Strong Higher Order Mutation-Based Test Data Generation

Mark Harman
CREST Centre
University College London
Gower Street, London, UK

Yue Jia
CREST Centre
University College London
Gower Street, London, UK

William B. Langdon
CREST Centre
University College London
Gower Street, London, UK

ABSTRACT

This paper introduces SHOM, a mutation-based test data generation approach that combines Dynamic Symbolic Execution and Search Based Software Testing. SHOM targets strong mutation adequacy and is capable of killing both first and higher order mutants. We report the results of an empirical study using 17 programs, including production industrial code from ABB and Daimler and open source code as well as previously studied subjects. SHOM achieved higher strong mutation adequacy than two recent mutation-based test data generation approaches, killing between 8% and 38% of those mutants left unkilled by the best performing previous approach.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Experimentation

Keywords

Mutation testing, Test data generation

1. INTRODUCTION

Mutation testing is a fault based testing technique, first proposed by DeMillo et al. [7] and Hamlet [13] and widely studied since [22]. The idea underpinning mutation testing is to seed faults to assess the adequacy of the testing approach. The fault-seeded version of the original program is called a 'mutant'. Research activity in mutation testing is increasing and its tools and techniques are reaching a state of maturity and widespread applicability [22].

If a test case distinguishes the behaviour of the original program from that of one of its mutants, then the test case is said to 'kill' the mutant. If the test case merely causes the state to change after the mutation point is executed then the mutant is said to be 'weakly' killed. However, if the test case causes this state change to propagate to an output, where an observable failure is observed, then the test case is said to 'strongly' kill the mutant.

ESEC/FSE'11, September 5–9, 2011, Szeged, Hungary.

Strong mutation testing embodies a more demanding test adequacy criterion than weak mutation testing, making it preferable that a test suite would be strong mutation adequate, where possible [41]. By definition a test that strongly kills a mutant must also weakly kill it, but not necessarily *vice versa*.

If a mutant and the original program are semantically identical then the mutant is said to be 'equivalent'; no test case can kill it. Equivalent mutants are a problem for mutation testing, because equivalence is undecidable, making it hard to know whether an unkilled mutant is killable.

Based on the types of faults seeded, mutation testing can classified as 'first order' or 'higher order'. First order mutation seeds only simple faults, generated by a single syntactic change to the original program. Higher order mutation combines simple first order faults to simulate more complex faults, motivated by a desire to capture subtle faults [21].

Higher Order Mutation Testing has been the subject of much recent attention [5, 16, 27]. As well as its ability to model more complex masking faults [20], there is evidence to suggest that it may reduce mutation effort [38] and also the proportion of mutants that are equivalent [25, 35].

There has been much work on different techniques and tools for generating mutants, with over 250 publications on mutation testing. However, the literature contains only 10 publications (about 4% of the total) that address the problem of automatically generating test data to kill mutants [22]. A summary of these 10 papers is presented in Table 1. While mutation generation remains important, it is also clearly desirable to be able to use mutation testing to generate test cases as well as to asses them.

Previous work on the generation of test data to kill mutants has used traditional structural-oriented test data generation techniques, for example, traditional symbolic execution [8, 26, 30, 31, 32], Dynamic Symbolic Execution (DSE) [34, 37, 42] and Search Based Software Testing (SBST) [4, 11]. However, all of the existing techniques are designed to achieve only weak mutation adequacy and only for first order mutants. There is neither existing work on killing higher order mutants, nor any work on generating strong mutation adequate test data.

This paper presents SHOM, a novel hybrid approach that draws on previous work from both DSE and SBST to achieve strong higher order mutation adequacy[1]. The paper presents evidence to support the claim that SHOM is efficient and effective for both first order and higher order mutation.

[1]A first order mutant is a special case of a higher order mutant so SHOM also achieves first order mutation adequacy.

The contributions of the paper can be summarised as follows:

1. We introduce a hybrid test data generation approach for strongly killing both first order and higher order mutants. We evaluate our approach on 17 subject programs, including 7 real world programs (four from two different closed source industrial systems and three for which source code is publicly available). For backward compatibility with comparable recent studies (that use C) and the older ones (that use Fortran) we also include C versions of 10 of the smaller programs studied in this previous work. However, our study also includes programs an order of magnitude larger than any of these smaller programs.

2. We report the results of an empirical evaluation of SHOM's efficiency and effectiveness for strong first order mutation adequacy. The results show that SHOM can kill up to 38% of the first order mutants left unkilled using reachability and infection, which in turn kills up to 36% of the mutants left unkilled using reachability alone.

3. We also report the results of a further empirical study of SHOM's efficiency and effectiveness for strong second order mutation adequacy. The results show that SHOM can kill up to 48% of the second order mutants left unkilled using reachability and infection, which in turn kills up to 41% of the mutants left unkilled using reachability alone.

The rest of this paper is organised as follows. Section 2 introduces our hybrid DSE/SBST approach, while Section 3 briefly describes implementation details. Section 4 explains the experimental method, the results of which are discussed in Section 5. Section 6 introduces related work, and the paper concludes with Section 7.

2. STRONGLY KILLING HIGHER ORDER MUTANTS USING DSE AND SBST

We first define a mutant and a higher order mutant and what it means to kill them, before explaining our approach to generating test data using a combination of DSE and SBST to strongly kill higher order mutants.

DEFINITION 1 (FIRST ORDER MUTANT). *A first order mutant p' of a program p is constructed by making a single syntactic change to p. A transformation that produces a mutant from the original program is called a 'mutation operator'.*

Of course there remains the question of what is a 'single syntactic change'. There are many definitions of such sets of mutation operators in the literature [1, 33]. For our purposes, it is only important to define first order mutation so that we can define higher order mutation in terms of it. Higher order mutation can only be formally defined with respect to a set of first order mutation operators.

DEFINITION 2 (HIGHER ORDER MUTANT). *Given a set of first order mutation operators M, if a mutant p' is created from a program p by the application of k operators from M then p' is said to be a k^{th} order mutant of p.*

Definition 2 of Higher Order Mutation subsumes Definition 1 of first order mutation because setting $k = 1$ in Definition 2 yields Definition 1. In general, care will be required to ensure that all of the k mutation operations creates a distinct syntactic change when applied to p. It may also be necessary to define the order of application of the k mutation operators, since different application orders may produce different overall syntactic effect. However, we leave these topics for future studies on higher order mutation.

Higher order mutants are generally easier to kill than first order mutants. However, there also exists a small set of higher order mutants that is harder to kill than the first order mutants from which they are constructed. This type of higher order mutant is known as a subsuming higher order mutant [21]. Figure 1 shows a simple illustrative example of subsuming higher order mutant. Both mutant 1 and mutant 2 are so-called 'dumb' mutants (those which are very easy to kill). In this case both are killed by *any and every* test cases; the dumbest possible. However, the higher order mutants created by inserting both mutant 1 and mutant 2 together is far from dumb. It is much harder to kill than either of its first order mutants. Essentially, in this sort of situation, fault masking can create subtle higher order bugs from unsubtle first order bugs.

In order to (strongly) kill a first order mutant the killing conditions are well studied in the literature: A test input needs to satisfy following three conditions: Reachability, Infection and Propagation (RIP), each of which subsumes the preceding condition(s):

1. **Reachability**: The location of the mutant in the program must be executed by the test case. We say the mutant is 'reached'. Reaching all mutants of a program can be achieved by any branch adequate test set, so reachability is an instance of branch coverage, which is widely studied in literature [2, 12, 19, 39].

2. **Infection**: Immediately after mutant execution, the original program state and that of the mutant must differ. We say, the mutant 'infects' the state. A test case that achieves infection for a mutant m is also said to 'weakly kill' the m [8, 22, 28].

3. **Propagation**: The infected state must propagate to some point in the program at which it can be observed, such as an output statement. A test case that achieves propagation for a mutant m is also said to 'strongly kill' the m [8, 22, 28].

2.1 Weakly Killing Mutants

DSE has proved to be an effective means of satisfying both the reachability and infection conditions [12, 39] and, as a result, there has been work on DSE as a technique for achieving weak mutation adequacy [34, 37, 42]. However, it has not been adapted to handle strong mutation.

Our approach uses DSE to generate weakly killing constraints and test data that satisfy them. When generating mutants, properties denoting reachability infection are collected for each mutant. The reachability property is captured by the set of critical predicate nodes that transitively control mutant reachability. This property is generated by traditional control dependence analysis. The second property is the infection constraint which is determined by the specific type of mutant. We use the infection conditions found in the work of DeMillo and Offutt [8].

Table 1: Mutation-based Test Data Generation. In referring to study example sizes, 'tiny' refers to laboratory programs consisting of a single procedure, while 'small' refers to slightly less trivial laboratory programs. The term 'non-trivial' is reserved for real world programs, neither written by students, nor the experimenters themselves, nor drawn from the 'Siemens Suite'. (†)The work of Offutt et al. [31, 32] reported basic block and dataflow coverage, but not Mutation Score. (‡)The work of Fraser and Zeller achieved (R)eachability and (I)nfection and also a constrained form of (P)ropagation, because it sought to maximize the mutant's effect on assertions, providing a form of propagation and also a way to maximise mutant impact.

Authors [Ref]	Year	(R)eaches, (I)nfects, (P)ropagates	Technique	Tool Available?	Subjects Studied	Subject Language	Largest Subject	Average mutation score	(F)irst order / (H)igher order
DeMillo and Offutt [30, 8]	1991	R,I (Weak)	Static Domain Reduction	Yes	5 tiny examples	Fortran	55 Lines	98%	F
Offutt et al. [31, 32]	1994	R,I (Weak)	Dynamic Domain Reduction	Yes	12 tiny examples	Fortran	100 Lines	Not† given	F
Liu et al. [26]	2006	R,I (Weak)	Dynamic Domain Reduction	No	5 tiny examples	C	21 Lines	95%	F
Zhang et al. [42]	2010	R,I (Weak)	DSE	Yes	5 small examples	C#	472 Lines	90%	F
Papadakis et al. [36]	2010	R,I (Weak)	DSE	No	5 tiny examples plus 3 small Siemens suite examples	C	500 Lines	63%	F
SHOM [this paper]	**2011**	**R,I,P (Strong)**	**DSE&SBST**	**Yes**	**8 non-trivial examples in Table 2 and 10 of the above listed in Table 3 for comparability**	**C**	**9,564 Lines**	**First: 69% Second: 71%**	**F&H**
Ayari et al. [4]	2007	R (Weak)	SBST	No	2 tiny examples	Java	72 Lines	88%	F
Papadakis et al. [37]	2010	R (Weak)	DSE	Yes	10 tiny examples	Java	100 Loc	90%	F
Fraser and Zeller [11]	2010	R,I (Firm)‡	SBST	Yes	2 non-trivial examples: Commons-Math & Joda- Time	Java	412 Classes	72%	F

2.2 Handling Higher Order Mutants

We adopt and adapt the previous work on DSE for first order mutation testing, so that it is able to handle higher order mutants in addition to first order mutants. A higher order mutant, m, of order n is a is a composition of n first order mutants. We shall call these n first order mutants the 'constituent' mutants of m. For each higher order mutant, there are two important cases to consider: Case 1: There exists a path that traverses all constituent first order mutants. Case 2: There does not exist such a path.

If Case 1 applies, then it is possible that the higher order mutant is a subsuming higher order mutant. A 'subsuming' higher order mutant is one that is harder to kill than any of its first order constituents, due to fault masking among the constituent first order mutants [21]. In testing terms, we may say that 'the sum of the collection of first order mutants is more demanding to test than the union of its parts'. However, if there does not exist a path that passes through all constituent first order mutants then, by definition, they cannot all mask one another and so the 'sum is merely the union of its parts' and is, therefore, easier to kill.

Of course, in Case 2 there could be a path that traverses some subset, S, of the constituent first order mutants, but this would mean that there would also be a lower order mutant composed of precisely the S constituent mutants. If we seek to progressively increase the order of mutants considered, then such a case will already have been encountered. Therefore, we focus our attention on Cases 1 and 2 as defined above.

Suppose a higher order mutant that we seek to weakly kill is constructed from a set of constituent first order mutants $f_1, \ldots, f_n$. If there is a path in the control flow graph of the program that passes through all the critical predicate nodes of $f_1, \ldots, f_n$ then the higher order mutant may be subsuming. This is Case 1. For these higher order mutants, we define the critical predicate nodes of the higher order mutant to be the union of the critical predicate nodes of the $f_1, \ldots, f_n$. By extension, the infection constraint of the higher order mutant is the conjunction of the infection constraints of $f_1, \ldots, f_n$.

If there is no such path (Case 2), then it is not possible to find a test case that executes all the constituent first order mutants that combine to make the higher order mutant. In this situation, our approach treats the higher order mutant as merely a set of first order mutants; it is killed if any of the constituent first order mutants is killed.

We use a different variant of the DSE algorithm to that previously used for mutation testing [34, 37, 42]. Our reachability approach is the same as previous work and this is inherited from the standard DSE approach to branch coverage [12, 39]. However, we handle infection constraints differently, because we need to retain and extend the constraints for subsequent generation of strongly killing test cases.

Figure 1: Illustrative example: two dumb first order mutants combine to make a subtler second order mutant

```
intputs: a, x, y
      1  z = x;
      2  z = z + y;
      3  if(a > 0)
      4     return z;
      5  else
      6     return 2 * x + z;
```

mutant 1: changes line 1 to z = ++x
mutant 2: changes line 2 to z = z + - -y
mutant 12 (2nd order mutant): combines mutant1 and mutant2 together

inputs	output schema			
tests	original	mutant 1	mutant 2	mutant 12
$a > 0$	$x + y$	$x + y + 1$	$x + y - 1$	$x + y$
$a \leq 0$	$3x + y$	$3x + y + 3$	$3x + y - 1$	$3x + y + 2$
	n/a	killed by all	killed by all	killed by half

Previous work uses a testability transformation to transform the traditional branch adequacy problem, which is handled well by DSE, into weak mutation adequacy. This is done by simply replacing mutants with additional branches, the predicates of which capture the infection constraint.

Our approach does not transform the program. Rather, once a mutation point is reached, our DSE variant continues to generate test data to satisfy the weak killing constraint. This allows us to retain a mapping of mutants and the corresponding infection constraints, so that we can assess the fitness of each individual mutant when it subsequently comes to the task of propagating infections. The pseudo code of this DSE algorithm is shown in Algorithm 1.

If the DSE approach fails to generate weakly adequate test data for a mutant, we use standard SBST approaches to seek to weakly kill it. This is because it is known [23] that DSE and SBST achieve coverage of distinct, but overlapping, sets of branches. For example, SBST is well adapted to test data generation in the presence of floating point computation. This motivated work on a hybrid DSE–SBST approach, now incorporated into the Pex tool [24].

However, for our experiments (reported in Section 4), we switch off this search based weak killing feature of the SHOM implementation, so that weak adequacy is achieved by DSE alone. This was because we wish to compare the additional effort required and effectiveness achieved in terms of strong adequacy compared to the DSE–only approaches to weak adequacy.

Having used DSE to generate weakly adequate test data our hybrid DSE–SBST approach uses SBST to search for test inputs that propagate infected data states to outputs, thereby transforming weak mutation into strong mutation. The next section explains our SBST approach to strong higher order mutation testing, which lies at the heart of our overall SHOM approach.

2.3 Strongly Killing Mutants

In order to strongly kill a mutant, its infection must be propagated to an output so that the fault is manifested as a failure. The propagation problem has previously been considered to be hard because there may be infinitely many paths from the infection point to the point at which an output occurs. Therefore, the problem of propagation, *for each mutant*, can be reduced to the path coverage problem. Even if we approximate path coverage, this process would still have to be repeated for each mutant and so the cost would potentially be prohibitive.

Our approach uses SBST to search for paths from the infection point to the output that are *more likely* to propagate the infection, based on heuristic assumptions about the differences in paths taken by the original and the mutant, which we seek to maximise using the search.

Algorithm 1 The dynamic symbolic execution algorithm

Require: the set of critical predicate nodes N reaching the mutant
Require: the $InfectionConstraint$ of the mutant
  For program P, randomly generate concrete test input T
  while within execution upper bound **do**
    execution path $p \leftarrow$ dynamic execution (P, T)
    symbolic expression $sc \leftarrow$ symbolic execution (P, T)
    if p does not reach the mutant **then**
      current critical node $n \leftarrow$ get next critical node (N, p)
      $p \leftarrow$update constraints (p, n)
      $T \leftarrow$ constraint solver(p, sc)
    else
      break
    end if
  end while
  weak killing constraint $wkc \leftarrow InfectionContraint \wedge p$
  $T \leftarrow$ constraint solver(wkc)
  return T

In this way, we do not explicitly try all paths from infection to output. Rather, we *search* for those more likely to propagate, guided by a fitness function that measures control flow differences between original program and mutant.

We first use a testability transformation to ensure that the program has only a single return point; the return of the procedure in which the mutant resides. This simple transformation is always possible, because multiple return statements can be directed to a single 'gathered' return point.

We seek to maximally disrupt the path taken by the mutant version of the program from the infection point to this unique return statement. This increases the likelihood that any output statement that can be executed after the infection point will be executed differently (or even not at all). This, in turn, increases the likelihood that the mutant's output will be distinguishable from that of the original, thereby strongly killing the mutant.

We wish to favour tests that maximise disagreement on predicate choices made by the original program and mutant, thereby maximally disrupting the control flow path from the infection to the return. If a test makes the mutant follow a different path to the original after execution then it is very likely to produce a different value at the return point, thereby strongly killing the mutant. Let $Branch(p, i, t)$ denote the branch taken by program p at predicate i on input t. Let $inf(m)$ denote the infection point of mutant m and let $ret(m)$ denote the return point of the procedure containing m. Let $pred(p, x, y)$ denote all critical predicates between point x and point y in program p.

We define the decision function d for program p and mutant m at predicate i on input t as follows:

$$d(p,m,i,t) = \begin{cases} 1 & \text{if } Branch(p,i,t) = Branch(m,i,t) \\ 0 & \text{if } Branch(p,i,t) \neq Branch(m,i,t) \end{cases}$$

Our fitness $f(p,m,t)$ of a test case t executed on a mutant m of an original program p is defined to maximise the average 'predicate disagreement' between m and p:

$$f(p,m,t) = \frac{\sum_{i \in Pred(m,inf(m),ret(m))} d(p,m,i,t)}{n}$$

Recent results [18] have demonstrated that random restart hill climbing provides an effective and efficient way to generate test data using SBST. Motivated by this finding, we use a random re-start hill climbing algorithm to search for the test inputs that propagate the infection, as shown in Algorithm 2. However, the particular choice of SBST algorithm is a parameter to our approach and a pluggable component to its implementation.

2.4 Preserving Weak Adequacy Using Constrained Search

Our representation and move operations are designed to *guarantee* that the previously obtained reachability and infection constraints are also satisfied by any candidate input we consider during the SBST phase of our overall approach. To do this we represent an individual candidate solution to the problem of killing a mutant as a conjunction of constraints. This conjunction starts off as the reachability and infection constraints, to which we may subsequently only ever add additional conjuncts during the search process.

In order to express a potential move to a new test input in the search, we add an extra conjunct to the current constraint, representing the result in Conjunctive Normal Form (CNF). In this way we can only ever consider weakly killing test cases. The constraint solver is used to generate a candidate using the extended CNF consisting of the weakly killing constraint plus some candidate new constraint. The test input generated by the constraint solver is then evaluated for fitness and, if it improves fitness, it becomes the new current solution in the hill climb.

Our 'constrained search' approach to searching for test data is a novel aspect of our mutant killing technique that has not been used in any previous work on SBST. It may find other applications in more general work on SBST outside the domain of mutation testing. It allows us to combine constraint solving and SBST in a manner that preserves the value captured by the constraints, while extending it to achieve some additional aspiration using search.

3. SHOM IMPLEMENTATION

Figure 2 depicts the architecture of SHOM, the implementation of our hybrid DSE–SBST approach to Strong Higher Order Mutation. To compute adequacy scores we use the tool MiLu[20, 21]. MiLu is a higher order mutant generation and assessment tool that supports general purpose first and higher order mutant generation for C. We used the subset of the Agrawal et al.'s 77 C mutation operators [1] that fall into the widely studied 'selective' mutation operators, defined and studied first by Offutt el al. [33]. We use our own implementation of the DSE phase so that we could extend it to include the subsequent SBST phase.

Algorithm 2 Out hill climbing algorithm

Require: A weak killing test T
Require: The weak killing constraint wkc
  if T kills the mutant strongly **then**
    return T
  else
    while current evaluation < max evaluation **do**
      $NeighboursTests \leftarrow$ neighbours(T)
      for all t in $NeighboursTests$ **do**
        if t kills the mutant strongly **then**
          return t
        end if
        for all t in $NeighboursTests$ **do**
          if fitness(t) > $bestfitness$ **then**
            $BestTest \leftarrow t$
            $bestfitness \leftarrow$fitness($BestTest$)
          end if
        end for
        if $bestfitness \leq$ fitness(T) **then**
          $T \leftarrow$ get a weak killing test T
        else
          $T \leftarrow BestTest$
        end if
      end for
    end while
  end if

We use the CIL transformation system [29] to pre-process the program and its mutants for the DSE and SBST phases of our implementation. However, this is merely a testability transformation that reduces constraint and path analysis effort. It does not alter the semantics of the program under test, nor does it affect the test adequacy criteria involved. As illustrated in Figure 2, the test data generated using our approach is evaluated on the mutants generated by MiLu, not the transformed versions.

Three transformation steps are performed. We first simplify the expressions denoting array indeces and other memory access operators. In this step, additional temporary variables are introduced to hold intermediate values for complex memory expressions which involve more than one memory reference. After this step, the lvalue of the simplified expression only contains a memory constructor. This simplifies our subsequent static analysis and dynamic symbolic , execution, by reducing the number of cases that have to be considered.

We also use CIL's standard transformations to simplify loop and switch statements, reducing all such control flow constructs to a simple canonical form, consisting of conditionals and branches. Once again, this leaves the semantics of the original unaltered, but eases the subsequent downstream analyses that we perform.

Finally, we transform each procedure to an equivalent single–entry/single–exit version, so that it contains exactly one single return statement, to which we seek to propagate the infection of all mutants that lie inside that procedure. As explained in the previous section, this simplifies strong mutation testing, since it means that our SBST phase need only consider a single exit node. For this single exist node, SBST seeks inputs that cause execution to flow from the infection point along a maximally disrupted control flow path to the exit node.

We also use CIL to perform our control dependence analysis. This collects the critical predicate nodes for each mutant, used to form the reachability and infection conditions. The dependence analysis is also used to identify those predicates for which SBST seeks to cause the mutant and original to disagree from infection to return.

The constraints for reachability and infection are represented in Conjunctive Normal Form. SHOM uses the Yices constraint solver [10] to solve these constraints. Yices is a Satisfiability Modulo Theories (SMT) constraint solver that uses a collection of advanced constraint solving techniques to find a satisfying assignment of values to variables in formulæ. We use it to satisfy the constraints for reachability and infection. Yices was chosen for two reasons:

1. Yices provides a C language application interface. This is necessary since we cannot simply use constraint solving as a 'black box' component. While this is possible for weak mutation killing techniques that simply use testability transformation to reformulate weak mutation as branch coverage, it is not possible for strong mutation. For strong mutation we require control over exactly which constraints need to be satisfied at each part of the overall SHOM process.

2. Yices provides state-of-the-art constraint solving. It supports a wide range of constraints, including linear expressions, scalar types, recursive datatypes, tuples, records, arrays and bit-vectors, all of which can arise in the constraints found in programming languages. It won first place for several of the categories of the 2005, 2006 and 2007 SMT-COMP competitions organised as part of the Computer Aided Verification Conference (CAV).

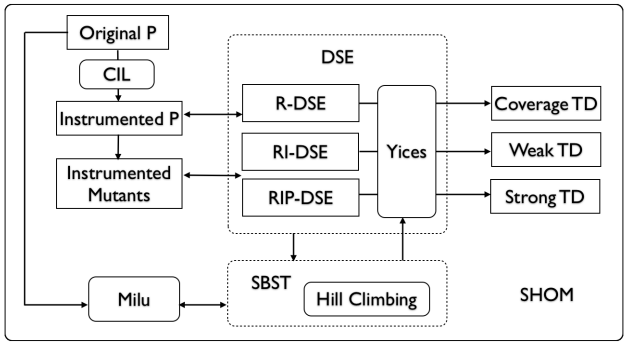

Figure 2: The SHOM Architecture. The DSE and SBST components were built from scratch. However, the DSE component delegates constraint solving to Yices [10]. It performs its analysis on transformed mutants, but all test data generated by SHOM is executed and evaluated by MiLu[20]. Transformation is performed CIL [29].

4. EMPIRICAL STUDY

In our studies we consider, separately, first order mutation and higher order mutation, because first order mutation has been the subject of previous work, while no other previous study has considered test data generation to kill higher order mutants. We consider only second order mutants and, for larger programs, only sets of randomized samples from the set of all possible second order mutants. Sampling is required because of the infeasibility of considering all higher order mutants due to the explosion in mutant numbers that occurs at higher orders.

4.1 Subject Programs Studied

We applied SHOM to the example subjects in Tables 2 and 3. The examples in Table 2 are non-trivial real world programs. Four are modules from closed source industrial production code. Two of these, DeFroster and F1, come from Daimler and are used in automative control systems for a rear window defrosting system and an engine controller respectively and have been used in previous studies [14]. The other two, Hash and Buff, come from ABB and are used in robot controller systems.

We cannot provide the source code for these examples, because they are proprietary closed source code from industrial partners with which we have Non Disclosure Agreements in place. However, to support replication and more robust evaluation, we also include three additional larger programs, for which source code is readily available.

The program Space is a widely studied interpreter for an array definition language used by the European Space Agency. It is not open source, but its code is available from the Software-artifact Infrastructure Repository (SIR) [9]. The other two programs, Gzip (v1.5) and GArray (v2.26) are both open source. Gzip is the widely used compression program. GArray is an array data structure used in the GNU Glib. All programs in this non-trivial subject set of examples are summarised in Table 2.

The second set, summarised in Table 3, contains smaller laboratory programs that have been widely studied in the literature on mutation-based test data generation. We include this set of relatively small programs to provide backward compatibility with these previously studied examples.

The set includes three programs taken from SIR that originated in the Siemens suite: Tcas is an aircraft anti-collision system. Schedule is a program that prioritises schedulers. Replace performs pattern matching and substitution.

The remainder are a sample of some of the tiny programs used in previous studies. We make no attempt to infer findings from the results obtained using these tiny examples, but include them to facilitate replication. Triangle classifies the type of a triangle by the lengths of its three edges. Bubble is the standard bubble sort algorithm. Days calculates the number of days between two given days. Find locates and sorts the input array with a given index. Mid returns the middle value of three inputs. GCD is Euclid's GCD algorithm and MinMax returns the minimum and maximum values of the input array.

4.2 Mutant Generation

Some of the programs studied perform no output. For example, many of the tiny programs simply compute a single value as their result. For such programs we need to clarify what we mean by 'output'. If we took an overly pedantic and literal definition of output, for example: 'something that appears on an output device', then all mutants of such programs would be equivalent, because no mutation can change a non-existent output. Therefore, we allow 'output' include the result of the computation returned to the environment (such as a return value or the result computed in a global variable).

For the larger programs where the code is not a support routine, but an entire program, there is no such issue. These larger programs perform output, to screen and/or files and this is monitored and compared with the output of the original to determine whether the mutant is strongly killed.

4.3 Research Questions

Our study asks three research questions, which we define here, explaining how our experiments are designed to address them.

RQ1: How first-order-adequate is SHOM? To explore SHOM's test effectiveness for strong first order mutation, we compare SHOM with RI-DSE. We also report on the improvement RI-DSE achieves over R-DSE. In both cases we generate test sets for R-DSE and RI-DSE and compute the number of mutants each kills strongly and compare this to the number of mutants strongly killed by SHOM. This allows us to evaluate the degree to which a reached mutant is infected and propagates merely by reaching it using DSE and also the degree to which those mutants infected using DSE also already happen to propagate. We repeat all experiments ten times and take the average to cater for the stochastic nature of the search algorithm.

RQ2: How second-order-adequate is SHOM? The number of higher order mutants grows exponentially with the order k, presenting obvious experimental design challenges. For all of the ten programs from Table 3, the total number of second order mutants is 392,458 (which is manageable). However, for the real world programs in Table 2, the total number of second order mutants is 63,799,406 (which is unmanageable).

The quadratic increase in the number of second order mutants makes it impractical to consider all second order mutants. We therefore adopt a sampling approach. For programs with 0–4,999 second order mutants we use 100% of the mutants. For programs with 5,000–49,999 mutants, we sample 10%. For programs with 50,000–499,999 mutants, we sample 1%. For programs with 500,000–4,999,999 mutants, we sample 0.1%. For programs with 5,000,000 or more mutants, we sample 0.01%. To avoid sampling bias, we sample randomly from the set of all second order mutants. We also repeat the sampling experiment 10 times and compute the average levels of strong second order mutation adequacy achieved over all 10 samples.

To answer RQ2 we compare SHOM with RI-DSE and compare RI-DSE with R-DSE. However, there is no previous work on generating test data to kill second order mutants (either weakly or strongly). Therefore, to provide a baseline for comparison, we use the union of all test data generated for each of the two first order mutants from which the second order mutant is constructed as follows:

Suppose s is a second order mutant with constituent first order mutants f_1 and f_2. We use R-DSE to generate test data to kill f_1, creating a set of test data d_1. We then use R-DSE to generate test data to kill f_2, creating a second set of test data d_2. We define the result of applying R-DSE to s to be $d_1 \cup d_2$. Similarly, for RI-DSE, we generate two test sets, one for each of f_1 and f_2 and define the test set produced by RI-DSE for s to be the union of the two.

Using this approach, R-DSE and RI-DSE should be capable, in theory, of killing all those second order mutants that are coupled to their first order constituents in a way that killing either first order mutant kills the second order mutant. However, for second order mutants where fault masking may take place, a test set that kills both constituent first order mutants it not guaranteed to kill the second order mutant.

Table 2: The 7 larger programs used in the experiments. The upper 5 are industrial programs, while the lowest 2 are open source.

Program Name	Lines Of Code	Functions	Branches	1st order mutants	2nd order mutants
`DeFroster`	237	2	76	215	22,732
`F2`	511	1	42	212	22,113
`Hash`	1,011	12	76	465	107,211
`Space`	9,564	136	1,190	4,410	9,715,606
`Buff`	1,371	14	182	1,544	1,189,040
`GArray`	808	58	17	1,363	926,286
`Gzip`	7,933	97	1,717	10,182	51,816,418

Table 3: 10 smaller programs included for backward compatibility with previous studies.

Program Name	Lines Of Code	Functions	Branches	1st order mutants	2nd order mutants
Triangle	88	1	32	253	31,522
Bubble	35	1	6	80	3,032
Days	86	1	28	242	28,849
Find	88	1	22	201	19,791
Mid	43	1	10	65	1,970
GCD	43	1	6	73	2,526
MinMax	44	1	6	39	657
Tcas	166	8	66	223	24,496
Replace	595	23	176	714	253,585
Schedule	425	18	66	230	26,000

RQ3: How efficient is the SHOM data generation approach? The efficiency was measured using both the elapsed time for test data generation and the number of fitness evaluations required. Again, to cope with the stochastic nature of the search process, each experiment was repeated ten times and averages are reported.

The time was recorded using the Linux `time` utility. This is the elapsed time, so it includes all time taken to generate mutants, test data and to run test data on the program under test. As such, the timing information denotes a worst case upper bound on the total amount of time a tester would be expected to wait for test data to be produced by each technique. The experiments we undertaken on a Macbook pro laptop with Intel Duo2 2.6 GHz CPU, 4GB Memory in the Ubuntu 10.10 operating system.

5. RESULTS AND ANALYSIS

In this section we present the answer to each research question in turn, indicating how the results answer each. We consider strong first order effectiveness, followed by strong second order effectiveness and finally report on the efficiency of the SHOM implementation.

5.1 SHOM's First Order Adequacy

The results relating to RQ1 are summarised in Table 4.
Answer to RQ1: As can be seen, SHOM produces increases in strong first order mutation adequacy compared to RI-DSE, which in turn produces noticable improvements on the strong adequacy achieved by R-DSE. For the smaller programs, the improvement in strong adequacy achieved by both RI-DSE and SHOM is less notable than it is for the larger programs.

Table 4: The complete results for all experiments. Columns labelled 'R-DSE' and 'RI-DSE' report, as percentages, the strong adequacy achieved by R-DSE and RI-DSE respectively for 1st and 2nd order mutants. Of the four columns labelled RI-DSE, the second two columns report the percentage of mutants left unkilled by R-DSE which are killed by RI-DSE. The four columns labelled 'SHOM' report the strong adequacy achieved by our new approach SHOM. The figures in the first two columns for SHOM report the average percentage of first and second order mutants killed over ten runs. The parenthetic numbers report the standard deviation. The figures in the second two columns for SHOM report the average percentage improvement of SHOM over RI-DSE for first and second order mutation. In these two columns the parenthetic numbers report the number of runs, out of ten, for which SHOM outperformed RI-DSE. The two columns labelled 'Time' report the average time taken by SHOM (in minutes). The final two columns, labelled 'Fitness' report the average number of kilo fitness evaluations required.

Program	**R**-DSE %		**RI**-DSE %				SHOM %				Time		Fitness	
	Order		Order		Imp. on **R**		Order (Std.)		Imp. on **RI** (# wins)		Order		Order	
	1st	2nd	1st	2nd	1st	2nd	1st	2nd	1st	2nd	1st	2nd	1st	2nd
Triangle	48	49	59	61	21	24	62 (1.6)	67 (3.8)	7 (10)	15 (10)	13	102	3	17
Bubble	76	77	76	77	0	0	76 (0.0)	77 (0.0)	0 (0)	0 (0)	22	141	0.2	8
Days	62	66	64	68	5	6	65 (0.5)	72 (2.6)	3 (10)	13 (10)	14	114	5	13
Find	64	59	69	60	14	2	69 (0.0)	61 (0.2)	0 (0)	3 (10)	28	191	2	9
Mid	65	62	66	73	3	29	82 (4.2)	82 (2.3)	47(10)	33 (10)	6	48	0.3	4
GCD	71	73	73	82	7	33	73 (0.0)	82 (0.0)	0 (0)	0 (0)	12	88	0.2	5
MinMax	75	64	77	75	8	31	77 (0.0)	76 (0.2)	0 (0)	4 (10)	22	84	0.1	3
Tcas	42	55	54	67	21	27	62 (2.1)	69 (9.1)	17 (10)	6 (10)	48	172	3	12
Replace	46	42	53	56	13	24	72 (2.2)	77 (11.5)	40 (10)	48 (10)	200	272	8	18
Schedule	55	57	57	62	4	12	69 (5.4)	70 (7.2)	28 (10)	21 (10)	110	202	4	15
Hash	51	54	56	61	10	15	63 (2.9)	64 (3.5)	16 (10)	8 (10)	81	128	5	8
Buff	63	64	71	73	22	25	82 (6.1)	85 (6.5)	38 (10)	44 (10)	152	176	11	7
GArray	64	68	77	81	36	41	82 (3.7)	86 (5.7)	22 (10)	26 (10)	95	131	2	3
DeFroster	53	55	62	63	19	18	66 (2.1)	68 (4.0)	11 (10)	14 (10)	102	272	2	11
F2	44	44	63	60	34	29	66 (1.2)	67 (8.4)	8 (10)	18 (10)	122	321	2	14
Space	30	32	46	51	23	28	52 (2.3)	57 (12.2)	11 (10)	12 (10)	1,423	884	43	18
Gzip	34	33	42	44	12	16	50 (1.5)	52 (13.4)	14 (10)	14 (10)	2,762	1,794	92	64
Average	55	56	62	65	15	21	69 (2.1)	71 (5.3)	15 (7.6)	16 (8.8)	307	301	11	13

This difference in behaviour is a further justification for including larger programs in the study of mutation-based test data generation. As we have seen, using only tiny examples may skew the results due to the relatively trivial nature of the test data generation problem for these tiny programs.

R-DSE and RI-DSE are entirely deterministic. SHOM builds on RI-DSE, but it is a randomised algorithm, so it can produce different values each time it is run. However, it is guaranteed to perform no worse than RI-DSE by construction, so we report the improvement it achieves (averaged over ten runs) together with standard deviation.

These are the first results reported for strong mutation test data generation so it is not possible to directly compare our results with previous findings, such as those in Table 1. Perhaps the closest work to ours is that of Frazer and Zeller [11].

Though Fraser and Zeller report on test data generation for Java, while we report on test data generation for C, their work is evaluated on two larger, non-trivial subjects and it achieves a form of propagation (to assertions in the program rather than outputs). Fraser and Zeller reported an overall average first order mutation score of 72% which lies between our weighted average strong first order mutation score for the whole programs (which was 59%) and that we achieved for the libraries (which is 76%).

There is a noticeable difference in the performance of all techniques for smaller and larger programs. For the smaller programs from Table 3, R-DSE is able to strongly kill between 42% and 76% of the first order mutants. RI-DSE can improve on this, but for some of the programs the test problem is so trivial that even weakly adequate test sets achieve high levels of strong mutation adequacy.

For the larger programs the results are more interesting. The behaviour of all three techniques falls into two distinct categories, depending on whether the larger program is a whole program or merely a collection of library routines to be called by some other program. Of the larger programs, Hash, Buff and GArray are each collections of routines to be called from elsewhere; these three programs consist of libraries of subordinate routines; they have no `main` function. The other four of the larger programs: DeFroster, F2, Space and Gzip, are invoked, in their entirety, from their `main` function so that the whole program is tested.

It has been known for some time [3] that whole program analyses are more challenging than inter-procedural analyses that focus on a single procedure. This is also true for test data generation. For the libraries, we need merely test each procedure in turn, thereby focusing the testing on a single procedure body rather than a whole program. The single procedure may call others in the library, so testing is still an inter-procedural activity, but it is not a 'whole program activity'.

This dichotomy between whole programs and libraries is borne out in the results. For the libraries, R-DSE is able to strongly kill between 51% and 64% of the first order mutants, whereas for the whole programs it kills between 30% to 53% of the mutants. RI-DSE improves on this, killing between 10% and 36% of the remaining mutants for the libraries and between 12% and 34% of the remaining mutants for the whole programs.

SHOM further improves strong first order mutation scores in all of the larger programs studied. For the library programs it manages to kill between 16% to 38% of the remaining mutants left unkilled by RI-DSE. For the whole programs, SHOM kills between 8% and 14% of the remaining mutants unkilled by RI-DSE.

5.2 SHOM's Second Order Adequacy

As can be seen from Table 4, on average, over all programs studied, all three techniques (R-DSE, RI-DSE and SHOM) are better at killing second order mutants than first order mutants. This is to be expected since second order mutants are, in general, coupled to first order mutants [22, 25]. These are the first results reported in the literature for automated test data generation to kill second order mutants, so they provide a baseline for future work.

The results also provide a baseline against which to evaluate SHOM. Over all programs studied, it produces an improvement in strong second order adequacy over RI-DSE, which, in turn, produces an improvement over R-DSE. Once again, we report average performance for SHOM (over ten runs) and standard deviation. Note that statistical tests such as the t-test or Mann Whitney test are not suitable here: The empirical evaluation is required to determine the size of this improvement, but SHOM is *guaranteed* to perform no worse than RI-DSE by construction.

For the larger programs from Table 2, the dichotomy between libraries and whole programs is evident for second order mutation (as it is for first order mutation). For whole programs, the adequacy of all techniques is reduced compared to that for libraries. Over all larger programs, RI-DSE kills between 15% and 41% of the second order mutants left unkilled by R-DSE, while SHOM further increases this effectiveness, killing between 8% and 44% of the mutants left unkilled by RI-DSE

5.3 SHOM's Efficiency

Table 4 reports the number of fitness evaluations and time required to kill all mutants. The number of fitness evaluations required is not dissimilar to that required for branch coverage of similar sized programs using search based techniques [2], so performance can be expected to be in line with previous work on SBST.

For the practicing software tester, the number of fitness evaluations, though machine-independent, will be of less interest; the results for the time taken to find an adequate test set are more important. The largest of the programs previously studied for mutation-based test data generation with C are the Siemen's suite examples (Schedule, Replace and Tcas from Table 3). For these programs we are able to generate a weakly killing test set in seconds.

It is not possible for us to compare these findings with the previously reported results from the literature on mutant test data generation for C. This is because the relevant papers for which a comparison would be meaningful reported in detail upon only the effectiveness (mutation score) of the approaches on which they reported and did not report the execution time details required for a comparison.

Of course, after two decades of Moore's Law, even were timing data available then, for those older studies from the 1990s, a head-to-head time-based comparison would have been grossly unfair to the achievements of previous work. For the more recently reported results (from 2010), even were timing data available, differences in techniques, platforms and configurations would also have made comparison problematic. In the present paper, we report on our execution times, configuration and platform details in order to support potential backward comparison in future work on strong and higher order mutation testing.

Mutation testing is generally regarded as a comparatively slow and expensive approach to testing. Despite this, it has endured through more than three decades as a research topic, perhaps because of results that demonstrate that it provides a particularly demanding test adequacy criterion and one that is attractively generic and flexible.

Given these historical perspectives, our timing findings are encouraging because they indicate that weak, strong and higher order mutation testing can all be used to generate test data within reasonable time on a standard laptop. Generation of test data by hand (the only currently available alternative for either strong or higher order mutation) would take considerably longer, and, using human effort rather than machine effort would be (perhaps prohibitively) more expensive.

6. RELATED WORK

Constraint Based Testing (CBT) was the first test data generation technique used for mutation testing. It was proposed by DeMillo and Offutt, based on the idea of control flow analysis and symbolic execution [8, 30]. Constraint based testing seeks to generate test data to kill mutants weakly by reaching and infecting mutants, thereby achieving the 'R' and 'I' of the 'RIP' framework described in Section 2. Offutt and DeMillo represent reachability as a set of path conditions, constructed using control flow analysis and symbolic execution and augment these path constraints with constraints that denote infection.

The initial approach to CBT suffered from several problems inherited from the state-of-the-art in symbolic evaluation available at the time and also from the static domain reduction technique used. It was unable to handle arrays, loops and nested expressions well. To overcome these restrictions, Offutt et al. proposed a dynamic domain reduction technique [31, 32]. The dynamic domain reduction technique uses a more sophisticated back-tracker to dynamically split domains.

Dynamic Symbolic Execution (DSE) [12, 39] is a more recent innovation that overcomes many of the limitations of traditional symbolic execution. Using DSE, non-linear path constraints are simplified by the instantiation of concrete runtime values, harvested from program execution. DSE has been used in several coverage based testing tools, such as DART [12], CUTE [39] and Pex [40].

DSE also provides a natural way to generate weakly adequate mutation-based test inputs. A simple testability transformation [15] can be used to augment the program with conditional statements, the predicates of which capture the infection constraints. By construction, covering the branches of the transformed program entails satisfying these infection constraints, thereby tranforming branch coverage into weak mutation coverage. This approach was first suggested by Liu et al. [26], and was implemented by Zhang et al. [42] and Papadakis et al. [36] who evaluated it on small programs, such as those found in Table 3.

Search Based Software Testing (SBST) [2, 17] has also been applied to the generation of weakly adequate mutation-based test data. Bottaci was the first to suggest using SBSE to kill mutants [6]. However, Search Based Mutation Test Generation remained unimplemented and unevaluated until the subsequent for work of Ayari et al. [4] and Fraser and Zeller [11], both of whom target Java. This previous work is not directly comparable to SHOM since SHOM targets C.

Ayari et al. evaluated their approach on small programs; essentially these are Java versions of those programs used in the previous work on mutant-based test data generation for Fortran and C, the largest of which is 70 Lines of Code. The more recent work by Fraser and Zeller reports results from a larger-scale study involving two Java programs, the larger of which contains 412 classes.

The SHOM approach introduced in this paper combines DSE and SBST. It uses DSE to achieve weak mutation adequacy and extends this with a constraint-aware search based approach that maintains weak adequacy, while seeking to propagate tests to achieve strong mutation adequacy. SHOM thus extends previous work by generating test data for strong mutation adequacy and by generating test data for higher order mutants. Like the recent work of Fraser and Zeller on Java, SHOM is evaluated on much larger examples; an order of magnitude larger than previous work on mutant-based test data generation for C.

7. CONCLUSIONS

In this paper we introduced a hybrid DSE and SBST approach to generate strongly adequate test data to kill first and higher order mutants. We implemented our approach in a tool called SHOM. We also implemented the two previously published approaches, based on reachability alone and reachability together with infection and used these implementations to evaluate our approach on 17 example programs. Our results show that SHOM is able to achieve higher levels of strong mutation coverage than either previously published approach for first order mutants. For second order mutants there is no previous work on test data generation so we compared our second order test sets with those composed from the union of the corresponding first order sets. Once again, SHOM was found to outperform approaches based on either reachability alone or reachability and infection.

8. REFERENCES

[1] H. Agrawal, R. A. DeMillo, B. Hathaway, W. Hsu, W. Hsu, E. W. Krauser, R. J. Martin, A. P. Mathur, and E. Spafford. Design of Mutant Operators for the C Programming Language. Technique Report SERC-TR-41-P, Purdue University, West Lafayette, Indiana, March 1989.

[2] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege. A Systematic Review of the Application and Empirical Investigation of Search-Based Test-Case Generation. *IEEE Transactions on Software Engineering*, 36(6):742–762, 2010.

[3] D. C. Atkinson and W. G. Griswold. The Design of Whole-Program Analysis Tools. In *International Conference on Software Engineering (ICSE '96)*, pages 16–27, 1996.

[4] K. Ayari, S. Bouktif, and G. Antoniol. Automatic Mutation Test Input Data Generation via Ant Colony. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'07)*, pages 1074–1081, London, England, 7-11 July 2007.

[5] F. Belli, N. Güler, A. Hollmann, G. Suna, and E. Yõldõz. Model-Based Higher-Order Mutation Analysis. In *Advances in Software Engineering*, volume 117 of *Communications in Computer and Information Science*, pages 164–173. Springer Berlin Heidelberg, 2010.

[6] L. Bottaci. A genetic algorithm fitness function for mutation testing. In *Proceedings of the 8th Wrokshop on Software Engineering using Metaheuristic INovative Algorithms (SEMINAL'01)*, pages 3–7, 2001.

[7] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer*, 11(4):34–41, April 1978.

[8] R. A. DeMillo and A. J. Offutt. Constraint-Based Automatic Test Data Generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

[9] H. Do, S. Elbaum, and G. Rothermel. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. *Empirical Software Engineering*, 10(4):405 – 435, Oct. 2005.

[10] B. Dutertre and L. M. de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *Proceedings of the 18th International Conference on Computer Aided Verification (CAV'06)*, pages 81–94, 2006.

[11] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. In *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA'10)*, pages 147–158, Trento, Italy, 12-16 July ISSTA '10. ACM.

[12] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*, volume 40 of *6*, pages 213–223, Chicago, Illinois, USA, 11–15 June 2005. ACM.

[13] R. G. Hamlet. Testing Programs with the Aid of a Compiler. *IEEE Transactions on Software Engineering*, 3(4):279–290, July 1977.

[14] M. Harman, Y. Hassoun, K. Lakhotia, P. McMinn, and J. Wegener. The Impact of Input Domain Reduction on Search-Based Test Data Generation. In *ACM Symposium on the Foundations of Software Engineering (FSE '07)*, pages 155–164, Dubrovnik, Croatia, September 2007.

[15] M. Harman, L. Hu, R. M. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper. Testability Transformation. *IEEE Transactions on Software Engineering*, 30(1):3–16, Jan. 2004.

[16] M. Harman, Y. Jia, and W. B. Langdon. A Manifesto for Higher Order Mutation Testing. In *Proceedings of the 5th International Workshop on Mutation Analysis (MUTATION'10)*, Paris, France, 6 April 2010. IEEE Computer Society. published with *Proceedings of the 3rd International Conference on Software Testing, Verification, and Validation Workshops*.

[17] M. Harman and B. F. Jones. Search-based Software Engineering. *Information and Software Technology*, 43(14):833–839, December 2001.

[18] M. Harman and P. McMinn. A Theoretical and Empirical Analysis of Evolutionary Testing and Hill Climbing for Structural Test Data Generation. In *International Symposium on Software Testing and Analysis (ISSTA'07)*, pages 73 – 83, London, United Kingdom, July 2007.

[19] M. Harman and P. McMinn. A Theoretical and Empirical Study of Search-Based Testing: Local, Global, and Hybrid Search. *IEEE Transactions on Software Engineering*, 36(2):226–247, 2010.

[20] Y. Jia and M. Harman. Constructing Subtle Faults Using Higher Order Mutation Testing. In *Proceedings of the 8th International Working Conference on Source Code Analysis and Manipulation (SCAM'08)*, pages 249–258, Beijing, China, 28-29 September 2008.

[21] Y. Jia and M. Harman. Higher Order Mutation Testing. *Journal of Information and Software Technology*, 51(10):1379–1393, October 2009.

[22] Y. Jia and M. Harman. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions of Software Engineering*, Available online from IEEE Explore, 2011.

[23] K. Lakhotia, P. McMinn, and M. Harman. Automated Test Data Generation for Coverage: Haven't We Solved This Problem Yet? In 4^{th} *Testing Academia and Industry Conference — Practice And Research Techniques (TAIC PART'09)*, pages 95–104, Windsor, UK, 4th–6th September 2009.

[24] K. Lakhotia, N. Tillmann, M. Harman, and J. de Halleux. FloPSy — Search-Based Floating Point Constraint Solving for Symbolic Execution. In 22^{nd} *IFIP International Conference on Testing Software and Systems (ICTSS 2010)*, pages 142–157, Natal, Brazil, November 2010. LNCS Volume 6435.

[25] W. B. Langdon, M. Harman, and Y. Jia. Efficient multi-objective higher order mutation testing with genetic programming. *Journal of systems and Software*, 83:2416–2430, December 2010.

[26] M.-H. Liu, Y.-F. Gao, J.-H. Shan, J.-H. Liu, L. Zhang, and J.-S. Sun. An Approach to Test Data Generation for Killing Multiple Mutants. In *Proceedings of the 22nd IEEE International Conference on Software Maintenance (ICSM'06)*, pages 113–122, Philadelphia, Pennsylvania, USA, 24-27 September 2006.

[27] P. Madiraju and A. S. Namin. ParaMu - A Partial and Higher-Order Mutation Tool with Concurrency Operators. In *Proceedings of the 6th International Workshop on Mutation Analysis (Mutation 2011)*, Berlin, Germany, March 2011.

[28] L. J. Morell. A Theory of Fault-Based Testing. *IEEE Transactions on Software Engineering*, 16(8):844–857, August 1990.

[29] G. Necula, S. McPeak, S. Rahul, and W. Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In R. Horspool, editor, *Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 209–265. Springer Berlin / Heidelberg, 2002.

[30] A. J. Offutt. *Automatic Test Data Generation*. Phd thesis, Georgia Institute of Technology, Atlanta, GA, USA, 1988.

[31] A. J. Offutt, Z. Jin, and J. Pan. The Dynamic Domain Reduction Approach for Test Data Generation: Design and Algorithms. Technical Report ISSE-TR-94-110, George Mason University, Fairfax, Virginia, 1994.

[32] A. J. Offutt, Z. Jin, and J. Pan. The Dynamic Domain Reduction Procedure for Test Data Generation. *Software:Practice and Experience*, 29(2):167–193, February 1999.

[33] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An Experimental Determination of Sufficient Mutant Operators. *ACM Transactions on Software Engineering and Methodology*, 5(2):99–118, April 1996.

[34] M. Papadakis and N. Malevris. An Effective Path Selection Strategy for Mutation Testing. In *Proceedings of the 16th Asia-Pacific Software Engineering Conference (APSEC'09)*, pages 422 – 429, Penang, Malaysia, 1-3 December 2009. IEEE Computer Society.

[35] M. Papadakis and N. Malevris. An Empirical Evaluation of the First and Second Order Mutation Testing Strategies. In *Proceedings of the 5th International Workshop on Mutation Analysis (MUTATION'10)*, Paris, France, 6 April 2010. IEEE Computer Society. published with *Proceedings of the 3rd International Conference on Software Testing, Verification, and Validation Workshops*.

[36] M. Papadakis and N. Malevris. Automatic mutation test case generation via dynamic symbolic execution. In *Proceedings of the 21st International Symposium on Software Reliability Engineering (ISSRE'10)*, California, USA, November 2010.

[37] M. Papadakis, N. Malevris, and M. Kallia. Towards Automating the Generation of Mutation Tests. In *Proceedings of the 5th Workshop on Automation of Software Teste (AST'10)*, pages 111–118, Cape Town, South Africa, 3-4 May 2010. ACM.

[38] M. Polo, M. Piattini, and I. Garcia-Rodriguez. Decreasing the Cost of Mutation Testing with Second-Order Mutants. *Software Testing, Verification and Reliability*, 19(2):111 – 131, June 2008.

[39] K. Sen, D. Marinov, and G. Agha. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'05)*, pages 263–272, Lisbon, Portugal, 2005.

[40] N. Tillmann and J. de Halleux. Pex–White Box Test Generation for .NET. In *Proceedings of the 2nd International Conference on Tests and Proofs (TAP'08)*, volume 4966, pages 134–153, Prato, Italy, April 2008.

[41] M. R. Woodward. Errors in Algebraic Specifications and an Experimental Mutation Testing Tool. *Software Engineering Journal*, 8(4):221–224, July 1993.

[42] L. Zhang, T. Xie, L. Zhang, N. Tillmann, J. de Halleux, and H. Mei. Test generation via dynamic symbolic execution for mutation testing. In *Proceedings of the 26th International Conference on Software Maintenance (ICSM'10)*, Timisoara, Romania, September 2010.