problem is the preserve of constraint solving approaches [21, 22, 24]. However, dependence analysis can dramatically reduce the space that needs to be considered in order to determine the crucial killing values and can be applied when constraint-based techniques fail to give an answer.

Reducing search space size in this way may also find application in the area of search-based test data generation systems, such as those based on genetic algorithms [17, 25, 26, 30].

6 Related Work

Voas and McGraw [27] were the first to suggest that dependence analysis (specifically program slicing) might be useful in mu-

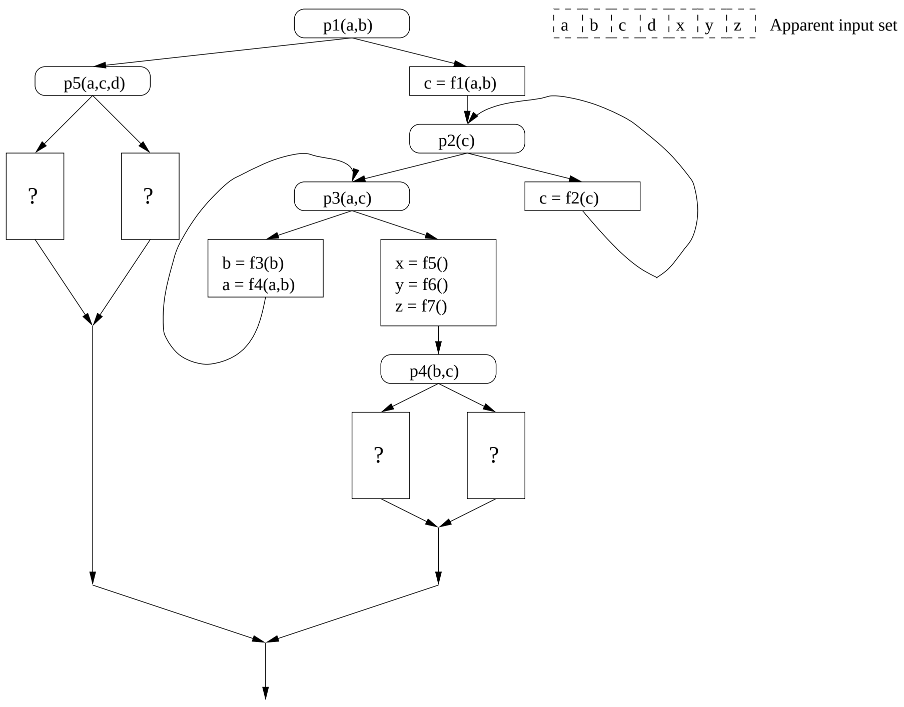


Figure 3: Generating Test Data to Kill a Mutant

tation analysis. They observed that syntax-preserving slicing can identify points unaffected by some statement and that these make poor candidates for fault injection, because the mutants so-created are ‘guaranteed’ to be equivalent. This is slightly inaccurate [12]. The observation becomes true when only reference preserving mutations are considered.

The present authors [12] developed the initial proposal of Voas and McGraw, showing how amorphous slicing [3, 8, 9, 11] can be used to assist human analysis of particularly stubborn mutants. This work considered slicing as a means of assisting the human analysis, rather than as a way of determining equivalence.

Offut et al. [19, 20, 21, 23, 22] have considered the problems of test data generation and equivalent mutant detection extensively, using both constraint-based techniques and compiler optimisations.

Initial work focussed on compiler optimisations [19]. The idea was originally proposed by Baldwin and Sayward [2] as early as 1979, but remained unexplored until 1994, when Offutt and Craft implemented a set of compiler-optimisation heuristics and evaluated them. The approach consists of looking at mutants which appear to implement traditional peep-hole compiler optimisations [1]. Such compiler optimisations were designed to create faster, but equivalent programs and so a mutant which implements a compiler optimisation will, by definition, be an equivalent mutant regardless of the probe point and inspection set. Offutt and Craft’s empirical study of this idea, indicated that while helpful, there was some way to go if the equivalent mutant problem was to be overcome. They found that the set of heuristics they implemented was able to detect about 10% of the equivalent mutants.

This work on compiler optimisation was improved upon in work by Offutt et al. [20, 21, 23, 22], which focussed on constraint-based approaches which seek to determine mutant equivalence by formulating a set of constraints. These constraints were originally used to generate test data (by solving the constraints). The more recent work also indicated that heuristics designed to determine when such a set of constraints fails to be satisfiable is also rather effective as a means of determining equivalence. Empirical studies showed that the approach could achieve a detection rate of about 50%.

Initially [21, 23, 22], the approach considered constraints over logical formulæ whose free variables were input variables to the program. Later work [20] used set-based constraints, where sets of intervals of possible input values were propagated through the program under test. Both these techniques are more fine grained than the work presented here. They are capable of providing values of test data, rather than simply identifying which variables are significant. This observation motivates the augmented mutation testing process which we propose in the next section.

7 A Dependence and Constraint Based Mutation Analysis Process

We do not propose dependence analysis as a replacement for constraint-based approaches. Rather we suggest that it should form an additional tool in the armory of the mutation analyst. In particular, where constraint based approaches have failed to produce test data, dependence analysis will help to focus the manual search for test data by constraining the input domain to the interesting variables. Furthermore, an initial dependence analysis may help to avoid the creation of certain classes of equivalent mutants, which would save later effort using a constraint-based system.

This symbiotic relationship is now developed into a process for mutation testing which combines dependence analysis, amorphous slicing and constraint-based test data generation and constraint based detection of equivalent mutants.

The process is depicted in Figure 4. The first phase is the creation of mutants. The process proposed here is independent of the approach taken to mutant creation. The next phase consists of detecting equivalent mutants using dependence analysis. This is a natural next phase as dependence analysis is less computationally expensive than constraint based approaches. This means that those mutants which dependence analysis can detect as equivalent are quickly disposed of. The next phase is the constraint-based approach developed by Offutt et al. This generates test data which kills some mutants and determines others to be equivalent.

The mutants which remain are those which are not determined to be equivalent by either dependence analysis or by constraint based analysis. These we call 'stubborn mutants'. These may be equivalent, but our two technologies of dependence analysis and constraint-based analysis simply proved too weak to detect them.

Stubborn mutants will ultimately have to be considered by a human. However, before the human is forced to look at the program code of a mutant, two more phases of automatic processing take place. The first of these is amorphous slicing (as described in more detail in [12]). This produces a simplified program tailored to the question of whether or not the mutant is equivalent. The second is domain reduction which can be achieved using the JR–dependence approach as described in this paper.

Thus the human mutation tester is finally presented with a set of stubborn mutants to analyse. Each is simplified using amorphous slicing and the human is guided in their choice of test data to consider by JR–dependence analysis.

8 Conclusions and Future Work

This paper has shown how dependence-based analysis can be used to assist in the detection of equivalent mutants and in the narrowing of the search space which needs to be considered in the generation of test data to kill a mutant.

It has been argued that the Jackson and Rollins style of dependence analysis (JR–Dependence) is more suited to mutation testing than the more widely used, Program Dependence Graph driven approaches. This is because the JR–style of analysis is more fine grained, relating not only nodes, but also the variables which are important at given nodes. This allows us to ask, for example, which variables are important at the entry node to the program and which variables need to be inspected at chosen probe points in order to avoid the equivalent mutant problem.

These dependence based techniques do not subsume constraint-based techniques, which may also detect equivalent mutants and which additionally generate test data. However, dependence analysis provides the tester with a complementary technology, to be used in tandem with constraint-based approaches to reduce the number of equivalent mutants which enter the constraint-based phase and to help to narrow the search space for data where constraint based approaches are unable to create data.

The paper culminates in a process for mutation testing which augments the existing constraint-based process with three additional dependence-based steps. One of these was previously proposed by the present authors, and is based upon amorphous slicing as a 'last resort' assistant to human analysis of stubborn mutants. The other two are new, and form a pre- and post- analysis phase, augmenting the existing mutation testing process. They avoid generation of mutants which JR–dependence can determine to be equivalent and help restrict attention to the effective set of input variables, where constraint based techniques are unable to generate killing test data.

Acknowledgements

We would like to thank Jeff Offutt and Martin Woodward for helpful discussions concerning mutation testing which helped to shape our ideas on the integration of dependence analysis techniques into the classic mutation testing process. The four anonymous referees also provided helpful comments which have improved the presentation of the paper.

This work is supported in part by EPSRC grants GR/M58719 and GR/M78083.

References

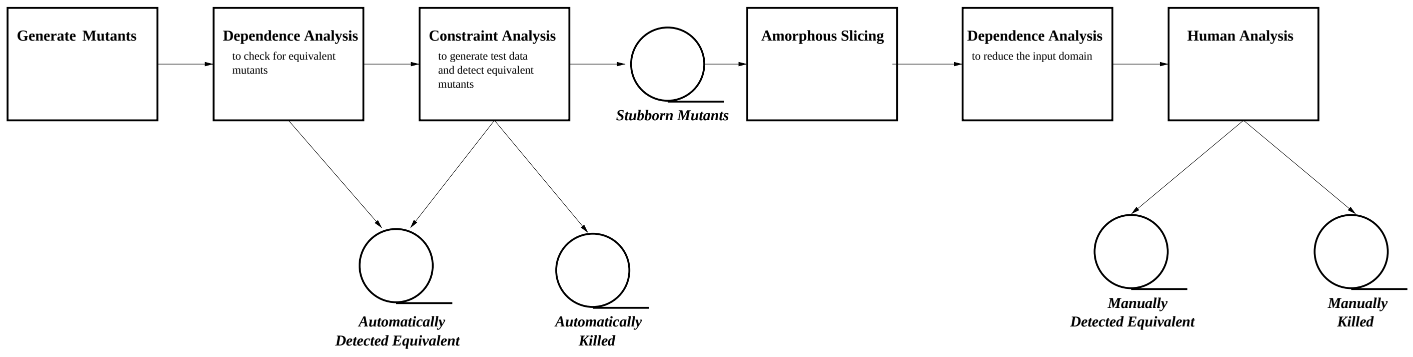
[1] AHO, A. V., SETHI, R., AND ULLMAN, J. D. *Compilers: Principles, techniques and tools*. Addison Wesley, 1986.

[2] BALDWIN, D., AND SAYWARD, F. Heuristics for determining equivalence of program mutations. *Research Report 276, Department of Computer Science, Yale University* (1979).

[3] BINKLEY, D. W. 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., 1999), ACM Press, New York, NY, USA, pp. 519–525.

[4] DANICIC, S. *Dataflow Minimal Slicing*. PhD thesis, University of North London, UK, School of Informatics, Apr. 1999.

[5] FERRANTE, J., OTTENSTEIN, K. J., AND WARREN, J. D. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems 9*, 3 (July 1987), 319–349.



Generate Mutants
Dependence Analysis
to check for equivalent mutants
Constraint Analysis
to generate test data and detect equivalent mutants
Stubborn Mutants
Amorphous Slicing
Dependence Analysis
to reduce the input domain
Human Analysis
Automatically Detected Equivalent
Automatically Killed
Manually Detected Equivalent
Manually Killed

Figure 4: A Mutation Testing Process which Combines Dependence and Constraint Analysis

[6] FRANKL, P. G., WEISS, S. N., AND HU, C. All-uses vs mutation testing: An experimental comparison of effectiveness. *Journal of Systems Software 38* (1997), 235–253.

[7] GALLAGHER, K. B., AND LYLE, J. R. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering 17*, 8 (Aug. 1991), 751–761.

[8] HARMAN, M., AND DANICIC, S. Amorphous program slicing. In 5^{th} *IEEE International Workshop on Program Comprenhesion (IWPC'97)* (Los Alamitos, California, USA, May 1997), IEEE Computer Society Press, pp. 70–79.

[9] HARMAN, M., FOX, C., HIERONS, R. M., BINKLEY, D. W., AND DANICIC, S. Program simplification as a means of approximating undecidable propositions. In 7^{th} *IEEE International Workshop on Program Comprenhesion (IWPC'99)* (Los Alamitos, California, USA, May 1999), IEEE Computer Society Press, pp. 208–217.

[10] HARMAN, M., AND GALLAGHER, K. B. Information and Software Technology, Special issue on program slicing, volume 40, numbers 11 and 12.

[11] HARMAN, M., SIVAGURUNATHAN, Y., AND DANICIC, S. Analysis of dynamic memory access using amorphous slicing. In *IEEE International Conference on Software Maintenance (ICSM'98)* (Los Alamitos, California, USA, Nov. 1998), IEEE Computer Society Press, pp. 336–345.

[12] HIERONS, R. M., HARMAN, M., AND DANICIC, S. Using program slicing to assist in the detection of equivalent mutants. *Software Testing, Verification and Reliability 9*, 4 (1999), 233–262.

[13] HORWITZ, S., AND REPS, T. The use of program dependence graphs in software engineering. In 14^{th} *International Conference on Software Engineering* (Melbourne, Australia, 1992), pp. 392–411.

[14] HORWITZ, S., REPS, T., AND BINKLEY, D. W. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems 12*, 1 (1990), 26–61.

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

[16] JACKSON, D., AND ROLLINS, E. J. Chopping: A generalisation of slicing. Tech. Rep. CMU-CS-94-169, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, July 1994.

[17] JONES, B., STHAMER, H.-H., AND EYRES, D. Automatic structural testing using genetic algorithms. *The Software Engineering Journal 11* (1996), 299–306.

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

[19] OFFUTT, A. J., AND CRAFT, W. M. Using compiler optimization techniques to detect equivalent mutants. *Software Testing, Verification and Reliability 4* (1994), 131–154.

[20] OFFUTT, A. J., JIN, Z., AND PAN, J. The dynamic domain reduction approach to test data generation. *Software Practice and Experience 29*, 2 (January 1999), 167–193.

[21] OFFUTT, A. J., AND PAN, J. Detecting equivalent mutants and the feasible path problem. In *Annual Conference on Computer Assurance (COMPASS 96), IEEE Computer Society Press* (Gaithersburg, MD, June 1996), pp. 224–236.

[22] OFFUTT, A. J., AND PAN, J. Automatically detecting equivalent mutants and infeasible paths. *Software Testing, Verification, and Reliability 7*, 3 (Sept. 1997), 165–192.

[23] OFFUTT, A. J., PAN, J., TEWARY, K., AND ZHANG, T. An experimental evalutation of data flow and mutation testing. *Software Practice and Experience 26* (1996), 165–176.

[24] PAN, J. Using constraints to detect equivalant mutants. Master's thesis, George Mason University, 1994.

[25] PARGAS, R. P., HARROLD, M. J., AND PECK, R. R. Test-data generation using genetic algorithms. *The Journal of Software Testing, Verification and Reliability 9* (1999), 263–282.

[26] TRACEY, N., CLARK, J., AND MANDER, K. Automated program flaw finding using simulated annealing. In *International Symposium on Software Testing and Analysis (ISSTA 98)* (March 1998), pp. 73–81.

[27] VOAS, J., AND MCGRAW, G. *Software Fault Injection*. Wiley, 1998.

[28] WAGNER, T. A., AND GRAHAM, S. L. Integrating incremental analysis with version management. In *Proceedings of the* 5^{th} *European Software Engineering Conference (ESEC'95)* (Sept. 1995), W. Schfer and P. Botella, Eds., Lecture Notes in Computer Science Nr. 989, Springer-Verlag, pp. 205–218.

[29] WAGNER, T. A., AND GRAHAM, S. L. Incremental analysis of real programming languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI-97)* (New York, June 15–18 1997), vol. 32, 5 of *ACM SIGPLAN Notices*, ACM Press, pp. 31–43.

[30] WEGENER, J., STHAMER, H., JONES, B. F., AND EYRES, D. E. Testing real-time systems using genetic algorithms. *Software Quality 6* (1997), 127–135.

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

[32] WOODWARD, M. R., AND HALEWOOD, K. 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, July 1988).

[33] YUR, J., RYDER, B. G., AND LANDI, W. A. An incremental flow-and context-sensitive pointer aliasing analysis. In *Proceedings of the 21st International Conference on Software Engineering* (Los Alamitos, California, USA, May 1999), IEEE Computer Society Press, pp. 442–452.

[34] YUR, J.-S., RYDER, B. G., LANDI, W. A., AND STOCKS, P. Incremental analysis of side effects for C software systems. In *Proceedings of the 19th International Conference on Software Engineering (ICSE '97)* (Los Alamitos, California, USA, May 1997), IEEE Computer Society Press, pp. 422–432.