Mutation 2000: Uniting the Orthogonal*

A. Jefferson Offutt
ISE Department, 4A4
George Mason University
Fairfax, VA 22030-4444 USA
703-993-1654
ofut@ise.gmu.edu
www.ise.gmu.edu/faculty/ofut/

Roland H. Untch
Department of Computer Science
Middle Tennessee State University
Murfreesboro, TN 37132-0048
615-898-5047
untch@mtsu.edu
www.mtsu.edu/~untch/

Abstract

Mutation testing is a powerful, but computationally expensive, technique for unit testing software. This expense has prevented mutation from becoming widely used in practical situations, but recent engineering advances have given us techniques and algorithms for significantly reducing the cost of mutation testing. These techniques include a new algorithmic execution technique called schema-based mutation, an approximation technique called weak mutation, a reduction technique called selective mutation, heuristics for detecting equivalent mutants, and algorithms for automatic test data generation. This paper reviews experimentation with these advances and outlines a design for a system that will approximate mutation, but in a way that will be accessible to everyday programmers. We envision a system to which a programmer can submit a program unit and get back a set of input/output pairs that are guaranteed to form an effective test of the unit by being close to mutation adequate. We believe this system could be efficient enough to be adopted by leading-edge software developers. Full automation in unit testing has the potential to dramatically change the economic balance between testing and development, by reducing the cost of testing from the major part of the total development cost to a small fraction.

1. Introduction

Mutation analysis has a rich and varied history, with major advances in concepts, theory, technology, and social viewpoints. This history begins in 1971, when Richard Lipton proposed the initial concepts of mutation in a class term paper titled "Fault Diagnosis of Computer Programs." It was not until the end of the 1970's, however, before major work was published on the subject [1, 2, 3]; the DeMillo, Lipton, and Sayward paper [3] is generally cited as the seminal reference.

PIMS [1, 4, 5, 6], an early mutation testing tool, pioneered the general process typically used in mutation testing of creating mutants (of Fortran IV programs), accepting test cases from the users, and then executing the test cases on the mutants to decide how many mutants were killed. In 1987, this same process (of add test cases, run mutants, check results, and repeat) was adopted and extended in the Mothra mutation toolset [7, 8, 9, 10], which provided an integrated set of tools, each of which performed an individual, separate task to support mutation analysis and testing. Because each Mothra tool is a separate command, it was easy to incorporate, and thus experiment with, additional types of processing. Although a few other mutation testing tools have been developed since Mothra [11, 12, 13], Mothra is likely the most widely known mutation testing system extant.

Despite the relatively long history of mutation testing, the software development industry has failed to employ it. We posit that the three primary reasons why industry has failed to use mutation testing are the lack of economic incentives for stringent testing, inability to successfully integrate unit testing into software development processes, and difficulties with providing full and economical automated technology to support mutation analysis and testing. The first reason, the lack of economic incentives for applying highly advanced testing techniques, is beyond the scope of this paper, which is primarily technological in nature. On the other hand, software is increasingly being used to perform essential roles in applications that require high reliability, including safety-critical software (avionics, medical, and industrial control) infrastructure-critical software (telephony and networks), and commercial en-

*Supported by the National Science Foundation under awards CCR-9804011 and CCR-9707792.

terprises (e-commerce and business-to-business transactions). This increasing reliance on software implies that software must be increasingly be more reliable, thus we may expect there to be more economic incentives for applying high-end testing techniques such as mutation in the future.

The other two reasons for the lack of commercial success of mutation are primarily technological in nature. During the 1990s, a number of technological and theoretical advances were made in the application of mutation analysis and testing. Most of these advances are orthogonal, that is, they affect different aspects of mutation testing. This paper summarizes many of these advances and discusses ways to incorporate mutation into standard software development. It is thought that these advances, once united, can allow a truly practical mutation system to be built that can be used by real programmers on real software projects to greatly increase the reliability of their software products.

Before going into detail about these advances, a discussion of how mutation is used is given from a procedural point of view. Following that, a number of advances for applying mutation are discussed, which leads to a new process for how mutation can be applied. We envision a test tool that provides almost complete automation to the tester. A programmer submits a software module, and after a few minutes of computation, the tool responds with a set of test cases that are assured to provide the software with a very effective test, and a set of outputs that can be examined to find failures in the software. Furthermore, these input-output pairs can be used as a basis for debugging when failures are found. To be used by industry, this technology must be integrated with compilers, debuggers, and report generators.

1.1. The Mutation Analysis Process

Mutation analysis induces faults into software by creating many versions of the software, each containing one fault. Test cases are used to execute these faulty programs with the goal of distinguishing the faulty programs from the original program. Hence the terminology; faulty programs are *mutants* of the original, and a mutant is *killed* by distinguishing the output of the mutant from that of the original program.

Mutants either represent likely faults, a mistake the programmer could have made, or they explicitly require a typical testing heuristic to be satisfied, such as execute every branch or cause all expressions to become zero. Mutants are limited to simple changes on the basis of the *coupling effect*, which says that complex faults are coupled to simple faults in such a way that a test data set that detects all simple faults in a program will detect most complex faults The coupling effect was first hypothesized in 1978 [3], then supported empirically in 1992 [14], and has been demonstrated theoretically in 1995 [15, 16].

Mutation analysis provides a test *criterion*, rather than a test *process*. A *testing criterion* is a rule or collection of rules that imposes requirements on a set of test cases. Test engineers measure the extent to which a criterion is satisfied in terms of *coverage*; a set of test cases achieves 100% coverage if it completely satisfies the criterion. Coverage is measured in terms of the requirements that are imposed; partial coverage is defined to be the percent of requirements that are satisfied. *Test requirements* are specific things that must be satisfied or covered; for example, reaching statements are the requirements for statement coverage and killing mutants are the requirements for mutation. Thus, a test criterion establishes firm requirements for how much testing is necessary; a test process gives a sequence of steps to follow to generate test cases. There may be many processes used to satisfy a given criterion, and a test process need not have the goal of satisfying a criterion. In precise terms, mutation analysis is a way to measure the quality of the test cases and the actual testing of the software is a side effect. In practical terms however, the software is tested, and tested well, or the test cases do not kill mutants. This point can best be understood by examining a typical mutation analysis process.

When a program is submitted to a mutation system, the system first creates many mutated versions of the program. A *mutation operator*[1] is a rule that is applied to a program to create mutants. Typical mutation operators, for example, replace each operand by every other syntactically legal operand, or modify expressions by replacing operators and inserting new operators, or delete entire statements. Figure 1 graphically shows a traditional mutation process. The solid boxes represent steps that are automated by traditional systems such as Mothra, and the dashed boxes represent steps that are done manually.

Next, test cases are supplied to the system to serve as inputs to the program. Each test case is executed on the original program and the tester verifies that the output is correct. If incorrect, a bug has been found and the program should be fixed before that test case is used again. If correct, the test cases are executed on each mutant program. If the output of a mutant

[1] The terminology varies; they are also sometimes called *mutant operators*, *mutagenic operators*, *mutagens*, *mutation transformations*, and *mutation rules* [17].

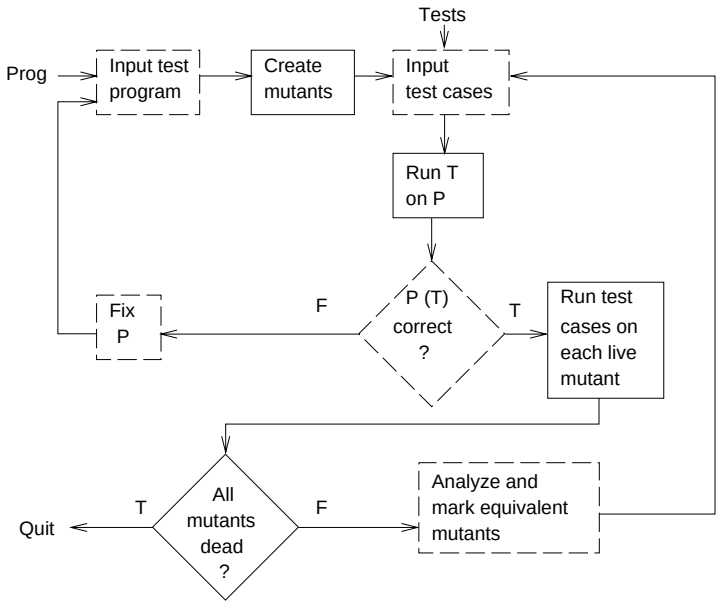

Figure 1: **Traditional Mutation Testing Process. Solid boxes represent steps that are automated and dashed boxes represent steps that are manual.**

program differs from the original (correct) output, the mutant is marked as being *dead*. Dead mutants are not executed against subsequent test cases.

Once all test cases have been executed, a *mutation score* is computed. The mutation score is the ratio of dead mutants over the total number of non-equivalent mutants. Thus, the tester's goal is to raise the mutation score to 1.00, indicating that all mutants have been detected. A test set that kills all the mutants is said to be *adequate* relative to the mutants.

If (as is likely) mutants are still alive, the tester can enhance the set of test cases by supplying new inputs. Some mutants are functionally *equivalent* to the original program. Equivalent mutants always produce the same output as the original program, so cannot be killed. Equivalent mutants are not counted in the mutation score. Note that even if the tester has not found any faults by using the previous set of test cases, the mutation score gives some indication of the extent of the testing. Moreover, the live mutants point out inadequacies in the test cases. In most cases, the tester creates test cases to kill specific live mutants. This process of adding new test cases, verifying correctness, and killing mutants is repeated until the tester is satisfied with the mutation score. A mutation score threshold can be set as a policy decision to require testers to test software to a predefined level.

2. Using Mutation Analysis to Detect Faults

Many research papers about mutation (including our own) have obscured the issue of how and when failures are found when using mutation. In standard IEEE terminology [18], a *failure* is an external, incorrect behavior of a program (an incorrect output or a runtime failure). A *fault* is the group of incorrect statements in the program that causes a failure. Failures in the software are detected when test cases are executed against the original program. The tester must decide whether the output of the program on each test case is correct. If the output is correct, the process continues as described above. If the output is incorrect, then a failure has been found and the process stops until the associated fault can be corrected. This leads to the fundamental premise of mutation testing, as coined by Geist [19]: **In practice, if the software contains a fault, there will usually be a set of mutants that can only be killed by a test case that also detects that fault**.

3. Bringing Down the Barriers

One of the barriers to the practical use of mutation testing is the unacceptable computational expense of gen-

erating and running vast numbers of mutant programs against the test cases. The number of mutants generated for a software unit is proportional to the product of the number of data references and the number of data objects [20]. Typically, this is a large number for even small software units. Because each mutant program must be executed against at least one, and potentially many, test cases, mutation analysis requires large amounts of computation. This is shown in Figure 1 in the box labeled "`Run test cases on each live mutant`". It is by far the most computationally expensive step in mutation testing.

The other barrier to more widespread use of mutation testing is the amount of manual labor involved in using this technique. For example, manual equivalent mutant detection is quite tedious and developing mutation adequate test cases can be very labor-intensive.

Recent advances show promise in bringing down both of these barriers. We first describe advances for reducing the computational expense of mutation analysis and then review research work that has been successful in partially automating much of the labor-intensive portions of mutation testing. We continue by suggesting how these advances can be combined in a manner that can lead to a practical mutation testing system in the near future.

3.1. Reducing the Computational Cost of Mutation Analysis

Recall that the major cost of mutation analysis arises from the computational expense of generating and running vast numbers of mutant programs. Approaches to reduce this computational expense usually follow one of three strategies: *do fewer*, *do smarter*, or *do faster*. The "do fewer" approaches seek ways of running fewer mutant programs without incurring intolerable information loss. The "do smarter" approaches seek to distribute the computational expense over several machines or factor the expense over several executions by retaining state information between runs or seek to avoid complete execution. The "do faster" approaches focus on ways of generating and running each mutant program as quickly as possible.

3.1.1. Selective Mutation – a "do fewer" approach

Mothra used 22 mutation operators, of which the six most populous account for 40% to 60% of all mutants. This is typical of mutation systems – the goal was to include as much testing as possible by defining as many mutants as possible. These six mutants, and others, are in some sense redundant; that is, test sets that are generated to kill only mutants generated from the other mutant operators are very effective in killing mutants generated from the six. Wong and Mathur suggested the idea of *constrained mutation* to be applying mutation with only the most critical mutation operators being used [21]. This idea was later developed by Offutt et al. as an approximation technique called *selective mutation* that tries to select only mutants that are truly distinct from other mutants [20, 22].

Results showed that of the 22 mutation operators used by Mothra, 5 turn out to be "key" operators. In experimental trials, those five operators provided almost the same coverage as non-selective mutation, with cost reductions of at least four times with small programs, and up to 50 times with larger programs. The 5 sufficient operators are ABS, which forces each arithmetic expression to take on the value 0, a positive value, and a negative value, AOR, which replaces each arithmetic operator with every syntactically legal operator, LCR, which replaces each logical connector (AND and OR) with several kinds of logical connectors, ROR, which replaces relational operators with other relational operators, and UOI, which inserts unary operators in front of expressions. Future mutation systems will have the goal of minimizing the number of mutation operators – getting as much testing strength as possible with as few mutants as possible.

3.1.2. Mutant Sampling – a "do fewer" approach

First proposed by Acree [23] and Budd [24], in sampling only a randomly selected subset of the mutant programs are run. The effects of varying the sampling percentage from 10% to 40% in steps of 5% were later investigated by Wong [25]. A 10% sample of mutant programs, for example, was found to be only 16% less effective than a full set in ascertaining fault detection effectiveness.

An alternative sampling approach is proposed by Şahinoğlu and Spafford [26] that does not use samples of some *a priori* fixed size but rather, based on a Bayesian sequential probability ratio test, selects mutant programs until sufficient evidence has been collected to determine that a statistically appropriate sample size has been reached.

3.1.3. Weak Mutation - a "do smarter" approach

Research systems such as Mothra execute mutant programs until they terminate, then compare the final output of the program with the output of the original program. Originally proposed by Howden [27], *weak mutation* is an approximation technique that compares

the internal states of the mutant and original program immediately after execution of the mutated portion of the program. That is, weak mutation ensures that the necessity condition is satisfied, but not the sufficiency condition.

Weak mutation has been discussed theoretically [28, 29, 30] and studied empirically [31, 32, 33, 34]. Howden's original proposal stated that the states should be compared "after" the mutated statement, without elaborating on exactly when. Morell's concept of "extent" [28] and Woodward and Halewood's "firm" mutation [29] suggested that the comparison could be done at any point after the mutated statement.

The Leonardo system [35, 34], which was implemented as part of Mothra, did two things. It implemented a working weak mutation system that could be easily compared with strong mutation, and evaluated the extent/firm concept by allowing comparisons to be made at four different locations after the mutated component: (1) after the first evaluation of the innermost expression surrounding the mutated symbol, (2) after the first execution of the mutated statement, (3) after the first execution of the basic block that contains the mutated statement, and (4) after **each** execution of the basic block that contains the mutated statement (execution stops as soon as an invalid state is detected). Experience with Leonardo indicated that weak mutation was able to generate tests that were almost as effective as tests generated with strong mutation, and that at least 50% and usually more of the execution time was saved. Moreover, it was found that the most effective point at which to compare the program states was after the first execution of the mutated statement.

3.1.4. Other "do smarter" approaches

Using novel computer architectures to distribute the computational expense over several machines represents another " do smarter" strategy. Work has been done to adapt mutation analysis systems to vector processors [36], SIMD machines [37], Hypercube (MIMD) machines [38, 39], and Network (MIMD) computers [40]. Because each mutant program is independent of all other mutant programs, communication costs are fairly low. At least one tool was able to achieve almost linear speedup for moderate sized program functions [38].

In another "do smarter" approach, Fleyshgakker and Weiss describe algorithms that improve the run time complexity of conventional mutation analysis systems at the expense of increased space complexity [41]. By intelligently storing state information, their techniques factor the expense of running a mutant over several related mutant executions and thereby lower the total computational costs. In the best case, these techniques can improve the speed by a factor proportional to the average number of mutants per program statement.

3.1.5. Schema-based Mutation Analysis – a "do faster" approach

Most mutation systems have worked by interpreting many slightly different versions of the same program. Although interpretation-based systems make the management of the mutant executions convenient, this conventional method has significant problems. Automated mutation analysis systems based on the conventional interpretive method are slow, laborious to build, and usually unable to completely emulate the intended operational environment of the software being tested. To solve these problems, Untch developed a new execution model for mutation, the Mutant Schema Generation (MSG) method [42, 12].

Instead of mutating an intermediate form, the MSG method encodes all mutations into one source-level program, a "metamutant". This program is then compiled (once) with the same compiler used during development and is executed in the same operational environment at compiled-program speeds. Because mutation systems based on mutant schemata do not need to provide the entire run-time semantics and environment, they are significantly less complex and easier to build than interpretive systems, as well as more portable. Benchmarks show `TUMS`, an MSG-based prototype mutation analysis system, to be significantly faster than `Mothra`, with speed-ups as high as an order-of-magnitude observed.

3.1.6. Other "do faster" approaches

Another way of avoiding interpretive execution is the *separate compilation* approach, wherein each mutant is individually created, compiled, linked and run. The `Proteum` system [13] is an example of this approach. When mutant run times greatly exceed individual compilation/link times, a system based on such a strategy will execute 15 to 20 times faster than an interpretive system. When this condition is not met, however, a *compilation bottleneck* [39] may result.

To avoid compilation bottlenecks, DeMillo, Krauser, and Mathur developed a *compiler-integrated* program mutation scheme that avoids much of the overhead of the compilation bottleneck and yet is able to execute compiled code [11]. In this method, the program under test is compiled by a special compiler. As the compilation process proceeds, the effects of muta-

tions are noted and *code patches* that represent these mutations are prepared. Execution of a particular mutant requires only that the appropriate code patch be applied prior to execution. Patching is inexpensive and the mutant executes at compiled-speeds.

3.2. Reducing Burdensome Manual Tasks

Manually developing test cases that are mutation adequate requires a great deal of effort. Additionally, determining which mutant programs are equivalent to the original program is a very tedious and error-prone activity. Progress has been made on partially automating both of these tasks and is described next.

3.2.1. Automatic Test Data Generation

One of the most difficult technical tasks in testing software is that of generating the test case values needed to satisfy the testing criterion. In his dissertation [9], Offutt developed a technique called *constraint-based test data generation* (*CBT*), which creates test data that comes reasonably close to satisfying mutation. CBT is based on the observation that a test case that kills a mutant must satisfy three conditions. The first is that the mutated statement must be reached; this is called the *reachability condition*. The second condition requires the execution of the mutated statement to result in an error in the program's state; this is called the *necessity condition*. The third condition, the *sufficiency condition*, states that the incorrect state must propagate through the program's computation to result in an output failure. *Godzilla* is a test data generator that uses constraint-based testing to automatically generate test data for Mothra [10].

Godzilla describes these conditions as mathematical systems of constraints. Reachability conditions are described by constraint systems called *path expressions*. Each statement in the program has a path expression that describes all execution paths through the program to that statement. The path expression is an assertion that is true if the statement is reached. The necessity condition is described by a constraint that is specific to the mutant operator and requires that the computation performed by the mutated statement create an incorrect intermediate program state. Because expressing the sufficiency condition as a set of constraints requires knowing in advance the complete path a program will take (in general, undecidable), Godzilla does not attempt to automatically satisfy this condition directly.

Godzilla conjoins each necessity constraint with the appropriate path expression constraint. The resulting constraint system is solved to generate a test case such that the constraint system is true. Experimentation [43] has verified that constraint-based testing creates test cases that kill over 90% of the mutants for most programs. CBT uses control-flow analysis, symbolic evaluation, and information about mutants to create the constraints, and a constraint satisfaction technique called *domain reduction* to generate test values.

CBT suffers from several shortcomings that prevent it from working in some situations and hamper its applicability in practical situations. Many of these shortcomings stem from weaknesses associated with symbolic evaluation and include problems handling arrays, loops, and nested expressions. Godzilla occasionally fails to find test cases, and for some programs it fails a large percentage of the time. This is partly because of problems with the technique, partly because of insufficiently general approaches to handling expressions, and partly because Godzilla employed relatively unsophisticated search procedures.

More recently, a test data technique called the *dynamic domain reduction procedure* was developed to address most of these problems [44, 45]. The dynamic domain reduction procedure (DDR) uses part of the CBT approach, and also draws from Korel's dynamic test data generation approach [46, 47] and symbolic evaluation. It uses a direct "domain reduction" method for deriving values, rather than function minimization methods as used by Korel or linear programming-like methods as used by Clarke [48]. Korel's dynamic method [47] executes a program along one specific path by starting with a particular input. When a branching point is reached, if the current inputs will cause the appropriate branch to be taken, the inputs will remain the same. If a different branch is required, then the inputs are dynamically modified to take the correct branch using function minimization. DDR also works by choosing a specific path, but there are no initial values, and the values are derived in-process from initial input domains.

Unlike dynamic symbolic evaluation [49, 50], DDR creates sets of values that represent conditions under which a path will be executed. Thus, the results of dynamic symbolic evaluation attempt to represent all possible values that will execute a given path, while dynamic domain reduction only results in a small set of possible values. While this is more limited, it is also more practical for real programs.

The dynamic nature of DDR, which combines analysis of the software with satisfaction of constraints and test data generation, allows better handling of arrays and expressions. DDR also incorporates a sophisticated back-tracking search procedure to partially

solve a problem that caused previous methods to fail. Because of the historical basis, the DDR procedure will always work when CBT does, and also in many cases when CBT does not.

The DDR procedure walks through the program control flow graph, generating test data along the way. Each input variable is initially given a large set of potential values (its *domain*) and, as branches are taken in the control flow graph, the domains for the variables involved in the predicates are reduced so that the appropriate predicates would be true for any assignment of values from the domain. When choices for how to reduce the domains must be made, a search process is initiated and choices are systematically made to try to find a choice that allows the subsequent edges on the path to be executed. When the procedure is finished, the remaining values for the variables' domains represent sets of test cases that will cause execution of the path. If any variable's domain is empty, the search process failed due to one of two possible reasons. One, the path is infeasible, so no satisfying values could be found. Two, it was very difficult to find values that execute the path; this could be because the constraints were too complicated or there are relatively few inputs that will execute the path.

3.2.2. Partial Automatic Equivalent Mutant Detection

A major problem with practically applying mutation is that of equivalent mutant programs. Equivalent mutants can be thought of as "dead-weight" in the testing process – they do not contribute to the generation of test cases, but require lots of time and attention from the tester. Equivalent mutants have traditionally been detected by hand, which is very expensive and time-consuming, and restricts the practical usefulness of mutation testing.

Although recognition of equivalent programs is in general undecidable [51], the idea of using compiler-optimization techniques to recognize some if not most equivalent mutants was suggested by Baldwin and Sayward in 1979 [52]. This technique was tried in a limited way by hand in Tanaka's 1981 thesis [53]. Offutt and Craft [54] refined, extended, and implemented the Baldwin and Sayward suggestions in a tool that was integrated with Mothra. This led directly to the idea of using constraint-based testing to detect equivalent mutants, which was implemented in a tool that detected almost 50% of the equivalent mutants [55, 56].

The constraint-based technique uses mathematical constraints to automatically detect equivalent mutants. The general idea is that if a constraint system that is created to kill a mutant is infeasible, then that mutant is equivalent. Although recognizing infeasible constraints is a difficult problem that cannot be solved in general, heuristic approximations have been developed that are quite effective. This approach also subsume all of the previous compiler-optimization techniques.

Hierons, Harman, and Danicic have gone one step further and use program slicing to detect equivalent mutants [57]. This approach in turn subsumes the constraint-based technique.

Unfortunately, no automated system will be able to detect all equivalent mutants, thus to complement the technique of recognizing equivalent mutants, we suggest that the remaining equivalent mutants can be safely ignored. Although this requires the tester to be willing to accept less than full mutation coverage, results indicate that the loss will not usually be significant, and the testing will still be more effective than testing with most other testing techniques. Although this approach is not completely satisfying from a theoretical view, it is an eminently practical engineering solution to a practically impossible problem.

3.3. Procedural Advances using Mutation

The traditional mutation testing process as shown in Figure 1 suffers from several problems. The major loop of entering test cases, running the original program, checking the output, running mutants, and marking equivalent mutants is very human intensive. The advances in automatic test data generation led to viewing test cases as throw-away items, rather than valuable resources. This in turn leads to the realization that checking whether the original program is correct on each test case does not have to be in the major loop, but can be postponed until later. This, combined with the automation of steps that were previously manual, allows us to eliminate the human tester from the main mutation loop.

Figure 2 presents a new process for applying mutation testing. Initially, a set of test cases is automatically generated and those test cases are executed against the original program, and then the mutants. The tester defines a "threshold" value, which is a minimum acceptable mutation score. If the threshold has not been reached, then test cases that killed no mutants (termed *ineffective*) are removed. This process is repeated, each time generating test cases to target live mutants, until the threshold mutation score is reached. Up to this point, the process has been entirely automatic. To finish testing, the tester will examine expected output of the effective test cases, and fix the

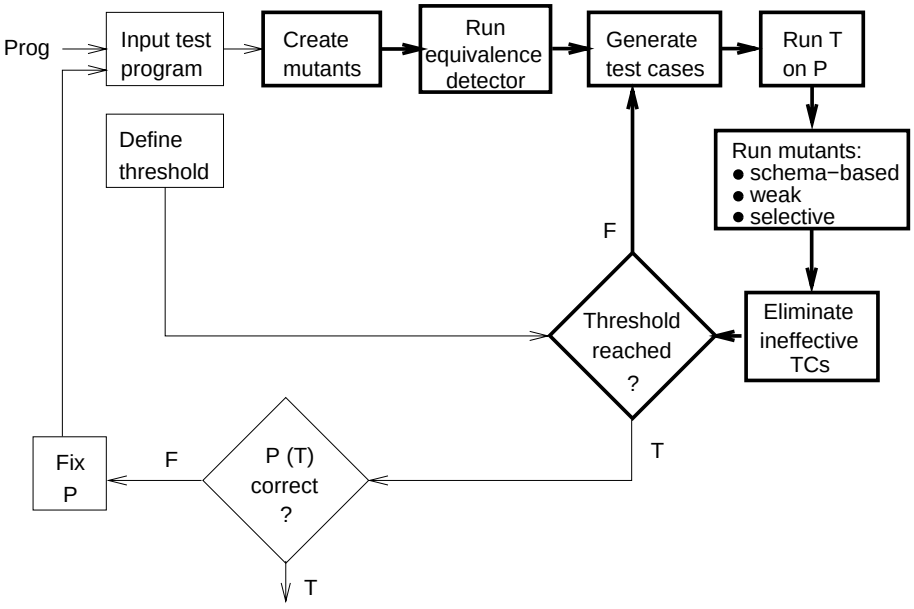

Figure 2: **New Mutation Testing Process.**
Bold boxes represent steps that are automated;
remaining boxes represent steps that are manual.

program if any faults are found.

In both the traditional and this new process, the major part of the time and effort of mutation is in the loop of generating, running, and disposing of test cases. As before, the most significant computational expense is in the mutation engine. The difference is that the new mutation engine will employ a schema-based approach (or one of the other "do faster" techniques) to execute the code, apply weak mutation to reduce the amount of execution, and use selective mutation to reduce the number of mutations. Given that the improvements given in Section 3.1 are orthogonal (rather than serial), combining these techniques will yield geometric (rather than incremental) performance boosts. Uniting these orthogonal techniques should reduce the amount of execution time for the mutation engine by **orders of magnitude**, making it possible to process moderate routines within a reasonable period of time.

A significant innovation in this new process is that the major loop (the boxes and arrows in bold in Figure 2) contains no manual steps. All manual steps are outside the loop and only need to be done once. In fact, the only significant manual step is that of deciding if the outputs of each test case is correct. Some progress on constructing automated test oracles has been made [58, 59]. However even without an automated test oracle, by disposing of ineffective test cases before checking outputs we can significantly reduce the workload of the tester. Additionally, the threshold input allows the practical tester to use approximation in the coverage criterion. This approximation heuristic does not attempt to find an exact solution to the testing problem. Software testing is an imperfect science and we see no reason for coverage to be exact. Rather, a application of a coverage criterion must be cost-effective and must always improve the situation—by providing better test cases. The use of a threshold heuristic meets this requirement.

4. A Practical and Effective Mutation Analysis System

Using these technological advances and process improvements, practical, 21st century mutation systems will be faster and more practical, and require significantly less human interaction. Figure 3 presents a high level architectural view of this type of testing system.

In this system, a program to be tested is submitted to the `schemata generator`, which produces a `metamutant` that incorporates all the mutants of the test program into one program. The `schemata generator` also produces a `mutant data store`, which is used to store statistics about the mutants such as which are alive and which have been killed. The `constraint & slicing analyzer` integrates the pre-

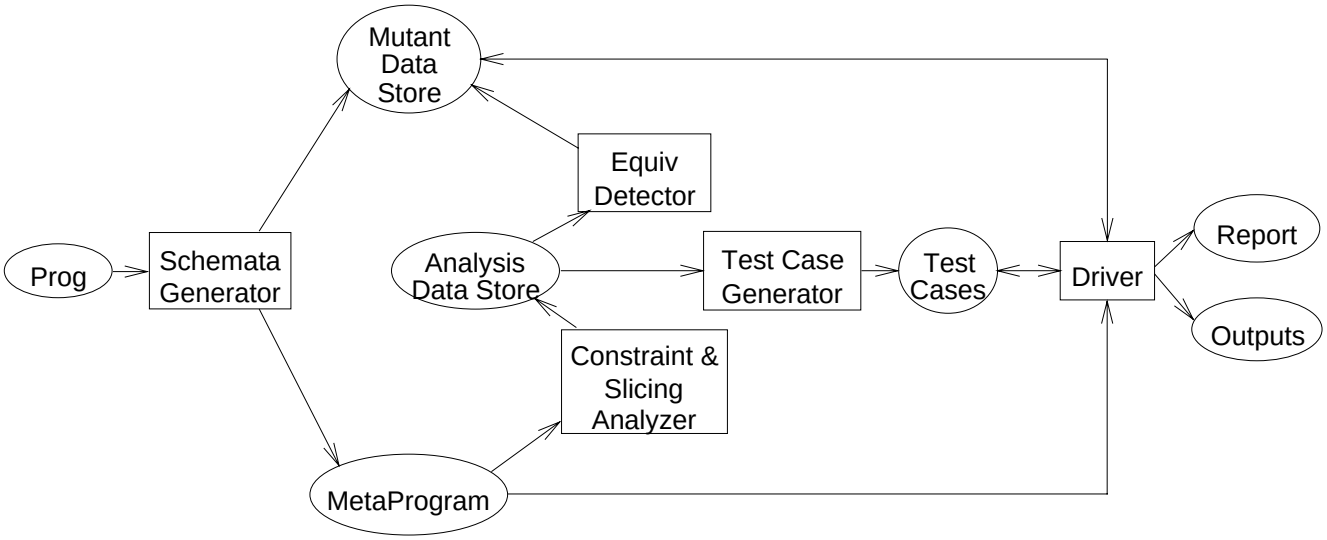

Figure 3: **Architecture of a Practical, Efficient Mutation Testing System**

processing steps for test data generation and equivalent mutant detection into one tool. It produces constraints on mutants as well as control, data flow, and slicing information about the program and its mutants. The `analysis data store` is used by the `equivalent mutant detector` to detect and mark mutants that are equivalent in the `mutant data store`. The `test case generator` uses the `analysis data store` to generate test cases to try to kill each mutant. The `driver` compiles the `metamutant`, runs each test case on the original program, then on each mutant, saving only test cases that kill at least one mutant. It continues to obtain and execute more test cases until the threshold percentage of mutants is reached. The results of running the mutants are saved in the `mutant data store`, and a report summarizing how many mutants have been killed is generated. The output of the original program on each effective test case is saved for examination by the tester.

The `schemata generator` only generates mutants using the selective operators. This consists of mutants that replace each arithmetic operator with each other arithmetic operator, replace each relational operator with each other relational operator, replace each logical connector operator with each other logical connector operator, and that modify expressions by inserting unary operations that cause each expression to be zero, negative, positive, and that modify each expression by very small amounts. The `metamutant` incorporates weak mutation semantics, so that each mutant will not execute completely, but will only execute to the end of the basic block that contains the mutated statement.

Because parallelism depends very heavily on the type of hardware available, we do not automatically assume it will be incorporated in future mutation systems. However, if it is thought to be helpful, the `driver` in Figure 3 could be modified to execute programs from the `metamutant` in parallel.

5. Conclusions

By combining the recent technological advances and an improved testing process, future mutation testing tools will be orders of magnitude faster than previous research systems and will require significantly less human involvement. Additionally, experience has shown that these systems can be built easier and faster than previous systems.

We envision a test tool that provides almost complete automation to the tester. A programmer submits a software module, and after a reasonable period of computation, the tool responds with a set of test cases that are assured to provide the software with a very effective test, and a set of outputs that can be examined to find failures in the software. Furthermore, these input-output pairs can be used as a basis for debugging when failures are found. This technology can be integrated with compilers, debuggers, and report generators.

6. Acknowledgments

We would like to thank the many students and faculty colleagues who contributed to the ideas and tool development to support the research contained in this paper, including Roger Alexander, Michael Craft, Scott Fichter, Dr. Robert Geist, Fred Harris, Dr. Mary Jean Harrold, Zhenyi Jin, Ammei Lee, Stephen Lee, Tracey Oakes, Jie Pan, Dr. Gregg Rothermel, and Christian Zapf.

References

[1] T. Budd and F. Sayward, "Users guide to the Pilot mutation system," technical report 114, Department of Computer Science, Yale University, 1977.

[2] R. G. Hamlet, "Testing programs with the aid of a compiler," *IEEE Transactions on Software Engineering*, vol. 3, pp. 279–290, July 1977.

[3] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *IEEE Computer*, vol. 11, pp. 34–41, April 1978.

[4] T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "The design of a prototype mutation system for program testing," in *Proceedings NCC, AFIPS Conference Record*, pp. 623–627, 1978.

[5] R. J. Lipton and F. G. Sayward, "The status of research on program mutation," in *Digest for the Workshop on Software Testing and Test Documentation*, pp. 355–373, December 1978.

[6] 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.

[7] A. J. Offutt and K. N. King, "A Fortran 77 interpreter for mutation analysis," in *1987 Symposium on Interpreters and Interpretive Techniques*, (St. Paul MN), pp. 177–188, ACM SIGPLAN, June 1987.

[8] 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*, (Banff Alberta), pp. 142–151, IEEE Computer Society Press, July 1988.

[9] A. J. Offutt, *Automatic Test Data Generation*. PhD thesis, Georgia Institute of Technology, Atlanta GA, 1988. Technical report GIT-ICS 88/28.

[10] R. A. DeMillo and A. J. Offutt, "Constraint-based automatic test data generation," *IEEE Transactions on Software Engineering*, vol. 17, pp. 900–910, September 1991.

[11] 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 (COMPSAC' 92)*, (Tokyo, Japan), Kogakuin University, IEEE Computer Society Press, September 1991.

[12] R. H. Untch, M. J. Harrold, and J. Offutt, "Schema-based mutation analysis." In preparation.

[13] M. E. Delamaro and J. 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)*, (New Brunswick, NJ), pp. 79–95, July 1996.

[14] A. J. Offutt, "Investigations of the software testing coupling effect," *ACM Transactions on Software Engineering Methodology*, vol. 1, pp. 3–18, January 1992.

[15] K. S. H. T. Wah, "Fault coupling in finite bijective functions," *The Journal of Software Testing, Verification, and Reliability*, vol. 5, pp. 3–47, March 1995.

[16] K. S. H. T. Wah, "A theoretical study of fault coupling," *The Journal of Software Testing, Verification, and Reliability*, vol. 10, pp. 3–46, March 2000.

[17] D. Wu, M. A. Hennell, D. Hedley, and I. J. Riddell, "A practical method for software quality control via program mutation," in *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, (Banff, Alberta, Canada), pp. 159–170, IEEE Computer Society Press, July 1988.

[18] IEEE, *IEEE Standard Glossary of Software Engineering Terminology*. ANSI/IEEE Std 610.12-1990, 1996.

[19] R. Geist, A. J. Offutt, and F. Harris, "Estimation and enhancement of real-time software reliability through mutation analysis," *IEEE Transactions on Computers*, vol. 41, pp. 550–558, May 1992. Special Issue on Fault-Tolerant Computing.

[20] A. J. Offutt, A. Lee, G. Rothermel, R. Untch, and C. Zapf, "An experimental determination of sufficient mutation operators," *ACM Transactions on Software Engineering Methodology*, vol. 5, pp. 99–118, April 1996.

[21] W. E. Wong, M. E. Delamaro, J. C. Maldonado, and A. P. Mathur, "Constrained mutation in C programs," in *Proceedings of the 8th Brazilian Symposium on Software Engineering*, (Curitiba, Brazil), pp. 439–452, October 1994.

[22] A. J. Offutt, G. Rothermel, and C. Zapf, "An experimental evaluation of selective mutation," in *Proceedings of the Fifteenth International Conference on Software Engineering*, (Baltimore, MD), pp. 100–107, IEEE Computer Society Press, May 1993.

[23] A. T. Acree, *On Mutation*. PhD thesis, Georgia Institute of Technology, Atlanta GA, 1980.

[24] T. A. Budd, *Mutation Analysis of Program Test Data*. PhD thesis, Yale University, New Haven CT, 1980.

[25] W. E. Wong, *On Mutation and Data Flow*. PhD thesis, Purdue University, December 1993. (Also Technical Report SERC-TR-149-P, Software Engineering Research Center, Purdue University, West Lafayette, IN).

[26] M. Şahinoğlu and E. H. Spafford, "A bayes sequential statistical procedure for approving software products," in *Proccedings of the IFIP Conference on Approving Software Products (ASP–90)* (W. Ehrenberger, ed.), (Garmisch-Partenkirchen, Germany), pp. 43–56, Elsevier/North Holland, New York, Sept. 1990.

[27] W. E. Howden, "Weak mutation testing and completeness of test sets," *IEEE Transactions on Software Engineering*, vol. 8, pp. 371–379, July 1982.

[28] L. J. Morell, "Theoretical insights into fault-based testing," in *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, (Banff Alberta), pp. 45–62, IEEE Computer Society Press, July 1988.

[29] 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*, (Banff Alberta), pp. 152–158, IEEE Computer Society Press, July 1988.

[30] 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.

[31] 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*, (London UK), pp. 313–319, IEEE Computer Society Press, August 1985.

[32] B. Marick, "Two experiments in software testing," technical report UIUCDCS-R-90-1644, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana-Champaign Illinois, November 1990.

[33] B. Marick, "The weak mutation hypothesis," in *Proceedings of the Fourth Symposium on Software Testing, Analysis, and Verification*, (Victoria, British Columbia, Canada), pp. 190–199, IEEE Computer Society Press, October 1991.

[34] A. J. Offutt and S. D. Lee, "An empirical evaluation of weak mutation," *IEEE Transactions on Software Engineering*, vol. 20, pp. 337–344, May 1994.

[35] A. J. Offutt and S. D. Lee, "How strong is weak mutation?," in *Proceedings of the Fourth Symposium on Software Testing, Analysis, and Verification*, (Victoria, British Columbia, Canada), pp. 200–213, IEEE Computer Society Press, October 1991.

[36] A. P. Mathur and E. W. Krauser, "Mutant unification for improved vectorization," technical report SERC-TR-14-P, Software Engineering Research Center, Purdue University, West Lafayette IN, April 1988.

[37] 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*, (Banff Alberta), pp. 171–177, IEEE Computer Society Press, July 1988.

[38] 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*, (Chicago, Illinois), pp. II–257–266, August 1992.

[39] B. Choi and A. P. Mathur, "High-performance mutation testing," *The Journal of Systems and Software*, vol. 20, pp. 135–152, February 1993.

[40] C. N. Zapf, "Medusamothra – a distributed interpreter for the mothra mutation testing system," M.S. thesis, Clemson University, Clemson, SC, August 1993.

[41] V. N. Fleyshgakker and S. N. Weiss, "Efficient Mutation Analysis: A New Approach," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 94)*, (Seattle, WA), pp. 185–195, ACM SIGSOFT, ACM Press, Aug. 17–19 1994.

[42] 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*, (Cambridge MA), pp. 139–148, June 1993.

[43] R. A. DeMillo and A. J. Offutt, "Experimental results from an automatic test case generator," *ACM Transactions on Software Engineering Methodology*, vol. 2, pp. 109–127, April 1993.

[44] J. Offutt, Z. Jin, and J. Pan, "The dynamic domain reduction approach for test data generation: Design and algorithms," technical report ISSE-TR-94-110, Department of Information and Software Systems Engineering, George Mason University, Fairfax VA, September 1994.

[45] J. Offutt, Z. Jin, and J. Pan, "The dynamic domain reduction approach to test data generation," *Software–Practice and Experience*, vol. 29, pp. 167–193, January 1999.

[46] B. Korel, "Automated software test data generation," *IEEE Transactions on Software Engineering*, vol. 16, pp. 870–879, August 1990.

[47] B. Korel, "Dynamic method for software test data generation," *The Journal of Software Testing, Verification, and Reliability*, vol. 2, no. 4, pp. 203–213, 1992.

[48] L. A. Clarke, "A system to generate test data and symbolically execute programs," *IEEE Transactions on Software Engineering*, vol. 2, pp. 215–222, September 1976.

[49] L. A. Clarke and D. J. Richardson, "Applications of symbolic evaluation," *The Journal of Systems and Software*, vol. 5, pp. 15–35, January 1985.

[50] R. E. Fairley, "An experimental program testing facility," *IEEE Transactions on Software Engineering*, vol. SE-1, pp. 350–3571, December 1975.

[51] T. A. Budd and D. Angluin, "Two notions of correctness and their relation to testing," *Acta Informatica*, vol. 18, pp. 31–45, November 1982.

[52] D. Baldwin and F. Sayward, "Heuristics for determining equivalence of program mutations," research report 276, Department of Computer Science, Yale University, 1979.

[53] A. Tanaka, "Equivalence testing for fortran mutation system using data flow analysis," Master's thesis, School of Information and Computer Science, Georgia Institute of Technology, Atlanta GA, 1981.

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

[55] A. J. Offutt and J. Pan, "Detecting equivalent mutants and the feasible path problem," in *Proceedings of the 1996 Annual Conference on Computer Assurance (COMPASS 96)*, (Gaithersburg MD), pp. 224–236, IEEE Computer Society Press, June 1996.

[56] A. J. Offutt and J. Pan, "Detecting equivalent mutants and the feasible path problem," *The Journal of Software Testing, Verification, and Reliability*, vol. 7, pp. 165–192, September 1997.

[57] R. Hierons, M. Harman, and S. Danicic, "Using program slicing to assist in the detection of equivalent mutants," *Software Testing, Verification, and Reliability*, vol. 9, pp. 233–262, December 1999.

[58] E. Mikk, "Compilation of z specifications into c for automatic test result evaluation," in *9th International Conference of Z Users (ZUM'95)*, (Limerick, Ireland), pp. 167–180, Springer-Verlag Lecture Notes in Computer Science Volume 967, J.P. Bowen and M.G. Hinchey (Eds.), September 1995.

[59] D. K. Peters and D. L. Parnas, "Using test oracles generated from program documentation," *IEEE Transactions on Software Engineering*, vol. 24, pp. 161–173, March 1998.