# Mutant Subsumption Graphs

Bob Kurtz[†], Paul Ammann[†], Marcio E. Delamaro[*], Jeff Offutt[†], Lin Deng[†]
[*]Instituto de Ciências Matemáticas e de Computaçao, Universidade de São Paulo, São Carlos, SP, Brazil
[†]Software Engineering, George Mason University, Fairfax, VA, USA
Emails: {rkurtz2,pammann,offutt,ldeng2}@gmu.edu, delamaro@icmc.usp.br

***Abstract*—Mutation testing researchers have long known that many generated mutants are not needed. This paper develops a graph model to describe redundancy among mutations. We define "true" subsumption, a relation that practicing test engineers would like to have, but cannot due to issues of computability. We also define dynamic subsumption and static subsumption as approximations of "true" subsumption. We explore the properties of the approximate subsumption relations in the context of a small example. We suggest possible uses for subsumption graphs.**

***Keywords*—*mutation testing, subsumption***

## I. Introduction

Mutation testing [8] is a test criterion that can be used to design test sets or evaluate test sets by generating a set of alternate programs, or *mutants*, and determining how many of these mutants are detected by the test suite. Mutants are typically generated by mutation operators, but the alternate programs are not limited to such approaches. For example, in Automated Program Repair (APR), alternate programs may find potential "patches" to repair the program by incorporating fragments of code from elsewhere in the program, snippets generated by learning algorithms, or code generated from specifications[1].

The notion of subsumption has traditionally been used to compare test criteria: a criterion C1 subsumes C2 if every set of tests that satisfy C1 also satisfy C2 [2]. First Kuhn [26], then Lau and Yu [28], and later Kaminski and Ammann [21] used the concept (although not the term) to create hierarchies of faults, where tests that detect certain faults are guaranteed to detect others. Jia and Harman [18] used subsumption to distinguish between useful and non-useful higher order mutants. This paper uses subsumption in a more narrow way: one mutant *subsumes* another if at least one test kills the first and every test that kills the first also kills the second. This differs from *operator subsumption*, in which a mutation operator $op_1$ subsumes another, $op_2$, if tests that kill all mutants created from $op_1$ also kill all mutants created from $op_2$.

Subsumption relationships identify *redundancy* in sets of mutants and hence can be used to optimize approaches to both mutant and test generation. Specifically, subsumed mutants may not need to be generated, and test generation methods can target subsuming mutants. Thus, more knowledge about subsumption can help us build more efficient mutation testing tools, significantly improving the practical applicability of mutation in industry.

[1]Le Goues et al. [13] provide a recent overview of the state of the art in APR, including a summary of approaches to generating candidate program patches and a list of top research challenges.

Subsumption has potential benefit outside of the traditional application of testing. For example, one of the key challenges in APR is searching for program patches that change behavior for the tests that fail without changing behavior for tests that pass. Some APR approaches have implemented this search with genetic algorithms, but the subsumption relation potentially offers a much richer and semantically meaningful structure to search. Hence, despite the difficulties in computing the subsumption relation, there are strong motivations to engineer approaches to computing it exactly when possible and approximating it when not.

The problem with subsumption is the usual problem with general approaches to program analysis and testing: determining a subsumption relationship between two mutants is undecidable in general, and often challenging for cases where it can be computed. Hence, analysis approaches to subsumption can only offer partial answers.

Dynamic subsumption [1] provides a black-box approach to approximating the subsumption relationship based on the observed behavior of mutants with respect to a specific test set. One important result is that dynamic subsumption can be used to minimize sets of mutants with respect to that test set. Empirical observations on the Siemens suite [9], [17] showed that the number of redundant mutants, even when using selective mutation approaches, is quite large. This large gap suggests a need for more inquiry into the basic properties of both dynamic subsumption and its relation to "true" subsumption. Towards this end, this paper probes the dynamic subsumption relationship for a specific program. Specifically, we look at the subsumption relationship between mutants in the form of a directed graph. For the example program, we examine how the graph evolves as the mutants are incrementally exposed to tests. In the context of the example, we examine how static analysis can help refine the graph.

This paper differentiates between three types of subsumption. *True subsumption* assumes full knowledge of the relationships among mutants and though valuable as a concept, is undecidable to compute, either through enumeration or analysis. *Dynamic subsumption* is computed relative to a specific set of tests. As the number of tests tends towards the entire domain of the artifact under test (not possible, of course), dynamic subsumption approaches "true" subsumption. Alternatively, *static subsumption* is computed based on an analysis of the mutants, either by hand or automated. Again, as the analysis tends toward capturing the complete semantics of the artifact under test, (again not possible), static subsumption also approaches "true" subsumption.

This paper is organized as follows. Section II develops the "true" subsumption relation and defines the "true" subsumption graph. In section III, we define the dynamic subsumption graph and investigate its properties with a detailed example. In section IV we define the static subsumption graph and explore how it can approximate "true" subsumption in the context of the example. Section V presents related work. Sections VI and VII discuss conclusions and future work.

## II. Subsumption Graphs

Even though "true"' subsumption is not computable, it is useful to define it formally so as to provide a goal for the approximation approaches. In this paper, the term *subsume* without any modifiers means "true" subsumption.

Let $M$ be a set of mutants on some artifact $A$. Denote an element of $M$ as $m$, possibly subscripted. We say that $m_i$ *subsumes* $m_j$, denoted $m_i \rightarrow m_j$, iff:

1) There exists some test $t$ such that $m_i$ and $A$ compute different outcomes on $t$ ($t$ kills $m_i$).
2) For every possible test $t$ for $A$, if $m_i$ computes a different outcome than $A$ on $t$, then so does $m_j$.

Three things are notable about the definition. First, "true" subsumption considers the entire input domain of $A$. That is, there are no limits on the test set.

Second, because of the first property, the definition does not allow vacuous subsumption. In other words, equivalent mutants[2] are not part of the "true" subsumption relation. This is different from Jia and Harman's definition of subsuming HOMs [18], which only requires the second property. That is, we require that we have a test that kills the subsuming mutant, whereas they do not. This matters because if a mutant subsumes another, but is equivalent to the original program (and thus cannot be killed), the subsumption relationship is irrelevant.

Third, the subsumption relation naturally forms a graph, which we call the *Mutation Subsumption Graph* or MSG. We defer the precise definition of the graph until we discuss some additional terms.

Let $M_{killable}$ denote the set of non-equivalent mutants. A *minimal* set of mutants, $M_{minimal}$, is a subset of $M_{killable}$ with the following two properties:

1) Any test set that kills each mutant in $M_{minimal}$ also kills each mutant in $M_{killable}$.
2) If any mutant is removed from $M_{minimal}$, the first property no longer holds.

Note that minimality for mutants is defined in terms of *all* possible test sets. Section III gives a related definition that holds with respect to a particular, fixed test set.

Consider two mutants $m_i$ and $m_j$ in $M$ that compute exactly the same function for all possible tests. We say that $m_i$ and $m_j$ are *indistinguishable*[3]. From an analysis perspective, $m_j$ adds no benefit beyond $m_i$, and hence is redundant. With this observation, we are now ready to define the MSG:

1) Nodes in the MSG are maximal sets of indistinguishable mutants.
2) Edges in the MSG represent the subsumption relation.

Note that edges in the MSG only connect sets of mutants in $M_{killable}$. All equivalent mutants occupy a single, isolated node in the graph.

If we ignore the equivalent mutants, then the remaining root nodes in the MSG capture everything needed to test artifact $A$. In particular, if we restrict attention to $M_{killable}$ and choose one mutant from each root node, then the resulting mutant set is minimal.

In drawing such graphs, it is common to show only the transitive reduction of the subsumption relation; otherwise the graphs quickly become unreadable.

## III. Dynamic Subsumption Graphs

We have no way to compute the "true" subsumption graph, so we define *dynamic subsumption* to be an approximation to subsumption that depends a specific set of test cases. We build on the notion of dynamic subsumption to define dynamic subsumption graphs and dynamic minimal mutant sets.

Our previous paper [1] defined a score function that specifies which mutants are killed by each test. Given a finite set of mutants $M$ and a finite set of tests $T$, the score function $S$ is a binary matrix with $|T|$ rows and $|M|$ columns, where $S(i,j)$, $i = 1,..,|T|$, $j = 1,..,|M|$, is true *iff* test $t_i$ kills mutant $m_j$. Thus, the matrix represents which mutants each test in $T$ kills. A mutant $m_x$ is said to *dynamically subsume* mutant $m_y$ if some test in $T$ kills $m_x$ and any test in $T$ that kills $m_x$ also kills $m_y$. That is, $m_x$ dynamically subsumes $m_y$ with respect to $T$, again denoted $m_x \rightarrow m_y$, iff $S(i,x)$ is true for at least one $i$ in $1..|T|$ and $S(i,x) \Rightarrow S(i,y)$, $\forall i = 1..|T|$, where "$\Rightarrow$" is the logical implication operator.

Recall that if two mutants $m_x$ and $m_y$ compute exactly the same function for *all* possible tests, then $m_x$ and $m_y$ are *indistinguishable*. We refine this notion for the context of a specific test set: if two mutants are killed by the same subset of test set $T$, the mutants are considered to be *indistinguished* (thus far), regardless of semantic differences in the mutated code. Similarly, if two tests in $T$ kill precisely the same set of mutants, then the tests are considered to be *indistinguished*.

We capture the dynamic subsumption relationship among mutants with a directed graph, the *Dynamic Mutant Subsumption Graph* or DMSG. Each node in the DMSG represents a maximal set of indistinguished mutants and edges in the DMSG represent the dynamic subsumption relationship. More specifically, if $m_x$ dynamically subsumes $m_y$, then there is an edge from the node containing $m_x$ to the node containing $m_y$. Note that since the DMSG depends on a specific test set $T$, it is *not* the same as the MSG restricted to $M_{killable}$, which is defined in terms of all possible tests. Instead, the DMSG approximates the MSG, a process we explore in the next section.

A *dynamic minimal mutant set* is defined with respect to a particular test set $T$ and is a subset $M_{dynamic}$ of $M$ such

[2]As usual in mutation testing, $m_i$ is *equivalent* to $A$ if $m_i$ and $A$ compute the same outcome on all possible inputs.

[3]Even though $m_i$ and $m_j$ are equivalent *to each other*, we use the term *indistinguishable* instead of *equivalent* to avoid confusion with common usage in mutation testing. That is, an equivalent mutant computes exactly the same outcome as $A$, but a pair of indistinguishable mutants may compute any outcome.

that the minimal test sets (from $T$) for $M$ and the minimal test sets (also from $T$) for $M_{dynamic}$ are identical [1]. Just as the root nodes of the MSG identify minimal sets of mutants with respect to all possible tests, the root nodes of the DMSG identify dynamic minimal sets of mutants [1].

## A. Dynamic Subsumption Graph Example

Table I shows a score function for an example with four tests, $T = \{t_1, t_2, t_3, t_4\}$, and four mutants, $M = \{m_1, m_2, m_3, m_4\}$. All tests in $T$ and all mutants in $M$ are distinguished–that is, there are no duplicate rows or columns.

TABLE I. SUBSUMPTION MATRIX EXAMPLE

| | $m_1$ | $m_2$ | $m_3$ | $m_4$ |
|---|---|---|---|---|
| $t_1$ | ✓ | ✓ | | ✓ |
| $t_2$ | ✓ | | ✓ | ✓ |
| $t_3$ | | | | ✓ |
| $t_4$ | | ✓ | | ✓ |

Given the definition of dynamic subsumption, every test that kills $m_1$ also kills $m_4$, thus $m_1 \rightarrow m_4$. Likewise, $m_2 \rightarrow m_4$, $m_3 \rightarrow m_1$, and $m_3 \rightarrow m_4$. Drawing these relationships in a graph results in Figure 1. Note that for simplicity in the graph, relationships that can be inferred by transitivity, such as $m_3 \rightarrow m_4$, are not shown. The double circles around $m_2$ and $m_3$ indicate that they are dynamically *minimal*, that is, no other mutant dynamically subsumes them.

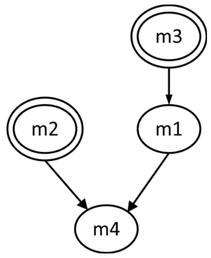

Fig. 1. Example dynamic mutant subsumption graph.

The DMSG makes it easy to identify a dynamically minimal set of mutants–it is simply the root nodes ($\{m_2, m_3\}$ in Figure 1). Any subset of $T$ that kills these mutants will kill all mutants in $M$.

## B. Growth of the Dynamic Subsumption Graph

Dynamic subsumption is based on specific tests. Consequently, the dynamic subsumption relationship varies as individual tests from $T$ are applied to the mutants in $M$. Based on the score function in Table I, we can observe the growth of the DMSG as tests are individually added. This is shown in Figure 2.

After $t_1$ is executed, mutants $m_1$, $m_2$, and $m_4$ are killed but indistinguished and mutant $m_3$ is live. The node containing live mutants–$m_3$ in this case–is shown with a dashed circle. No dynamic subsumption relationship exists at this point. Executing $t_2$ kills all mutants. Mutants $m_2$ and $m_3$ now dynamically subsume $m_1$ and $m_4$, which are indistinguished. Executing $t_3$ distinguishes $m_1$ and $m_4$, with $m_1$ dynamically subsuming $m_4$. Adding $t_4$ also breaks the dynamic subsumption relationship between $m_2$ and $m_1$. As in Figure 1, the

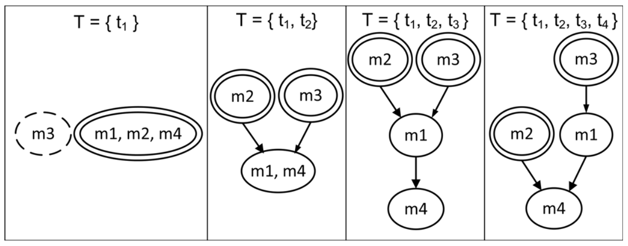

Fig. 2. Growth of the dynamic mutant subsumption graph.

nodes in the dynamically minimal set are drawn with double circles.

## C. Subsumption State Model

When constructing a DMSG, certain characteristics of a mutant can change as additional tests are added. A mutant may be either live or killed, distinguished or indistinguished, and (dynamically) minimal or subsumed. These characteristics are orthogonal and thus can be in one of several states. Theoretically, eight states are possible, except that a live mutant cannot be dynamically minimal or subsumed, since the dynamic subsumption relationship is not defined for live mutants. Thus, a mutant can be in one of six states:

1) live and distinguished (LD)
2) live and indistinguished (LI)
3) killed, distinguished, and (dynamically) minimal (KDM)
4) killed, distinguished, and subsumed (KDS)
5) killed, indistinguished, and (dynamically) minimal (KIM)
6) killed, indistinguished, and subsumed (KIS)

The mutant state model for dynamic subsumption is shown in Figure 3. Before any tests are run, all mutants are *live* and *indistinguished* (LI). From here, a mutant can move to one of three states. If a test kills more than one mutant in state LI, that mutant is now *killed*, *indistinguished*, and *minimal* (KIM). Note that we use the term "minimal" in a loose sense at this point; since a minimal node may contain more than one mutant, any of those mutants could be considered to be dynamically minimal. If a test kills one mutant in state LI, and no others, that mutant is now *killed*, *distinguished*, and *minimal* (KDM). If a test is added, and all mutants but one are killed, the remaining live mutant will now be *live* and *distinguished* (LD).

If a test kills a single mutant in state LD, the mutant will now be *killed*, *distinguished*, and *minimal* (KDM). Note that a live mutant can never directly become dynamically subsumed, because there cannot be a mutant that is killed by fewer tests than a newly-killed mutant. Thus, there are no transitions from any *live* nodes to any *subsumed* nodes in Figures 3.

Once killed, a *minimal* (KDM or KIM) mutant can become *subsumed* (KDS or KIS), and a *killed subsumed* (KDS or KIS) mutant can become *minimal* (KDM or KIM). Once *distinguished*, a mutant can never become *indistinguished*.

A *killed*, *indistinguished*, and *minimal* (KIM) mutant or a *killed*, *indistinguished*, and *subsumed* (KIS) mutant can become *distinguished* (KDM or KDS) if all other mutants that were *indistinguished* from it are killed by other tests, or if only one of the *indistinguished* mutants is killed. It may retain its *minimal/subsumed* property, or that property may reverse. Transitions to the same state are always possible, but for simplicity these transitions are not shown.

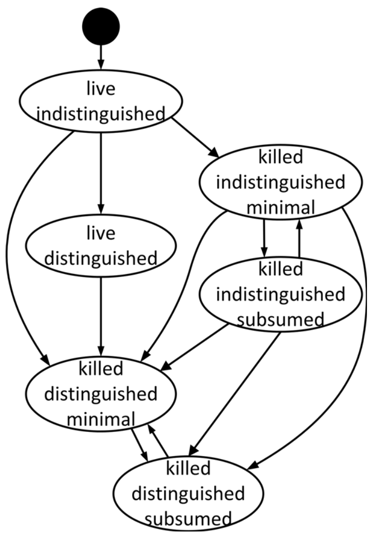
live indistinguished
killed indistinguished minimal
live distinguished
killed indistinguished subsumed
killed distinguished minimal
killed distinguished subsumed

Fig. 3. Mutant state model for dynamic subsumption.

Tables II through VI provide simple examples to illustrate each state transition. In these examples, the state change refers to that of mutant $m_1$ as a result of the last test.

TABLE II. Transitions LI → LD, LI → KDM, & LD → KDM

LI → LD

| | $t_1$ |
|---|---|
| $m_1$ | |
| $m_2$ | ✓ |

LI → KDM

| | $t_1$ |
|---|---|
| $m_1$ | ✓ |
| $m_2$ | |

LD → KDM

| | $t_1$ | $t_2$ |
|---|---|---|
| $m_1$ | | ✓ |
| $m_2$ | ✓ | |

TABLE III. Transitions LI → KIM, KDM → KDS, & KIM → KIS

LI → KIM

| | $t_1$ |
|---|---|
| $m_1$ | ✓ |
| $m_2$ | ✓ |

KDM → KDS

| | $t_1$ | $t_2$ |
|---|---|---|
| $m_1$ | ✓ | ✓ |
| $m_2$ | ✓ | |

KIM → KIS

| | $t_1$ | $t_2$ |
|---|---|---|
| $m_1$ | ✓ | ✓ |
| $m_2$ | ✓ | ✓ |
| $m_3$ | ✓ | |

TABLE IV. Transitions KDS → KDM & KIS → KIM

KDS → KDM

| | $t_1$ | $t_2$ | $t_3$ |
|---|---|---|---|
| $m_1$ | ✓ | ✓ | |
| $m_2$ | | ✓ | ✓ |

KIS → KIM

| | $t_1$ | $t_2$ | $t_3$ |
|---|---|---|---|
| $m_1$ | ✓ | ✓ | |
| $m_2$ | ✓ | ✓ | |
| $m_3$ | ✓ | | ✓ |

TABLE V. Transitions KIM → KDM & KIM → KDS

KIM → KDM

| | $t_1$ | $t_2$ |
|---|---|---|
| $m_1$ | ✓ | |
| $m_2$ | ✓ | ✓ |

KIM → KDS

| | $t_1$ | $t_2$ |
|---|---|---|
| $m_1$ | ✓ | ✓ |
| $m_2$ | ✓ | |

## *D. Dynamic Subsumption Example in Java*

This section illustrates a DMSG on the `cal()` method, a program from Ammann and Offutt's testing textbook [2].

TABLE VI. Transitions KIS → KDM & KIS → KDS

KIS → KDM

| | $t_1$ | $t_2$ | $t_3$ |
|---|---|---|---|
| $m_1$ | ✓ | ✓ | |
| $m_2$ | ✓ | ✓ | ✓ |
| $m_3$ | ✓ | | ✓ |

KIS → KDS

| | $t_1$ | $t_2$ | $t_3$ |
|---|---|---|---|
| $m_1$ | ✓ | ✓ | ✓ |
| $m_2$ | ✓ | ✓ | |
| $m_3$ | ✓ | | |

`cal()` calculates the number of days between two given days in the same year. We use `cal()` because it is small enough to be be comprehensively tested with a relatively small test set, but complex enough to generate interesting subsumption graphs. The signature for `cal()` is:

```
public int cal (int month1, int day1,
                int month2, int day2, int year)
 preconditions :
       1 <= month1 <= month2 <= 12
       day1, day2 appropriate for month1, month2
       day1, day2 in the same year
       1 <= year <= 10000
```

Note that the preconditions on `cal()` only allow valid inputs, which greatly reduces the complexity of `cal()`, since error checking is not required. The restriction to valid inputs also bounds the portion of the input domain from which tests are drawn. We designed a test set using input space partitioning [2]. The input domain model (IDM) is:

- month1, month2: 1 .. 12
- day1, day2: 1 .. 31
- year: { divisible-by-400, divisible-by-100, divisible-by-4, not-divisible-by-4 }
- constraints:
  month2 >= month1
  day1, day2 <= length of the corresponding months
  if month1 == month2, day2 >= day 1
  1 <= year <= 10000

We used the Advanced Combinatorial Testing System (ACTS) [20] to generate pairwise combinations of months and years, with days selected at random (subject to the above constraints). ACTS generated 90 test cases for our IDM. For the remainder of the paper, we refer to this test set as the "pairwise test set."

We then used the muJava mutation analysis tool [29] to create the DMSG. muJava normally removes mutants from consideration after they are killed, but we needed to compute **all** tests that killed each mutant. So we modified muJava in two ways. First, we created a command line interface[4], then we added an option to execute tests on all mutants, dead or alive. muJava generated 173 mutants for `cal()`.

We executed the test cases with muJava to generate a score function as described in section III, then analyzed the score function to generate dynamic subsumption relationships as tests were added. The tests killed 145 mutants, and we analyzed the remaining 28 mutants by hand and determined that all were equivalent. muJava uses 11 statement level mutation operators [32], as shown in Table VII. Later in the paper, some of these acronyms have a 'B,' 'U,' or 'S,' at the end (e.g., AORB), to indicate whether the mutation operator

[4]This will soon be released on the muJava website [32].

applies to a binary operation, unary operation, or short-circuit expression.

TABLE VII. 11 METHOD-LEVEL MUTATION OPERATORS FOR JAVA

| Operator | Description |
|---|---|
| AOR | Arithmetic Operator Replacement |
| AOI | Arithmetic Operator Insertion |
| AOD | Arithmetic Operator Deletion |
| ROR | Relational Operator Replacement |
| COR | Conditional Operator Replacement |
| COI | Conditional Operator Insertion |
| COD | Conditional Operator Deletion |
| SOR | Shift Operator Replacement |
| LOR | Logical Operator Replacement |
| LOI | Logical Operator Insertion |
| LOD | Logical Operator Deletion |
| ASR | Assignment Operator Replacement |

The final DMSG resulting from executing the pairwise test set is shown in Figure 4. Each node represents a set of one or more mutants that are indistinguished from each other. The node labels give the mutation operator that created the (first) mutant in that node, and how many additional mutants are in the node. The equivalent mutants are in the top left node with a dashed oval. The first equivalent mutant is *AOIS*, and there are 27 additional equivalent mutants. The graph contains seven nodes of dynamically minimal mutants, as marked with double ovals. That is, even though muJava generated 173 mutants, we need **only seven** to design tests that ensure 100% mutation score on all mutants. Note that some of these minimal nodes contain multiple indistinguished mutants, so this graph yields 128 possible sets of minimal mutants.

We then identified all of the minimal test sets in the pairwise test set that were sufficient to kill all dynamically minimal mutant sets. While in general the problem of identifying all minimal test sets is NP-complete [1], the `cal()` method is relatively simple so we were able to use a brute-force algorithm to identify 21,960 minimal test sets from our set of 90 tests.

It is clear from the growth of DMSGs as shown in Figure 2 that additional tests tend to distinguish more mutants, making the graph more complex. To investigate this further, we repeated the analysis with both a smaller and larger set of tests. For the smaller test set, we selected one of the identified minimal test sets, which contained six tests. For the larger test set, we again used ACTS to develop a test set using full combinatorial rather than pairwise coverage of months and years, resulting in 312 tests (the "combinatorial test set"). The three test sets are compared in Table VIII. With the minimal test set, the resulting DMSG was somewhat simpler, though nearly all of the minimal mutant nodes were identified. With the combinatorial test set, the resulting DMSG was only slightly more complex than for the pairwise, and contained only two more groups of dynamically minimal mutants.

While the DMSG created from the larger combinatorial test set is closer to the "true" MSG, it may still be missing nodes and edges that would be produced by some even larger test set. The small increase in graph complexity caused by the large increase in the number of tests suggests that the impact of additional tests on graph complexity decreases once a mutation-adequate test set has been achieved.

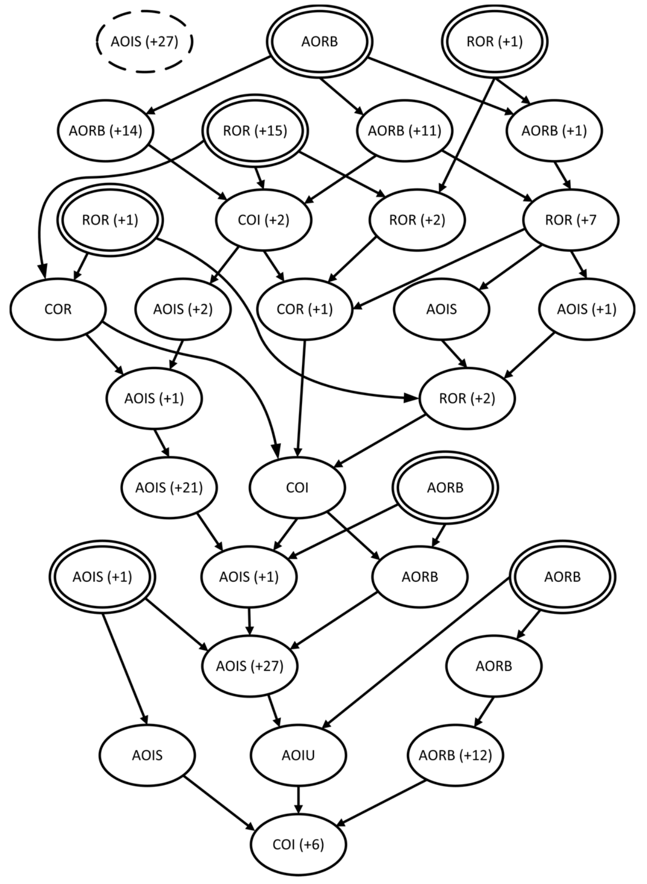

Fig. 4. Final dynamic mutant subsumption graph of `cal()`, based on the pairwise test set.

TABLE VIII. MINIMAL TESTS FOR `CAL()` USING PAIR-WISE AND COMBINATORIAL COVERAGE

| | Tests | Mutant Nodes | Minimal Mutant Nodes |
|---|---|---|---|
| Minimal | 6 | 16 | 6 |
| Pairwise | 90 | 31 | 7 |
| Comb-inatorial | 312 | 33 | 9 |

## *E. Dynamic Subsumption Example in C*

To determine whether a similar DMSG would be generated if different mutation operators were applied, we implemented `cal()` in C and used the Proteum mutation analysis tool [7] to generate and run a set of mutants against the same test cases used for the muJava mutants. muJava's mutation operators are based on the selective set [31], whereas Proteum's are not. Thus Proteum generated 891 mutants as compared to muJava's 173.

Although the Proteum mutant set was much larger, the same 90 test cases in the pairwise test set killed all but 71 mutants, all of which were hand-determined to be equivalent. This strongly suggests that the Proteum mutant set has a large amount of redundancy, as well as confirming the value of selective mutation. Intuitively we expected to see a fairly similar DMSG, with a similar number of dynamically minimal mutant sets. Instead, we found that the graph was astonishingly

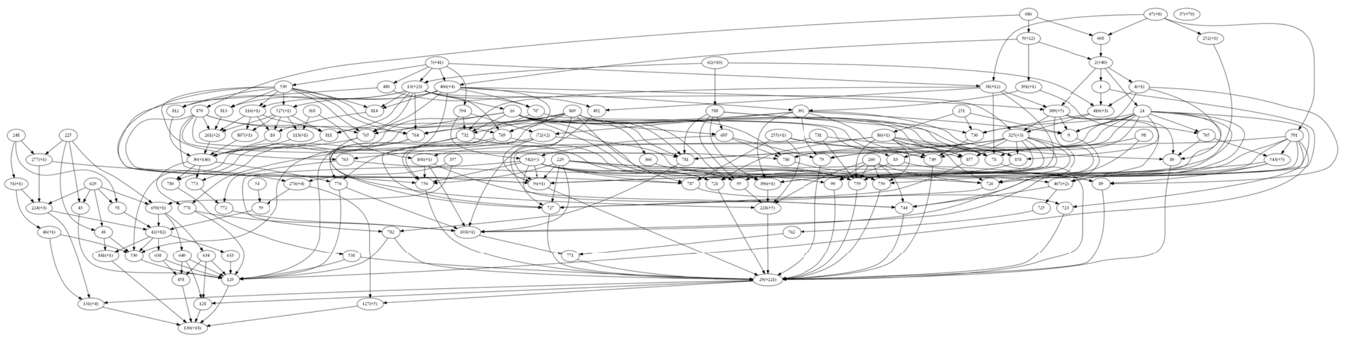

Fig. 5. Final dynamic subsumption graph of `cal()`'s Proteum mutants, based on the pairwise test set.

complex, as summarized in Table IX and Figure 5. Note that the graph has far too many nodes to put in a paper in a way that is completely readable, so this figure attempts to illustrate the complexity of the graph in an abstract thumbnail form.

This suggests that graph complexity is driven primarily by the number of mutants, not the number of tests. We could not determine minimal test sets for Proteum's mutants with a brute-force approach. Thus we used a sampling technique[5] to generate 100 minimal test sets, and found that the sizes of the minimal test sets were somewhat larger on average, with 7.2 tests per minimal test set for Proteum as compared to 5.9 tests per minimal test set for muJava. None of the minimal test sets identified for the Proteum mutants were exactly duplicated in the minimal test sets for the muJava mutants.

TABLE IX. COMPARING MUJAVA / JAVA WITH PROTEUM / C

| Language / Tool | Mutants | Tests | Mutant Nodes | Minimal Mutant Nodes | Avg Test Set Size |
|---|---|---|---|---|---|
| Java / muJava | 173 | 90 | 31 | 7 | 5.9 |
| C / Proteum | 891 | 90 | 128 | 18 | 7.2 |

## IV. STATIC SUBSUMPTION GRAPHS

Just as a DMSG can be constructed relative to a specific test set, it is possible to construct a *Static Mutant Subsumption Graph* or SMSG, by using static analysis.

Static analysis can identify whether $m_x$ and $m_y$ are distinguishable with three possible outcomes:

- Distinguishable
- Not distinguishable
- Unknown

If $m_x$ and $m_y$ are known to be distinguishable, then they are in separate nodes in the SMSG. If they are known to be indistinguishable, then they are in the same node in the SMSG. If the relationship is unknown, then we cannot place them in the SMSG.

Static analysis can also identify satisfaction of the "true" subsumption properties. More formally, consider a pair of mutants $m_x$ and $m_y$. For each of the two properties required of the "true" subsumption, static analysis results in one of three outcomes with respect to that property:

1) First property: There exists some test that kills $m_x$.
   - Holds
   - Does not hold
   - Unknown
2) Second property: Every test that kills $m_x$ also kills $m_y$.
   - Holds
   - Does not hold
   - Unknown

If static analysis finds that both "true" subsumption properties hold and $m_x$ and $m_y$ are distinguishable, we say that $m_x$ statically subsumes $m_y$. Further, if $m_x$ statically subsumes $m_y$, then it must be the case that $m_x \rightarrow m_y$ in the "true" subsumption relation. In other words, static subsumption is sound: in contrast to the DMSG, edges in the SMSG always correspond to edges in the MSG.

Discovery of just one of the two subsumption properties in isolation can be useful. For example, if mutants $m_x$ and $m_y$ have the same reachability conditions, and the infection condition for $m_x$ implies the infection condition for $m_y$, then we can conclude that all tests that kill $m_x$ also kill $m_y$. Then if $m_y$ is determined (perhaps by manual analysis) to be equivalent, it is not necessary to analyze $m_x$, since it must also be equivalent.

Static analysis can also show that one of the two subsumption properties is not satisfied. This finding can be used to remove unsound edges from the DMSG. We illustrate this process in the example below.

To show how static subsumption applies to our example, we selected a small subset of the DMSG for manual analysis, as shown in Figure 6.

The subset is taken from the lower right corner of the full DMSG in Figure 4. For ease of reference, we have added muJava's mutant "number," an internal index it uses, to each node. *AORB 3* is in the dynamically minimal set of mutants, and subsumes the mutants in the other nodes (a total of 20 mutants). For reference, Figure 7 contains a piece of the correct code from `cal()`, with five embedded mutants shown. The original statements have line numbers and the mutants are

[5]The sampling proceeded as follows. First we drew test cases at random from the pairwise test set until an adequate test set was obtained. Then we examined, in random order, each test in the resulting set. If the test was necessary for adequacy, we left it in. Otherwise, we removed it. The result was a minimal set.

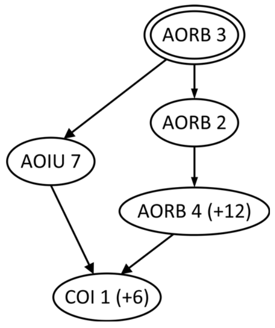

Fig. 6. Partial dynamic subsumption graph.

shown with the Δ sign and a comment indicating the mutation operator and muJava's number. That is, Figure 7 represents six different programs, the original and five mutants.

```
1 if (month2 == month1)
Δ if (!(month2 == month1)) // COI 1
2     numDays = day2 - day1;
Δ     numDays = day2 / day1; // AORB 2
Δ     numDays = day2 % day1; // AORB 3
Δ     numDays = day2 + day1; // AORB 4
3 else
4 {
5     ...
6 }
7 return numdays;
Δ return -numdays; // AOIU 7
```

Fig. 7. Part of the source code of `cal()` , with five mutants shown.

Mutant *COI 1* and the six indistinguished mutants in the same node are killed by every test. These mutants cause an incorrect execution path regardless of the input test data, such as the change for *COI 1* that reverses the state of the month comparison. These mutants cannot be distinguished, and every other mutant will subsume them.

Mutant *AOIU 7* negates the return value, and is killed by every test for which the difference in days is non-zero; i.e., when month2 $\neq$ month1 or month2 $==$ month1 $\wedge$ day2 $\neq$ day1. This clearly subsumes being killed by all tests, so *AOIU 7* subsumes *COI 1*.

Mutant *AORB 4* is killed when month2 $==$ month1, but not when month2 $\neq$ month1. This kill signature is a subset of *COI 1*'s signature, so *AORB 4* subsumes *COI 1*. However, more tests exist that kill *AORB 4* than *AOIU 7*, so no subsumption relationship holds between *AORB 4* and *AOIU 7*. The 12 additional dynamically indistinguished mutants in this node were not addressed in this static example.

Mutant *AORB 2* is killed when month2 $==$ month1 and (day2 / day1) $\neq$ (day2 - day1). This allows some tests to pass when month2 $==$ month1, specifically when day2 $==$ day1 + 1. This is clearly a subset of *AORB 4*, so *AORB 2* subsumes *AORB 4* but not *AOIU 7*.

Mutant *AORB 3* is killed when month2 $==$ month1 and (day2 $\%$ day1) $\neq$ (day2 - day1). Clearly this mutant is not killed when day2 $==$ day1, and due to integer rounding, it is not killed by several other cases as well, so it subsumes *AOIU 7*. At first glance, it is difficult to see statically how this mutant relates to *AORB 2*, so a cursory static evaluation might not reveal a subsumption relationship between *AORB 3* and *AORB 2*. In fact, there exists a single test case (where day2 $==$ 4 $\wedge$ day1 $==$ 2) that will result in *AORB 3* being killed when *AORB 2* does not. Thus, given a highly capable static analysis functionality (or the right dynamic test case), the dynamic subsumption relationship between *AORB 3* and *AORB 2* would be broken.

This analysis allows us to refine the DMSG shown in Figure 6 by removing the unsound edge between *AORB 3* and *AORB 2*; the result is shown in Figure 8. The property that each mutant in the graph can be killed has been shown by providing a test. The property that killing a given mutant implies killing a subsumed mutant has been analyzed statically. Therefore, assuming that our (hand) analysis is mistake-free, Figure 8 refines part of the DMSG through static analysis, making it closer to the "true" MSG.

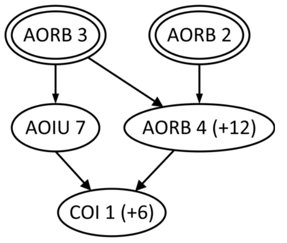

Fig. 8. Partial subsumption graph refined by static analysis.

We conclude from this exercise that dynamic subsumption based on test execution tends to be optimistic about establishing subsumption relationships between mutants, because any finite test set may exclude a range of test values that would disprove a subsumption relationship. On the other hand, static subsumption based on analysis may tend to be pessimistic in establishing subsumption relationships, as it may be difficult to statically determine when one mutation is killed in only a subset of cases of another.

## V. Related Work

The subsumption relation has been studied in a variety of contexts for many years. Chusho observed that measuring branch coverage over all branches in a program led to an overestimation of quality, and defined the notion of *essential branches* as a way of removing redundant branches from coverage measures [5]. In this paper, dynamically subsumed mutants play exactly the same role as non-essential branches do in the Chusho analysis. The difference is that this paper is "black-box," whereas the Chusho paper considers the actual structure of the code. Hence, the Chusho results hold for all test sets; our results are specific to a particular test set $T$.

Jia and Harman defined the notion of subsuming Higher Order Mutants (HOMs) [18]. The idea was that a single HOM could stand in for several mutants. Langdon et al. applied subsuming HOMs to relational operators [27]. Lau and Yu presented a fault hierarchy of faults in Disjunctive Normal Form (DNF) predicates [28]. Kaminski et al. [21] extended this work by defining special HOMs, which, though relatively few in number, still subsumed all of the Lau and Yu hierarchy. In terms of the relationship to this work, subsuming HOMs are defined by internal analysis of the artifact under consideration; in contrast, we observe dynamic subsumption with respect to a specific test set.

Yao et al. [40] studied the characteristics of "*stubborn*" mutants that are especially difficult to kill; defined as mutants that are not killed by a branch-adequate test set, but can be killed by hand-crafted tests. They further examined the relationship between different mutation operators and the frequency of equivalent and stubborn mutants generated by those operators. The relationship between stubborn and minimal mutants has not yet been examined; both would appear to be useful.

Kaminski et al. [22] observed that four of the seven mutants generated by Mothra's Relational Operator Replacement (ROR) were always subsumed by other mutants. The special treatment here was that the subsumed ROR operators depended on which operator appeared in the original code. Kapfhammer raised exactly the point that raw mutation scores led to overly optimistic evaluations of quality and defined subsuming mutants in the context of the Conditional Operator Replacement (COR) operator [19]. Again, in terms of the relationship to this work, eliminating mutants in these papers is done at the operator level before test cases are generated. Our approach to subsumption is based on the artifact's behavior after a specific test set is chosen.

Papadakis and Malevris [33] created a graph model of the mutants and used static symbolic execution to generate tests. Although their mutant graph does not model subsumption as ours does, the subsumption model could be incorporated in a way to make their test generation model more efficient.

Given that test set minimization is NP-complete, various researchers have developed test set minimization heuristics; Harrold et al. give an authoritative treatment [16]. Studies have investigated whether minimizing test sets with respect to various coverage criteria has an effect on fault detection of the remaining tests. A positive result [39] reported on a case study in which minimizing test sets with respect to the dataflow "all-uses" coverage did not significantly reduce fault detection ability. A subsequent study [35] on the Siemens suite came to a contradictory conclusion: minimizing test sets with respect to edge (or branch) coverage severely compromised fault detection. The relevance of test set minimization to mutant minimization is that minimal mutant sets are defined in terms of minimal test sets; hence fault-detection bias introduced by minimal test sets potentially affects minimal mutant sets as well. Further research is needed to evaluate this issue.

Automated Program Repair (APR) is a relatively young area, but interest is growing rapidly. Some approaches assume the prior existence of formally analyzable specifications [3], [4], [25], [36] or other programmer-defined additional information such as "hotspots" [23]. Other approaches attempt to extract implied contracts from executions [6], [34] or pattern repairs based on human-generated patches [24]. The SemFix approach [30] applies three techniques, fault isolation, statement-level specification inference, and program synthesis, to create patches tailored exactly to the fault under repair. The GenProg approach [10], [11], [12], [14], [15], [37], [38], uses genetic programming to address faults in legacy code. Patches in GenProg are usually fragments of other code taken from the same system, and fitness is determined by how many test cases pass. Given that APR often takes place in the context of a specific set of *repair tests*, the DMSG defined in this paper could potentially be used to organize and search patches.

In our other paper [1], we introduced dynamic mutation subsumption and defined the score function as a binary matrix. This paper extends that work by defining three different types of subsumption graphs, examining how dynamic subsumption changes as tests are added, and comparing this process across two different mutation systems.

## VI. Conclusions

This paper presents the subsumption graph, a visualization technique to support the analysis of the relationships between mutants. An example of subsumption graph growth is demonstrated, and a mutant state machine is described that provides a model for mutant behavior as tests are added.

We constructed a DMSG for a Java example. Adding dramatically more tests had little effect on the subsumption graph, but did increase the number of minimal test sets. Generating a companion SMSG through static analysis appears to be viable, and, with proper analysis techniques, may require less effort than the dynamic approach.

## VII. Future Work

In this paper, we observed a substantial difference in dynamically minimal mutants and minimal test sets between the Java and C implementations of the same program. Additional investigation is required to develop a deeper understanding of how the number of generated mutants impacts the dynamically minimal mutants and test sets.

Generation of a subsumption graph through static analysis might be a useful way to more quickly determine the minimal mutants and minimal test sets necessary for testing a program, without the effort of first developing a more complete test set to allow determination of the dynamic subsumption graph. Further investigation is needed to determine how static analysis tools might be used or modified to produce subsumption relationships between mutants. Once such graphs are developed, investigation of how to "merge" static and dynamic subsumption graphs might lead to closer approximation of the "true" subsumption graph for a program's mutant set.

If we can learn more about mutant subsumption graphs and create approximate minimal mutant sets, we could significantly increase the efficiency of mutation testing systems. muJava generated 173 mutants for `cal()`, but the minimal set only has seven. That is a potential savings of 2400%! If we could even get close to minimal mutant sets, this advance has the potential to dramatically change the return on investment calculation of mutation testing, making it available for practical use by professional software engineers.

## Acknowledgments

Bob Kurtz' research is supported by Raytheon. Prof. Marcio Delamaro's research is supported by FAPESP (Fundação de Amparo a Pesquisa do Estado de São Paulo), process number 2012/16950-5.

## References

[1] Paul Ammann, Marcio E. Delamaro, and Jeff Offutt. Establishing theoretical minimal sets of mutants. In *7th IEEE International Conference on Software Testing, Verification and Validation (ICST 2014)*, Cleveland, Ohio, March 2014. To appear.

[2] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, Cambridge, UK, 2008. ISBN 978-0-521-88038-1.

[3] A. Arcuri. On the automation of fixing software bugs. In *Proceedings of the 30th International Conference on Software Engineering, Doctoral Symposium*, pages 1003–1006, Leipzig Germany, May 2008.

[4] A. Arcuri and X. Yao. A novel co-evolutionary approach to automatic software bug fixing. In *Evolutionary Computation (IEEE World Congress on Computational Intelligence)*, pages 162–168, June 2008.

[5] T. Chusho. Test data selection and quality estimation based on the concept of essential branches for path testing. *IEEE Transactions on Software Engineering*, 13(5), May 1987.

[6] V. Dallmeier, A. Zeller, and B. Meyer. Generating fixes from object behavior anomalies. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE 2009)*, pages 550–554, Auckland, New Zealand, 2009.

[7] Márcio E. Delamaro and José C. Maldonado. Proteum-A tool for the assessment of test adequacy for C programs. In *Proceedings of the Conference on Performability in Computing Systems (PCS 96)*, pages 79–95, New Brunswick, NJ, July 1996.

[8] Richard A. DeMillo, Richard J. Lipton, and Fred G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, April 1978.

[9] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, October 2005.

[10] E. Fast, C. Le Goues, S. Forrest, and W. Weimer. Designing better fitness functions for automated program repair. In *12th Annual conference on Genetic and Evolutionary Computation (GECCO 2010)*, pages 965–972, Portland, Oregon, 2010.

[11] S. Forrest, T. Nguyen, W. Weimer, and C. Le Goues. A genetic programming approach to automated software repair. In *11th Annual conference on Genetic and Evolutionary Computation (GECCO 2009)*, pages 947–954, 2009.

[12] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: fixing 55 out of 105 bugs for $8 each. In *Proceedings of the 34th International Conference on Software Engineering (ICSE 2012)*, pages 3–13, Zurich, Switzerland, 2012.

[13] C. Le Goues, S. Forrest, and W. Weimer. Current challenges in automatic software repair. *Software Quality Journal*, 21:421–443, 2013.

[14] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, Jan-Feb 2012.

[15] C. Le Goues, W. Weimer, and S. Forrest. Representations and operators for improving evolutionary software repair. In *14th Annual conference on Genetic and Evolutionary Computation (GECCO 2012)*, pages 959–966, Philadelphia PA, USA, 2012.

[16] Mary Jean Harrold, R. Gupta, and Mary Lou Soffa. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology*, 2(3):270–285, July 1993.

[17] Marlie Hutchins, H. Foster, Thomas Goradia, and Thomas Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the Sixteenth International Conference on Software Engineering*, pages 191–200, Sorrento, Italy, May 1994. IEEE Computer Society Press.

[18] Yue Jia and Mark Harman. Constructing subtle faults using higher order mutation testing. In *2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 249–258, Beijing, September 2008.

[19] René Just, Gregory M. Kapfhammer, and Franz Schweiggert. Do redundant mutants affect the effectiveness and efficiency of mutation analysis? In *Eighth Workshop on Mutation Analysis (IEEE Mutation 2012)*, Montreal, Canada, April 2012.

[20] Raghu Kacker and Rick Kuhn. Automated combinatorial testing for software-beyond pairwise testing. Online, 2008. http://csrc.nist.gov/groups/SNS/acts/, last access June 2009.

[21] Garrett Kaminski and Paul Ammann. Using a fault hierarchy to improve the efficiency of DNF logic mutation testing. In *2nd IEEE International Conference on Software Testing, Verification and Validation (ICST 2009)*, pages 386–395, Denver, CO, April 2009.

[22] Garrett Kaminski, Paul Ammann, and Jeff Offutt. Improving logic-based testing. *Journal of Systems and Software, Elsevier*, 86:2002–2012, August 2013.

[23] C. Kern and J. Esparza. Automatic error correction of Java programs. In *Proceedings of the 15th International Conference on Formal Methods for Industrial Critical Systems (FMICS 2010)*, pages 67–81, Antwerp, Belgium, 2010.

[24] D. Kim, J. Song J. Nam, and S. Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 35th International Conference on Software Engineering (ICSE 2013)*, San Francisco, CA, May 2013. IEEE Computer Society Press.

[25] R. Konighofer and R. Bloem. Automated error localization and correction for imperative programs. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 91–100, November 2011.

[26] D. Richard Kuhn. Fault classes and error detection capability of specification-based testing. *ACM Transactions on Software Engineering Methodology*, 8(4):411–424, October 1999.

[27] William B. Langdon, Mark Harman, and Yue Jia. Efficient multi objective higher order mutation testing with genetic programming. *Journal of Systems and Software, Elsevier*, 83(12):2416–2430, 2010.

[28] M. F. Lau and Y. T. Yu. An extended fault class hierarchy for specification-based testing. *ACM Transactions on Software Engineering Methodology*, 14(3):247–276, July 2005.

[29] Yu-Seung Ma, Jeff Offutt, and Yong-Rae Kwon. MuJava : An automated class mutation system. *Software Testing, Verification, and Reliability, Wiley*, 15(2):97–133, June 2005.

[30] H. D. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: Program repair via semantic analysis. In *Proceedings of the 35th International Conference on Software Engineering (ICSE 2013)*, San Francisco, CA, May 2013. IEEE Computer Society Press.

[31] Jeff Offutt, Ammei Lee, Gregg Rothermel, Roland Untch, and Christian Zapf. An experimental determination of sufficient mutation operators. *ACM Transactions on Software Engineering Methodology*, 5(2):99–118, April 1996.

[32] Jeff Offutt, Yu-Seung Ma, and Yong-Rae Kwon. muJava home page. Online, 2005. http://cs.gmu.edu/∼offutt/mujava/, last access January 2014.

[33] Mike Papadakis and Nicos Malevris. Mutation based test case generation via a path selection strategy. *Journal of Information and Software Technology*, 54:915–932, September 2012.

[34] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, M. Carbin J. Bachrach, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard. Automatically patching errors in deployed software. In *ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP 09)*, pages 87–102, 2009.

[35] Gregg Rothermel, Mary Jean Harrold, Jeffery Ostrin, and Christie Hong. An empirical study of the effects of test set minimization on fault detection capabilities of test suites. In *14th IEEE International Conference on Software Maintenance (ICSM 1998)*, pages 34–43, Bethesda, Maryland, 1998. IEEE.

[36] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. In *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA '10)*, pages 61–72, Trento, Italy, 2010. ACM.

[37] W. Weimer, S. Forrest, C. Le Goues, and T. Nguyen. Automatic program repair with evolutionary computation. *Communications of the ACM*, 53(5):109–116, May 2010.

[38] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering (ICSE 2009)*, pages 364–374, Vancouver Canada, 2009.

[39] W. Eric Wong, Joseph R. Horgan, Saul London, and Aditya P. Mathur. Effect of test set minimization on fault detection effectiveness. In *Proceedings of the 17th International Conference on Software Engineering (ICSE 1995)*, pages 41–50, Seattle, Washington, 1995. ACM.

[40] Xiangjuan Yao, Mark Harman, and Yue Jia. A study of equivalent and stubborn mutation operators using human analysis of equivalence. In *Proceedings of the 36th International Conference on Software Engineering, (ICSE 2005)*, Hyderabad, India, May 2014. IEEE Computer Society Press.