

# An Empirical Evaluation of Weak Mutation

*A. Jefferson Offutt*

Department of Information and Software Systems Engineering  
George Mason University  
Fairfax, VA 22030

*Stephen D. Lee*

IBM Corporation A00/062  
P.O. Box 12195  
Research Triangle Park, NC 27709

February 24, 1996

## **Abstract**

Mutation testing is a fault-based technique for unit level software testing. Weak mutation was proposed as a way to reduce the expense of mutation testing. Unfortunately, weak mutation is also expected to provide a weaker test of the software than mutation testing does. This paper presents results from an implementation of weak mutation, which we used to evaluate the effectiveness versus the efficiency of weak mutation. Additionally, we examined several options in an attempt to find the most appropriate way to implement weak mutation. Our results indicate that weak mutation can be applied in a manner that is almost as effective as mutation testing, and with significant computational savings.

*Index Terms:* fault-based testing, firm mutation, mutation testing, software testing, weak mutation

# 1 INTRODUCTION

Mutation testing [6] is a powerful technique for unit level software testing that, as usually implemented [4], is computationally expensive. Weak mutation testing was proposed by Howden [13] as a refinement and extension of his earlier work on algebraic testing [12] and Foster’s work [8]. He suggested a modification to mutation testing that is computationally more efficient, but provides a less stringent test. The effectiveness of testing with weak mutation as compared to strong mutation has been open to question, and we present experimental work to try to answer this question. Our weak mutation system is based on the Mothra mutation system [4, 7], described in section 2. We describe our implementation by presenting the major differences between Mothra and our system, the details of which are found in Lee’s thesis [18].

Since mutation is primarily a unit testing technique, we performed this experiment on program units (subroutines and functions). We feel that unit testing is the most productive phase to find faults in software and where the bulk of our automated efforts should be focused. Testing large, monolithic software systems is less effective and may be impractical for detailed, white-box, techniques. Additionally, even if the questions we pose for weak mutation are applicable to integration testing, it is important that they first be answered for unit testing.

In this introduction, we summarize strong mutation testing, describe weak mutation testing, and then review some related work.

## 1.1 Strong Mutation Testing

Mutation testing is a fault-based technique for unit testing of software that has been widely studied in recent years [4, 6, 10, 15, 25]. Fault-based testing strategies are based on the notion of testing for specific kinds of faults. Mutation testing describes faults as simple syntactic changes to a program, called *mutations*. These mutations are used to create mutant versions of a test program, and test data are created to cause the mutants to fail. When a mutant fails, it is considered to be *killed* by the test case. The quality of a set of test cases is measured by the percentage of mutants *killed* by the test data. Some mutants are functionally equivalent to the original program and cannot be killed. A *mutation score* is the percentage of non-equivalent mutants that a test set has killed. A test set is *mutation-adequate* if its mutation score is 100%.

Although the number of possible faults for a given program can be large, mutation testing uses two principles to restrict the classes of mutations that are created: the *competent programmer hypothesis* [2] and the *coupling effect* [6]. The competent programmer hypothesis states that competent programmers tend to write programs that are “close” to being correct. Although a program written by a competent programmer may be incorrect, it will only differ from a correct version by relatively few faults. The coupling effect states that a test data set that distinguishes

all programs with simple faults is sensitive enough so that it will also distinguish programs with more complex faults [6]. Neither the competent programmer hypothesis nor the coupling effect are always true, but are heuristic guidelines used by mutation testing. The coupling effect has been supported analytically by Morell [22] and experimentally by Offutt [25].

Although the competent programmer hypothesis and coupling effect restrict the number of mutants generated by mutation systems, the number is still large. Acree [1] derives a formula giving the number of mutations for a program that is bounded by  $N^2$ , where  $N$  is the number of variable references in the program. Mothra, for example, creates 951 mutants for the commonly studied 30 line triangle classification program TRITYP. Although current mutation systems automate most of the mutation process [4], executing this many mutants for such a small program is a significant computational expense. Howden [13] suggested *weak mutation* as a way to reduce this computation.

## 1.2 Weak Mutation Testing

In Howden’s terminology,  $P$  is a program,  $C$  is a *simple component* of  $P$ ,  $C'$  is a mutated version of  $C$ , and  $P'$  is the mutated version of  $P$  containing  $C'$ . Weak mutation testing requires that a test case  $t$  causes  $C'$  to compute a different (incorrect) value than  $C$  does on at least one execution of  $C'$ . Even though  $C'$  may produce a different outcome,  $P'$  may still produce the same results as  $P$ . Since strong mutation requires programs  $P$  and  $P'$  to produce different output, weak mutation requires a less stringent, or *weaker*, test set than strong mutation.

Howden’s original paper does not provide a precise definition of program component, but describes components as “normally corresponding to elementary computational structures in a program”. He gives five types:

1. variable reference,
2. variable assignment,
3. arithmetic expression,
4. relational expression, and
5. boolean expression.

Although the idea is further refined in Howden’s book on functional testing [14], components have not previously been defined precisely enough for an implementation such as ours. It seems that just as the definition of strong mutation depends on the specific mutation operators implemented, which in turn depend on the language being tested, the definition of weak mutation depends on the definition of program component, which also depends somewhat on the language.

The advantage of weak over strong mutation is clear; weak mutation requires significantly less program execution. Its disadvantage is equally clear; weak mutation adequate test sets are less

effective than strong mutation adequate test sets (hence the term “weak”). There is little empirical evidence about how much less effective weak mutation is than strong mutation, or how much execution time is saved. Our goal in this research was to compare weak and strong mutation by directly measuring the relative strengths of the same sets of test data under both strong and weak mutation. We refer to the weak mutation score as *WMS* and the strong mutation as *SMS*.

Our weak mutation system is called *Leonardo* (*Looking at Expected Output Not After Return but During Operation*). Leonardo was built by modifying Mothra to compare the program states after mutated program components rather than the final output; in other respects Leonardo functions exactly like Mothra. We first discuss some of the work that has been done in the weak mutation area since Howden’s initial paper, then give some of the technical details of Leonardo, including several choices for the definition of a component. Next we discuss experimental results from Leonardo and present conclusions concerning the value of weak mutation.

### 1.3 Previous Work in Weak Mutation

Although weak mutation has been mentioned by several researchers [9, 10, 11, 19, 31], there has been little effort to evaluate its effectiveness. Hamlet [10] presented an early testing system that was embedded in a compiler and performed a version of instrumented weak mutation. Although the method differed significantly from later mutation systems, Hamlet’s system seems to be the first mutation-like testing system.

Girgis and Woodward [9] implemented a system for Fortran-77 programs that integrates weak mutation and data flow analysis. Their system instruments a source program to collect program execution histories. These execution histories are then evaluated to measure the completeness of test data with respect to weak mutation and several data flow path selection criteria. Their weak mutation system is *analytical* in nature, rather than *execution-based*, as Mothra is. Analytical mutation systems do not actually execute mutants, but try to analytically decide whether to kill each mutant. The system examines the execution history, and if the test case would have caused a mutant to produce an internal program state that differed from the original program’s internal state, the mutant is killed. Although this analytical approach is computationally less expensive than an execution-based approach, and was what Howden originally intended, it suffers from two problems. First, whether a mutant can be killed can only be obtained for a few kinds of mutants. Second, since no separate executions are being done for the mutants, the components must have a very localized extent, precluding several of the components that we have implemented (including the one that we found to be the most effective).

Girgis and Woodward’s system also only considers four of Howden’s five elementary program components (variable assignment, variable reference, arithmetic and relational expression), and only applies three types of mutations (wrong-variable, off-by-a-constant, and wrong-relational-operator). These transformations seem to correspond to Mothra’s scalar variable replacement

(`svr`), unary operator insertion (`uoi`), and relational operator replacement (`ror`) operators. Mothra includes 19 other mutant operators, which are listed elsewhere [4], and fully described by King and Offutt [15]. Since Leonardo is based on Mothra, it uses all 22 mutant operators. These additional operators allow for considerably more fault detection power than systems that use a small subset of them. Additionally, Leonardo uses several different definitions of “component”, including several that extend beyond the mutated statement and thus cannot be implemented analytically. These components are described in section 2.1.

Woodward and Halewood [31] introduced the idea of *firm mutation* by pointing out that weak and strong mutation represent extreme ends of what is actually a spectrum of mutation approaches. In mutation testing, we kill mutants by comparing the state of the mutant program with the state of the original program on the same test case. Weak and strong mutation differ principally in when they compare the states; strong mutation compares the final outputs of the programs and weak mutation compares the intermediate states after execution of the component. Woodward and Halewood point out that we can compare the states of the two programs at any point between the first execution of the mutated statement and the end of the program, yielding what they called *firm mutation*. Their paper states that a firm system was currently under development, so no results were available.

Firm mutation is similar to Morell’s concept of “extent” in fault-based testing [22, 23]. A *local extent* technique demonstrates that a fault has a local effect on the computation, and a *global extent* demonstrates that a fault will cause a program failure. Weak mutation is a local extent technique and strong mutation is a global extent technique. Morell also points out that we could require that the fault affect the program’s execution at any point between the local and global extents, depending on how far we require the incorrect program state to propagate. Richardson and Thompson [29] have used a path analysis approach to extend these ideas to require that a fault transfer from its origination point (corresponding to the location of the mutation) to some point later in the program’s execution. We have implemented weak mutation using several different components, which means Leonardo can be considered a firm mutation system.

Marick has also implemented a weak mutation system [19, 20] and reported results from using test data generated for weak and strong mutation to find faults that were injected into programs. The early results [19] predated his implementation, GCT [20], and his procedure involved considerable hand analysis (to inject faults, analyze the faults, etc). His results support the hypothesis that weak mutation has nearly the same effectiveness as strong mutation.

An analytical study of weak mutation by Horgan and Mathur [11] has shown that under certain conditions, test sets generated to satisfy weak mutation can also be expected to also satisfy strong mutation. For their proof, they assumed that programs are unary functions over  $Z$  modulo  $n$  (this assumption is made for convenience, but is not necessary to the proof), and assumed that a state of a program  $p \in P$  is encoded as a single integer. Their probabilistic analysis showed that weakly adequate test sets are also strongly adequate with very high probability. Constraint-based

testing [7, 24] provides indirect support for weak mutation. Test cases that are generated to cause a mutant to have an incorrect intermediate state (essentially satisfying weak mutation) kill their target mutants a large percentage of the time.

## 2 A WEAK MUTATION SYSTEM

Mothra is an execution-based mutation testing system to test unit-level software. Mothra parses source programs into a symbol table and an intermediate postfix language consisting of instructions in Mothra Intermediate Code (MIC). For each mutant of a program, Mothra creates a *mutant descriptor record* (MDR) that describes the changes to the MIC instructions necessary to produce that mutant. Mothra uses 22 mutation operators, representing more than 10 years of refinement through several mutation systems. These operators explicitly require the test data to meet statement and branch coverage, extremal values, and domain perturbation criteria. In addition, the operators directly model many types of faults. Test cases may be created interactively, through a query process or through a spreadsheet tool, or may be generated automatically by *Godzilla* [7]. Mothra executes the original program and each mutant by interpreting the MIC instructions. For each test case, the output of the original program is saved and compared against the output of each mutant to determine if the test case killed the mutant.

Leonardo was built by modifying Mothra’s interpreter so that it compares original and mutant program states at intermediate points in the program execution, rather than after program termination. By basing this weak mutation system on Mothra, we are able to incorporate all 22 mutation operators that Mothra uses, as well as use the other Mothra tools (i.e., for test data generation and creating mutants).

One other point bears mentioning about our weak mutation system. Howden’s paper suggests that it is unnecessary to re-execute the program for each mutant, but that all the mutations of a component could be tested with a single test. This could be done in an execution-based system by executing the program to the beginning of the component, saving the state of the program, and separately executing each mutated version of the component. An analytical system can execute the program to the beginning of the component, and decide if each mutant would cause an incorrect state. The execution-based approach was described by King and Offutt [15] as *split-stream execution* but has not been implemented. Both schemes avoid re-executing the test program up to the mutated component by saving the state of the program before the component. Since this can also be implemented in a strong mutation system, the idea of saving states before mutations can be considered as separate from weak mutation, so Leonardo does **not** use this.

## 2.1 Definition of A Component

Our view of a component corresponds to that of Woodward and Halewood’s [31]; we define a component as the location where the states of the original and mutant program are compared. In our experiments, we varied the component definition to determine not only whether weak mutation is viable but also to find the best comparison point. We chose four points for comparison, and used these to define four *variants* of weak mutation. A fifth variant is **STRONG** mutation, the standard approach of executing the programs to completion and comparing final output.

**EX-WEAK/1** (EXpression-WEAK/1) mutation compares the states after the **first** execution of the innermost expression that surrounds the mutant. We chose the innermost expression because the outermost expression would be equivalent to our next variant, **ST-WEAK/1**. As defined, **EX-WEAK/1** is equivalent to Girgis and Woodward’s comparison point. For some mutation operators, this definition had to be interpreted liberally. For example, the statement analysis, statement deletion, and return statement replacement operators affect the entire statement, not just an expression.

**ST-WEAK/1** (SStatement-WEAK/1) mutation compares the states after the **first** execution of the mutated statement. Thus, to kill a mutant that changes an assignment statement, the left-hand side variable must be assigned a different value, and to kill a mutant that changes a control flow statement, the program counter must be different. The handling of the statement analysis, statement deletion, and return statement replacement operators is identical under **EX-WEAK/1** and **ST-WEAK/1** mutation.

A basic block is a maximal sequence of instructions with a single entry and single exit. **BB-WEAK/1** (Basic-Block-WEAK/1 execution) mutation requires the states to be compared at the end of the **first** execution of the basic block that contains the mutated statement. For this variant, we found that there were many mutations within loops that could not be killed on the first iteration. This led us to extend the **BB-WEAK/1** variant to allow multiple executions of the mutated statement. **BB-WEAK/N** (Basic-Block-WEAK/N execution) mutation compares the states after *each* execution of the basic block. Execution is halted and the mutant is killed when either an incorrect state is found or when the basic block is executed more times than it was executed in the original program.

Figure 1 illustrates the comparison points of the four variants. The mutation shown is an arithmetic operator replacement that has substituted a plus for a multiplication. Under **EX-WEAK/1**, we compare the values of the expression  $(Z * Z)$  and  $(Z + Z)$ , whereas **ST-WEAK/1** compares the values of  $X$  after the statement is executed. Under **BB-WEAK/1**, we compare the program state at the end of the basic block indicated; under **BB-WEAK/N**, we compare at that point *every* time it is reached.

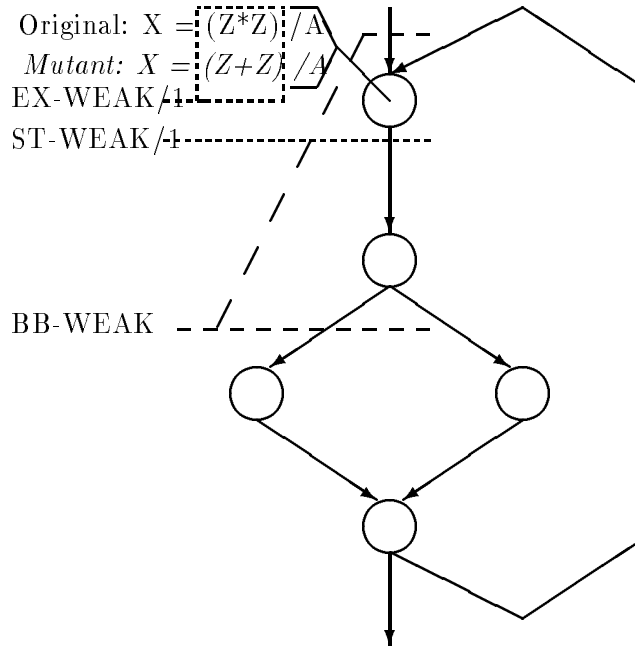


Figure 1: State Comparison Points

## 2.2 Portion of the State Compared

The third characteristic for weak mutation, other than program component and number of component executions, is the portion of the program state that is compared. Strong mutation compares only the output of a program; weak mutation compares intermediate states. The intermediate state used by Leonardo includes the entire variable space, the program counter, and the top of the expression stack for the postfix language interpretation. It is not always necessary to consider the entire state when comparing component outcomes; the portion of the state that may be affected depends on the mutation and the variant of weak mutation. For **EX-WEAK/1**, most mutations affect only the value of an expression; for these, we compare only the top item of the expression stack. For **ST-WEAK/1**, the left-hand side of assignment statements are compared, or the program counter if the mutation is in a decision statement. For **BB-WEAK/1** and **BB-WEAK/N**, we compare the entire variable space and program counter.

## 2.3 Equivalent Mutants Under Weak Mutation

The most time-consuming part of our experiments was determining the equivalent mutants for the programs. Unfortunately, the set of equivalent mutants of a program is different for each mutation variant. Although in some cases the equivalent mutants for one variant form a subset of the equivalent mutants for other variants, this is not always the case.

For example, if a mutant can be killed by some test case under **STRONG**, it can compute an

incorrect state at the end of at least one execution of a basic block, so it can also be killed under **BB-WEAK/N**. If a mutant cannot be killed, then it is equivalent, so the contrapositive is if a mutant is equivalent under **BB-WEAK/N**, it is also equivalent under **STRONG**. This means that the set of **BB-WEAK/N** equivalent mutants is a subset of the **STRONG** equivalent mutants. Conversely, a mutant's incorrect intermediate state may converge to a correct final state after the execution of the mutated basic block, so there can be mutants that are **STRONG** equivalent that are not **BB-WEAK/N** equivalent. These statements can be formalized by calling a mutant that can be killed by some test case *vulnerable*, and expressing the fact that a mutant  $m$  is vulnerable under variant  $n$  as  $V(m, n)$ . So,  $V(m, Strong) \implies V(m, BB - WEAK/N)$ , and the contrapositive is  $\neg V(m, BB - WEAK/N) \implies \neg V(m, Strong)$ . This is equivalent to  $E(m, BB - WEAK/N) \implies E(m, Strong)$ .

Since some mutants cannot cause an incorrect state on their first iteration in a loop, there are mutants that are equivalent under **BB-WEAK/1** but not **STRONG**. Similarly, there are mutants that are equivalent under **ST-WEAK/1** and **EX-WEAK/1** that are not equivalent under **STRONG**. Thus, the set of equivalent mutants for **BB-WEAK/1**, **ST-WEAK/1**, and **EX-WEAK/1** are different from **BB-WEAK/N** and **STRONG**.

## 2.4 Implementation Notes

To perform weak mutation, the intermediate states of the original program must be saved for each execution of each program component. Since the state contains the entire variable space and additional components, the amount of saved information may be relatively large. To limit the memory space required, Leonardo saves the original state for only one program component at a time. Thus, to execute a mutant, Leonardo first determines the program component being mutated, and executes the original program. As the original program is executed, the state of the program is saved after each execution of the mutated component. Finally Leonardo executes the mutant program, and compares the appropriate program states.

If the entire original program must be executed to record the intermediate states every time a mutant is executed most of the speed advantage of weak mutation will be lost. To reduce this loss, Leonardo executes mutants in order of their comparison points (i.e., the end of the mutated program component). The original program is first executed to a given comparison point, and then all mutants with that comparison point are executed. After these mutants are executed, Leonardo executes the original program to the next comparison point. By doing this, the original program only needs to be executed once for a group of mutants instead of once for each mutant. The number of mutants sharing a comparison point varies as the level of weak mutation changes. With **EX-WEAK**, there may be several comparison points for a single statement and only a few mutants may share a comparison point. With **BB-WEAK** mutation, there may be many mutants that share a comparison point and the execution time for the original becomes negligible. On the other hand, the amount of state to be compared is significantly less for **EX-WEAK** than **BB-WEAK**.

### 3 EXPERIMENTATION WITH WEAK MUTATION

If we execute the same test data on a weak and strong mutation system, *WMS* will usually be at least as high as *SMS* (*WMS* would *always* be at least as high as *SMS* if there were no equivalent mutants). This higher weak mutation score does not mean the test data is better, but that the measurement is weaker; thus using weak mutation to create test data that scores 100% may not test the software as effectively as strong mutation. On the other hand, weak mutation is less costly than strong mutation. This is a clear tradeoff of effectiveness versus efficiency and this section describes two experiments that evaluate this tradeoff. In the first experiment, we compared each of the weak mutation variants described above against strong mutation. We generated 10 separate 100% adequate test data sets for each weak mutation variant, and computed the strong mutation score for those test data sets. Using the Mothra system as an experimental platform for these experiments makes the mutation scores directly comparable. In the second experiment, we generated sets of test data that were less than mutation-adequate. The same test data sets were measured for adequacy against all four weak mutation variants, as well as strong mutation. The mutant execution times were also compared in terms of the number of source program statements executed. For both experiments, test data sets were generated automatically using Godzilla [7] and, when necessary, augmented by hand. The programs were run on a Sun SPARCstation SLC running SunOS version 4.1.

#### 3.1 Experimental Programs

These experiments were performed with eleven subroutines. This study involved a significant amount of hand-analysis (e.g., determining equivalent mutants), which would be difficult for larger, integrated subsystems. These routines were chosen to cover a fairly broad class of applications, and in addition, several have been studied elsewhere, which makes this study comparable with earlier studies. Table 1 lists the programs along with a brief description, the number of executable Fortran lines, and the number of mutants generated. The source for these programs are given in Lee's thesis [18]. Four of the 11 programs do not contain loops (MID, QUAD, TRITYP, and TRISMALL); for these BB-WEAK/1 is equivalent to BB-WEAK/N. Table 2 gives the number of equivalent mutants for each program and each mutation variant.

#### 3.2 Experiment 1

In our first experiment, we generated test sets that were 100% mutation-adequate for each variant, and then computed the mutation scores on strong mutation. To eliminate any bias introduced by a particular test case set, we generated 10 separate test sets for each program for each variant. Godzilla generates many redundant test sets, so we automatically eliminated all test cases that did not (weakly) kill at least one mutant. For example, Godzilla generates over 400 test cases for

Program	Description	Statements	Mutants
BUB	Bubble sort on an integer array.	11	338
CAL	Finds the number of days between any two given dates of the same year.	29	3009
EUCLID	Euclid’s greatest common divisor algorithm.	11	195
FIND	Partitions an array so that every element to the left of a key position is less than or equal to the key and every element to the right is greater than or equal to the key.	28	1022
INSERT	Sorts an array of integers using insertion sort.	14	460
MID	Returns the middle value of three integers.	16	183
PAT	Returns the starting position of a pattern if it is in the subject.	17	513
QUAD	Finds the root(s) of a quadratic equation.	10	359
TRISMALL	Classifies a triangle as equilateral, isosceles, scalene, or illegal.	13	544
TRITYP	Larger version of TRISMALL	28	951
WARSHALL	Calculates the transitive closure of a Boolean matrix.	11	305

Table 1: Experimental Program Descriptions

TRITYP; after reduction, there were an average of 23.5 test cases for **EX-WEAK/1**, 43 for **ST-WEAK/1** and 43.1 for **BB-WEAK/1**.

The strong mutation scores for the four weak mutation test sets are shown in Figure 2. Each point represents the average strong mutation score of the 10 sets of test data generated for that routine and that weak mutation variant. The standard deviation of the mutation scores was less than two in all cases [18]. Since MID, QUAD, TRISMALL, and TRITYP have no loops, **BB-WEAK/N** is equivalent to **BB-WEAK/1** and we show no scores for **BB-WEAK/N**. Since the purpose of Figure 2 is to compare the relative differences of the program’s scores for each weak mutation variant, we scaled the scores for PAT by adding .875 of the difference between the actual scores and 100 so that all programs could be shown together. PAT’s actual scores were 64%, 85.5%, 86%, and 92.5%.

The most interesting aspect of this graph is that the strong mutation score for the **BB-WEAK/N** test data sets were lower than those for **ST-WEAK/1** and **BB-WEAK/1**. Our intuition was that since

Program	Mutants	EX-WEAK/1	ST-WEAK/1	BB-WEAK/1	BB-WEAK/N	STRONG
BUB	338	32	42	42	28	34
CAL	3009	111	117	117	117	188
EUCLID	195	25	29	29	21	24
FIND	1022	79	106	106	58	75
INSERT	460	47	65	65	41	35
MID	183	3	3	3	3	13
PAT	513	71	106	106	38	47
QUAD	359	18	20	20	20	31
TRISMALL	544	51	53	53	53	97
TRITYP	951	93	101	101	101	109
WARSHALL	305	43	49	49	23	23

Table 2: Number of Equivalent Mutants

BB-WEAK/N forces the incorrect state to be closer to the end of the program’s execution (where the strong mutation score is computed), the test data generated to satisfy BB-WEAK/N would be stronger under strong mutation than those generated for ST-WEAK/1 and BB-WEAK/1. For most of our programs, just the opposite is true. This seems to be because both ST-WEAK/1 and BB-WEAK/1 mutation require the test case to cause an incorrect intermediate state the **first** time the mutated statement is executed. BB-WEAK/N only requires the test case to cause an incorrect intermediate state on **any** execution of the mutated statement. Causing an incorrect state the first time is a more stringent requirement and often requires a stronger (and usually larger) test case set.

Aside from PAT, weak mutation shows the worst performance for TRITYP, with strong mutation scores around 95%. Recall that weak mutation includes the program counter in state comparisons while strong mutation compares only the output values. When the TRITYP BB-WEAK/1 test sets were executed for strong mutation, a state difference existed at the end of execution for *all* mutants for at least one test case in each of the test sets. TRITYP contains three exit points for an “illegal” triangle. Therefore, a mutant may be weakly killed because it terminated at a different exit point even though it produced the correct output. This illustrates one of the weaknesses of weak mutation – all of the program state used in state comparison may not be relevant to the output. It also illustrates another observation – robust software is more difficult to test than fragile software.

The test case strength seems to be principally reflected in the number of test cases generated. In Figure 3, we graph the number of test cases needed to reach 100% coverage for each of our weak mutation variants. As in Figure 2, the numbers for ST-WEAK/1 and BB-WEAK/1 scores are higher than the numbers for EX-WEAK/1 and BB-WEAK/N for most of the programs. This may indicate that the dominating factor in the strength of the test set is the number of test cases. The only routine for which BB-WEAK/N was more effective is PAT, which is also the only program that required more test cases for BB-WEAK/N. PAT seems to stand out primarily in that it has a larger percentage of mutants that are equivalent under BB-WEAK/1 mutation yet killable under BB-WEAK/N mutation

Figure 2: Weak Mutation Variant Comparisons

(mutants that could not be killed **after** the first execution).

### 3.3 Experiment 2

In addition to comparing each weak mutation variant with strong mutation, we wanted to compare each variant directly with the other variants. To do this, we generated 10 separate sets of test cases for each program and ran these test sets against each weak mutation variant as well as strong mutation. These test case sets were not generated to be 100% adequate under any mutation variant; rather, we intentionally tried to keep the mutation scores below 90% for **ST-WEAK/1** to emphasize differences among the mutation variants. A histogram showing the average scores of the 10 test case sets for each program is shown in Figure 4. For each program, the mutation score for the four weak variants and for strong mutation is shown. Since **MID**, **QUAD**, **TRISMALL**, and **TRITYP** contain no loops, their **BB-WEAK/N** scores are the same as their **BB-WEAK/1** scores. Underneath each program is the average number of test cases in the 10 test case sets.

It seems natural to expect the scores to steadily decrease from left to right for each program. Again, this is true only for the routine **PAT**; the other routines with loops have a higher mutation score for **BB-WEAK/N**, indicating that **BB-WEAK/N** often provides a weaker measure of the test case set. At first examination, the programs **TRISMALL** and **MID** may look anomalous in Figure 4, because the scores for strong mutation are higher. This is not because more mutants were killed

Figure 3: Weak Mutation Test Case Count Comparisons

under strong mutation, but because there were more equivalent mutants under strong mutation. For example, assume there are 100 mutants for a program, 20 are equivalent under strong mutation, and 10 are equivalent under **BB-WEAK/N**. If a test case set killed 70 mutants under both variants, then the strong mutation score would be  $70/(100-20) = .875$ , and the **BB-WEAK/N** score would be  $70/(100-10) = .78$ .

Figure 5 charts the number of Fortran statements executed during weak interpretation as a percentage of the statements executed during strong mutation. This is a direct comparison, since we used the same intermediate code and the same interpreter for strong and weak mutation. By only measuring the statements executed, we are factoring out the overhead involved in a mutation test (setting up the test case values, applying the mutations, etc.). These factors depend on the design of the mutation system, not the technique itself. In Mothra, under strong mutation, this overhead accounts for 80 to 90% of the execution time; with certain modifications made during implementation of Leonardo, this overhead was reduced to less than 20%.

Figure 5 shows that a significant savings in execution can be achieved, even in the **BB-WEAK/N** case. It also indicates that **BB-WEAK/N** is not always significantly more expensive than **BB-WEAK/1**.

Figure 4: Weak Mutation Variant Comparisons

## 4 CONCLUSIONS AND RECOMMENDATIONS

In this paper, we presented results from an implementation of a weak mutation system, Leonardo. Leonardo is an execution-based mutation system, meaning that it executes each mutant separately. This, and the fact that Leonardo is derived from Mothra, allows us to consider all 22 mutation operators that Mothra uses. Additionally, its dynamic nature allowed us to define four variants of weak mutation, extending from the innermost expression to the enclosing basic block.

When we began this work, we expected that **BB-WEAK/N** weak mutation testing would provide the best test sets, but might be so expensive that **ST-WEAK/1** or **BB-WEAK/1** would be more cost effective. In other words, we expected some sort of cost/strength tradeoff. Surprisingly, these experiments suggest the opposite, that weak mutation is often more powerful when applied to small components such as **ST-WEAK/1** or **BB-WEAK/1** than large components such as **BB-WEAK/N**. Unfortunately, not only is this sometimes false (PAT being our exception), but we see no way to characterize which programs would be better tested using **BB-WEAK/N**. Another, not surprising, result is that the relationships between the weak and strong mutation scores differed greatly among the programs. Thus, our goal of finding a formula that will translate weak mutation scores to strong mutation scores is not realizable.

These experiments indicate that weak mutation is a viable alternative to strong mutation,

Figure 5: Weak Mutation Execution Time Comparisons

agreeing with the probabilistic results of Horgan and Mathur [11] and the experimental results of Girgis and Woodward [9] and Marick [19]. Based on these results, we recommend that weak mutation be used as a cost-effective alternative to strong mutation for unit testing of non-critical applications. We also recommend that weak mutation be applied using statement components or basic block components (**ST-WEAK/1** or **BB-WEAK/1**) rather than expression components (**EX-WEAK/1**). Note that Girgis and Woodward [9] used expression level components, and Marick [19] used expression level components. Since **ST-WEAK/1** mutation is easier to implement, is faster during comparison, and almost as effective as **BB-WEAK/1** mutation, we recommend applying weak mutation by comparing program states immediately after the first execution of the mutated statement.

For critical applications, however, strong mutation or a combination of the two is probably worthwhile. One strategy for combining weak and strong mutation is to generate a test set that provides 100% coverage for **ST-WEAK/1** weak mutation, and then identify the mutants that are equivalent under **ST-WEAK/1** and are in loops. These cannot be killed on the first iteration, but may be killable under strong mutation. Then we use strong mutation to try to kill them. Targeting these mutants can help increase the effectiveness of the test with much less cost than only using strong mutation.

Obviously, our results are based on unit testing of single subroutines, and it is not clear whether weak mutation would be as effective if applied to larger scale software. There are intuitive arguments in both directions. On the one hand, a larger program provides more opportunity for the incorrect

state of the mutant program to correct itself. Thus, any individual mutated statement has a smaller relative impact on the final state of the program. On the other hand, a larger program provides more opportunity for the mutated statement to expand its effect on the final program state. This question can probably only be answered empirically.

## 5 FUTURE WORK

There are several potential extensions of this work. The split-stream approach discussed in section 2 could significantly improve the execution savings. This approach also could be used in a strong mutation system, but has not been implemented because of the cost and complexity of saving the many states of the test program. We can measure the improvement by counting the number of program statements executed.

If there are  $V$  data items and  $R$  data references in the program, there are approximately  $V * R$  mutants [28]. Strong mutation executes each mutant to completion, which, if there are  $N$  statements, requires  $O(V * R * N)$  statements to be executed. Since weak mutation executes, on average, half the statements, its complexity is  $O((V * R * N)/2)$ . Split-stream execution would also execute half the statements on average, so its complexity is also  $O((V * R * N)/2)$ . Combining the two, however, allows us to reduce the number of statements executed per mutant to 1, giving a combined and reduced complexity of only  $O(V * R)$ .

One could also imagine many more than the four variants that Leonardo uses for state comparison. For example, an interval analysis could be done and the states could be compared after each interval, or we could compare the program states after executing strongly connected regions in the control flow graph. Although one of these may be slightly more effective than the components that Leonardo uses, we doubt that the results would differ significantly from comparing the states after basic blocks. Also, **ST-WEAK/1** could be extended to compare states after  $N$  executions of the statement (**ST-WEAK/N**), but since **ST-WEAK/1** differs only slightly from **BB-WEAK/1**, we expect that **ST-WEAK/N** would be similar to **BB-WEAK/N**.

At some level, weak mutation can be viewed as a way to improve the efficiency of a relatively expensive testing technique, mutation testing. Thus, it is probably worthwhile to put weak mutation in the broader context of other attempts to improve the efficiency. Mathur et al. [3, 17, 16, 21] have looked at ways of parallelizing mutation, with some success. We have recently implemented a version of Mothra on the hypercube that is interpretive and directly comparable to Mothra [27], as opposed to PMothra [3], which relies on separate compilation of each mutant. In addition, Krauser [5] has suggested a way to improve the performance of mutation systems by applying mutations to compiled executable code, rather than an interpreted intermediate code such as Mothra's. Untch et al. [30] have been working on a technique for using program schemas to implement mutation systems in a way that should achieve speeds that approach that of a compiler-based system, without

the expense of separately compiling each mutant. Finally, we are experimenting with a reduction technique called *selective mutation* [28, 26] that reduces the number of mutants needed by large amounts. Although these techniques have the potential of achieving orders of magnitude speedup, weak mutation could also be applied with any of these techniques to further increase the efficiency of mutation systems.

## 6 ACKNOWLEDGEMENTS

We would like to thank Dick Lipton and Rich DeMillo for suggesting that we use Mothra to evaluate weak mutation. Thanks also to Brian Marick for numerous helpful suggestions about the research and the paper.

## References

- [1] A. T. Acree. *On Mutation*. PhD thesis, Georgia Institute of Technology, Atlanta GA, 1980.
- [2] A. T. Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Mutation analysis. Technical report GIT-ICS-79/08, School of Information and Computer Science, Georgia Institute of Technology, Atlanta GA, September 1979.
- [3] B. Choi and A. P. Mathur. High performance mutation testing. *The Journal of Systems and Software*, 20(2):135–152, February 1993.
- [4] R. A. DeMillo, D. S. Guindi, K. N. King, W. M. McCracken, and A. J. Offutt. An extended overview of the Mothra software testing environment. In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, pages 142–151, Banff Alberta, July 1988. IEEE Computer Society Press.
- [5] R. A. DeMillo, E. W. Krauser, and A. P. Mathur. Compiler-integrated program mutation. In *Proceedings of the Fifteenth Annual Computer Software and Applications Conference*, Tokyo, Japan, September 1991. Kogakuin University.
- [6] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, April 1978.
- [7] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.
- [8] K. Foster. Error sensitive test case analysis. *IEEE Transactions on Software Engineering*, 6(3):258–264, May 1980.
- [9] M. R. Girgis and M. R. Woodward. An integrated system for program testing using weak mutation and data flow analysis. In *Proceedings of the Eighth International Conference on Software Engineering*, pages 313–319, London UK, August 1985. IEEE Computer Society.
- [10] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, 3(4), July 1977.

- [11] J. R. Horgan and A. P. Mathur. Weak mutation is probably strong mutation. Technical report SERC-TR-83-P, Software Engineering Research Center, Purdue University, West Lafayette IN, December 1990.
- [12] W. E. Howden. Algebraic program testing. *Acta Informatica*, 10(1):53–66, 1978.
- [13] W. E. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, 8(4):371–379, July 1982.
- [14] W. E. Howden. *Functional Programming Testing and Analysis*. McGraw-Hill Book Company, New York NY, 1987.
- [15] K. N. King and A. J. Offutt. A Fortran language system for mutation-based software testing. *Software-Practice and Experience*, 21(7):685–718, July 1991.
- [16] E. W. Krauser, A. P. Mathur, and V. Rego. High performance testing on SIMD machines. In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, pages 171–177, Banff Alberta, July 1988. IEEE Computer Society Press.
- [17] E. W. Krauser and Aditya P. Mathur. Program testing on a massively parallel transputer based system. In *Proceedings of the ISMM International Symposium on Mini and Microcomputers and their Applications*, pages 67–71, Austin TX, November 1986.
- [18] S. Lee. Weak vs. strong: An empirical comparison of mutation variants. Master’s thesis, Department of Computer Science, Clemson University, Clemson SC, 1991.
- [19] B. Marick. The weak mutation hypothesis. In *Proceedings of the Third Symposium on Software Testing, Analysis, and Verification*, pages 190–199, Victoria, British Columbia, Canada, October 1991. IEEE Computer Society Press.
- [20] B. Marick. *Using Weak Mutation with GCT*. Testing Foundations, Champaign Illinois, 1993.
- [21] Aditya P. Mathur and E. W. Krauser. Modeling mutation on a vector processor. In *Proceedings of the 10th International Conference on Software Engineering*, pages 154–161, Singapore, April 1988. IEEE Computer Society.
- [22] L. J. Morell. *A Theory of Error-Based Testing*. PhD thesis, University of Maryland, College Park MD, 1984. Technical Report TR-1395.
- [23] L. J. Morell. A theory of fault-based testing. *IEEE Transactions on Software Engineering*, 16(8):844–857, August 1990.
- [24] A. J. Offutt. *Automatic Test Data Generation*. PhD thesis, Georgia Institute of Technology, Atlanta GA, 1988. Technical report GIT-ICS 88/28, (Also released as Purdue University Software Engineering Research Center technical report SERC-TR-25-P).
- [25] A. J. Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering Methodology*, 1(1):3–18, January 1992.
- [26] A. J. Offutt, Ammei Lee, Gregg Rothermel, Roland Untch, and Christian Zapf. An experimental determination of sufficient mutation operators. *ACM Transactions on Software Engineering Methodology*, 1996. To appear.
- [27] A. J. Offutt, R. Pargas, S. V. Fichter, and P. Khambekar. Mutation testing of software using a mimd computer. In *1992 International Conference on Parallel Processing*, pages II–257–266, Chicago, Illinois, August 1992.
- [28] A. J. Offutt, Gregg Rothermel, and Christian Zapf. An experimental evaluation of selective mutation. In *Proceedings of the Fifteenth International Conference on Software Engineering*, pages 100–107, Baltimore, MD, May 1993. IEEE Computer Society Press.

- [29] D. J. Richardson and M. C. Thompson. The relay model for error detection and its application. In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, pages 223–230, Banff Alberta, July 1988. IEEE Computer Society Press.
- [30] R. Untch, A. J. Offutt, and M. J. Harrold. Mutation analysis using program schemata. In *Proceedings of the 1993 International Symposium on Software Testing, and Analysis*, pages 139–148, Cambridge MA, June 1993.
- [31] M. R. Woodward and K. Halewood. From weak to strong, dead or alive? An analysis of some mutation testing issues. In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, pages 152–158, Banff Alberta, July 1988. IEEE Computer Society Press.