skip me

skip me

A Practical System for Mutation Testing: Help for the Common Programmer

A. Jefferson Offutt [†]
ISSE Department
George Mason University
Fairfax, VA 22030
phone: 703-993-1654
fax: 703-993-1638
email: ofut@isse.gmu.edu

Abstract – *Mutation testing is a technique for unit testing software that, although powerful, is computationally expensive. 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, and algorithms for automatic test data generation. This paper 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.*

1 INTRODUCTION

Software testing activities are described in terms of the scope of the software being tested. *Unit* testing validates small subroutines and functions. *Integration* testing validates complete programs or significant subprograms. Most formal testing research is currently at the unit level, whereas most industry testing is at the integration level.

Unit testing is typically left to the individual programmers, who are given little or no formal training or test tools. The few tools that are available are usually test *support* tools such as drivers or test case managers, rather than tools that solve the hard problems of generating test cases to satisfy formal criteria. In extreme cases, the testing problem is solved by a huge investment in labor. It is common, when developing mission critical software, to use as many as one tester for every programmer. Testing researchers, on the other hand, try to develop testing techniques that can be utilized to test a wide variety of software. Formal testing techniques such as statement coverage, branch coverage, mutation analysis, and data flow coverage were developed to test software units, and there is little or no reason to believe that they can be applied to complete programs.

There are good reasons for both groups' view of testing. A tester in industry will claim that unit testing is too expensive, whereas a testing researcher will point out that we find failures more efficiently and in greater numbers during unit testing. Unfortunately, both sides are correct.

Unit testing is too expensive because software contains orders of magnitudes more pieces than integration components, which means more testing. Unit level testing methods are not currently used because they require manual application, and the labor costs are simply too high. To apply mutation testing, for example, a tester needs to analyze hundreds or thousands of similar versions of the software, in many cases performing a careful hand analysis of small differences between the programs, generate test cases for the program, and verify the outputs of each test case. It is reasonable for this to take four hours for a single subroutine, which means if the programmer is working on an air traffic control system with 100,000 subroutines, it will take almost 200 person-years to test the system! This is too expensive for practical application.

In software testing, a *failure* is external, incorrect behavior of a program – incorrect output, or runtime failure. A *fault* is the incorrect statement in the program that causes a failure. Because program units are so much smaller, testers can find failures more efficiently during unit testing. In addition, it is easier to track down and solve faults (debug) in software units. Software units also tend to be generic code, which make them more amenable to testing by general-purpose, formal strategies. And what happens to the faults that we do not find when we skip unit testing? They are left in the software for the users to find, and for the maintainers to fix. This hidden cost of not doing unit testing is borne not by the developers

[†]Partially supported by the National Science Foundation under grant CCR-93-11967.

or the testers, but by the users and maintainers of the software. This cost is eventually charged against the customers' "good will" towards the company, which is decreased every time a customer encounters a software failure.

Industry needs to apply unit testing, but researchers first need to develop the technology necessary to do so. Unit testing will only, **can** only, be practically applied if the techniques are automated as completely as theoretically possible. To use the known formal unit testing techniques, we need to decrease the cost of unit testing through massive automation. This paper discusses attempts to reduce the cost of one testing technique, mutation analysis, by a mixture of technological advances and approximation techniques, and presents a new process for applying mutation testing that is incorporated into a design for a new mutation system.

The eventual goal of this research is a testing system that is highly effective, efficient, and convenient to use. This system should be usable by programmers, and integrated with their normal software development environment. The testing system will allow the programmer to submit a program unit or module to the system, and receive back a set of test cases that are guaranteed to effectively test the program, and corresponding outputs that the programmer must examine to determine on what inputs the software failed. This system could be integrated with a debugger, so that the testing system supplies additional information about the test cases that failed; in particular, information about the program statements that the test cases targeted (which are likely places for the programmer to examine for the fault).

2 Mutation Testing Overview

Mutation testing helps a user create test data by interacting with the user to iteratively strengthen the quality of test data. During mutation testing, faults are introduced into a program by creating many versions of the program, each of which contains one fault. Test data are used to execute these faulty programs with the goal of causing each faulty program to fail. Hence we use the term mutation; faulty programs are *mutants* of the original, and a mutant is *killed* when a test case causes it to fail. When this happens, the mutant is considered *dead* and no longer needs to remain in the testing process since the faults represented by that mutant have been detected, and more importantly, it has satisfied its requirement of identifying a useful test case.

Figure 1 contains a small Fortran function with three mutated lines (preceded by the Δ symbol). Note that each of the mutated statements represents a separate program. The most recent mutation system, Mothra [2, 6], uses 22 mutation operators to test Fortran-77 programs. These operators have been developed and refined over 10 years through several mutation systems. The mutation operators 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 [3]. The coupling effect has been supported experimentally [9] and theoretically [8].

```
        FUNCTION Min (I,J)
        Min = I
    Δ   Min = J
        IF (J .LT. I) Min = J
    Δ   IF (J .GT. I) Min = J
    Δ   IF (J .LT. Min) Min = J
        RETURN
```

Figure 1: **Function Min.**

The mutation testing process begins with the construction of mutants of a test program. The user then adds test cases (generated manually or automatically) to the mutation system and checks the output of the program on each test case to see if it is correct. If the output is incorrect, a fault has been found and the program must be modified and the process restarted. If the output is correct, that test case is executed against each live mutant. If the output of a mutant differs from that of the original program on the same test case, the mutant is assumed to be incorrect and it is killed.

After each test case has been executed against each live mutant, each remaining mutant falls into one of two categories. One, the mutant is functionally *equivalent* to the original program. An equivalent mutant always produces the same output as the original program, so no test case can kill it. Two, the mutant is killable, but the set of test cases is insufficient to kill it. In this case, new test cases need to be created, and the process iterates until the test set is strong enough to satisfy the tester. The *mutation score* for a set of test data is *the percentage of non-equivalent mutants killed by that data.* We call a test data set *mutation-adequate* if its mutation score is 100%.

Figure 2 graphically shows the 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. Recent advances, described below, have enabled us to modify this process to make mutation testing more automated and more practical. We refer to the process shown in Figure 2 as the *traditional mutation testing process.*

2.1 The Cost of Mutation Testing

The major computational cost of mutation testing is incurred when running the mutant programs against

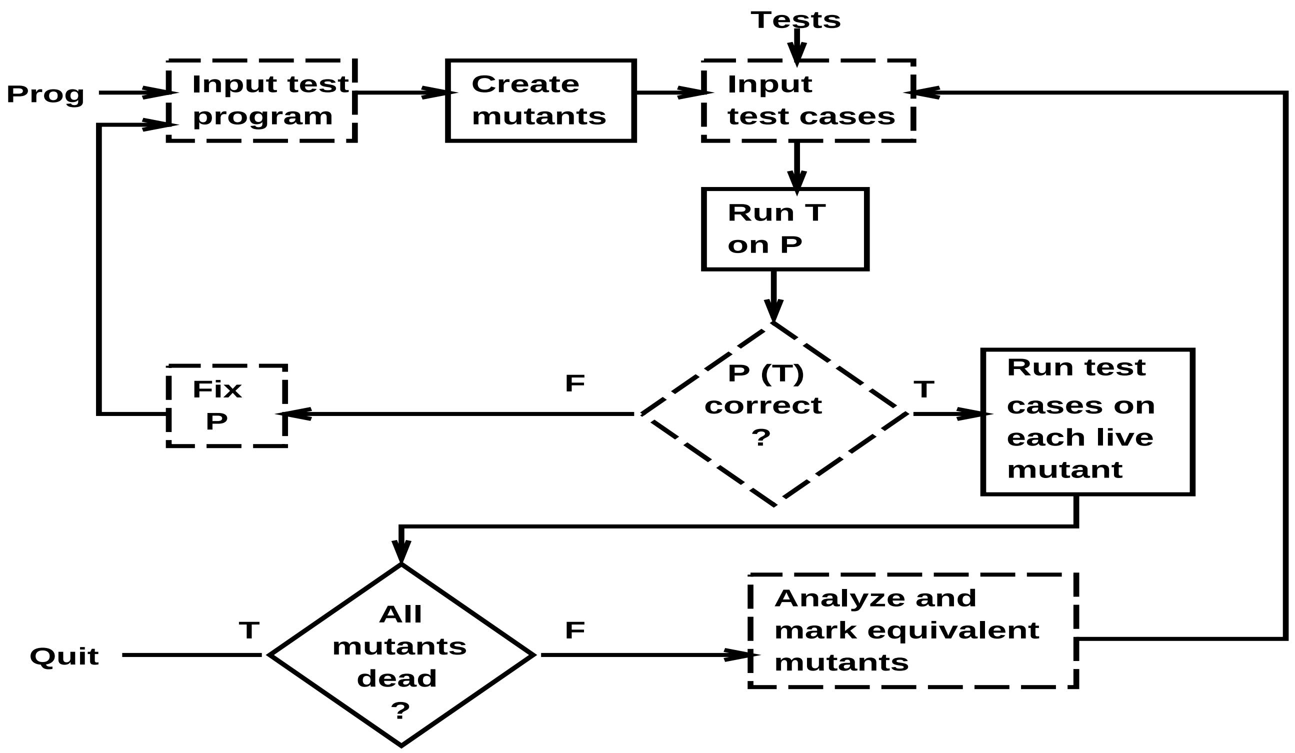

Figure 2: **Traditional Mutation Testing Process. Solid boxes represent steps that are automated and dashed boxes represent steps that are manual.**

the test cases. Budd [1] analyzed the number of mutants generated for a program and found it to be roughly proportional to the product of the number of data references times the number of data objects. Recent empirical measurements have validated this estimate over a number of programs [10]. Typically, this is a large number for even small program units. For example, 44 mutants are generated for the function `Min` shown in Figure 1. Since each mutant must be executed against at least one, and potentially many, test cases, mutation testing requires large amounts of computation. This is shown in Figure 2 in the box labeled "`Run test cases on each live mutant`". It is by the far the most computationally expensive step in mutation testing.

Until recently this cost has been too great to allow mutation to be used in a practical way. For example, a 30 minute procedure that can be written in one or two hours will take several hours to test using traditional mutation systems. Using the techniques described here, it should be possible to test the same procedure in 10 or 15 minutes.

There are also several manual costs associated with traditional mutation systems. In Figure 2, the solid steps are performed automatically, and the dashed steps are performed manually. The process we develop through this paper eliminates the two manual steps of inputting test cases and analyzing equivalent mutants, which dramatically reduces the human cost of applying mutation. Unfortunately, we cannot eliminate the resulting major human cost, determining if the output of each test case is correct. We do however, as a result of our test data generation ability, modify the mutation process so as to reduce the number of test cases for which the programmer needs to determine output correctness.

3 Constraint-based Test Data Generation

One of the most difficult technical task in test software is that of generating the test case values needed to satisfy the testing criteria. Constraint-based test data generation (*CBT*) is a set of procedures designed to create test data that satisfy mutation. CBT is based on the observation that a test case that kills a mutant must satisfy three conditions. The *reachability* condition is that the mutated statement must be reached. A further condition is that once the mutated statement is executed, the test case must cause the mutant program to behave erroneously—the fault that is being modeled must result in a failure in the program's behavior; this is called the *necessity condition*. The *sufficiency condition* states that the incorrect state must propagate through the program's computation to result in a failure. *Godzilla* is a test data generator that uses constraint-based testing to automatically generate test data for Mothra.

Godzilla describes these conditions on the test cases 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 each execution path through the program to that statement. The condition that the test case must cause an erroneous state is described by a constraint that is specific to the type of fault being modeled by each mutation, and requires that the computation performed by the mutated statement create an incorrect intermediate program state. This is called a *necessity constraint* because although an incorrect intermediate program state is necessary to kill the mutant, it is not sufficient to kill it. To kill the mutant, the test case must cause the program to create incorrect output, in which case the final state of the mutant program differs from that of the original program. Although satisfying the *sufficiency condition* is certainly desirable, it is impractical in practice. Completely determining the sufficiency condition implies knowing in advance the complete path a program will take, which is intractable.

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.

Constraint-based testing solves a major problem with using mutation testing as a practical method for testing software, that of creating test data. Constraint-based testing has been fully implemented and integrated with the Mothra testing system. Experimentation [4] has verified that constraint-based testing creates test cases that score well on the mutation system. DeMillo and Offutt [4] observed that an automatic test data generation capability allows us to view test cases as "throw-away" items rather than expensive, scarce resources. With this view, we can generate test cases, toss them at mutants, and then throw them away if they do not work.

This means that we can examine the output of a test case **after** mutants had been executed. By postponing this expensive (manual!) step, we only have to look at the output of the program on effective test cases, rather than all test cases.

4 Engineering Advances in Mutation Analysis

The new technological advances in mutation testing individually offer speedup in the application of mutation testing, and collectively may increase the speed of mutation tools by orders of magnitude. All of these techniques have been studied in laboratory experiments, although have not yet been implemented in a production system. These techniques fall into three broad categories. First, the speed of mutation execution is reduced by using *schema-based* mutant execution, *weak* mutation, and *selective* mutation. Second, automated test data generation is used to reduce manual creation of test cases. Third, an optimistic approach is used to reduce manual intervention into mutation analysis. These ideas are discussed in more detail in the following subsections.

4.1 Weak Mutation

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. *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 [5, 7, 11]. Experimental studies have shown that this technique can save at least 50%, and usually more, of the execution without a serious degradation in the quality of the test cases.

4.2 Schema-based Mutation Analysis

Mothra translates programs to an intermediate form and creates mutants by modifying the intermediate form. To execute the mutants, the intermediate form is interpreted. As an interpretive-based system, Mothra suffers from the expected problem — it is slow. In addition, interpretive-based systems are laborious to build and do not completely emulate the intended operational environment of the software being tested. A new technique for performing mutation analysis uses *program schemata* to encode all mutants for a program into one *metaprogram*, which is subsequently compiled and run at speeds substantially higher than achieved by previous interpretive systems. Preliminary performance improvements of over 300% are reported [13, 14]. This technique has the additional advantages of being easier to implement than interpretive systems, easier to port across a wide range of hardware and software platforms, and using the same compiler and run-time support system that is used during development and/or deployment.

4.3 Selective Mutation

Mothra uses 22 mutation operators, of which the six most populous account for 40 to 60% of all mutants. 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 redundant ones. *Selective mutation* is an approximation technique that selects only mutants that are truly distinct from other mutants [12, 10]. Recent results have shown that of the 22 mutation operators used by Mothra, only five are sufficient to ensure high quality test data. In experimental trials, selective mutation provides 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.

4.4 Optimistic Mutation

One of the most expensive manual steps of using previous mutations systems is determining which mutants are equivalent. The automatic test data generator within Mothra will generate test data that kills 95 to 99% of the mutants. If a tester is willing to accept less than full mutation coverage, then the equivalent mutants that cannot be automatically detected can be safely ignored. Although this also means that a few killable mutants will not be killed, the test cases that would be required to kill the last few mutants can be expected to add little testing power and can probably be safely left out. This obviates the need for the tester to determine which mutants are equivalent, significantly reducing the amount of work done by the human tester.

4.5 New Mutation Process

Figure 3 presents a new model of the mutation testing process. Initially, Godzilla will be used to generate a set of test cases (perhaps a test that is smaller than ultimately desired) and those test cases will be executed against the original program, and then the mutants. The tester will define 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*), will be removed. This process will be repeated, each time generating test cases to only 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 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. The significant difference between the processes is that the loop in the new process 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. There seems to be little hope of automating this step, although by disposing of ineffective test cases before checking outputs, we significantly reduce the workload of the tester.

5 A Practical and Effective Mutation Analysis System

Using these technological advances and process improvements, the next generation of mutation systems will be faster and more practical, and require significantly less human interaction. Figure 4 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*, a program that incorporates all the mutants of the test program into one program. The schemata generator also produces a *mutant data table*, which is used to store statistics about the mutants such as which are alive and which have been killed. The *test case generator* examines each mutant in the metamutant, and generates test cases to try to kill each mutant. The *driver* subsystem compiles the metamutant, runs each test case on the original program, then on each mutant. The results of running the mutants are saved in the mutant data table, 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 selective-style mutants. 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 will incorporate 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.

6 Conclusions

By combining the orthogonal techniques of schema-based mutation, weak mutation, and selective mutation, this system has the ability to execute mutants at speeds that are orders of magnitude faster than existing research systems such as Mothra. By including automated test data generation and treating equivalent mutants optimistically, this system would require significantly less human involvement, dramatically reducing the cost of using such a tool. Not only that, our

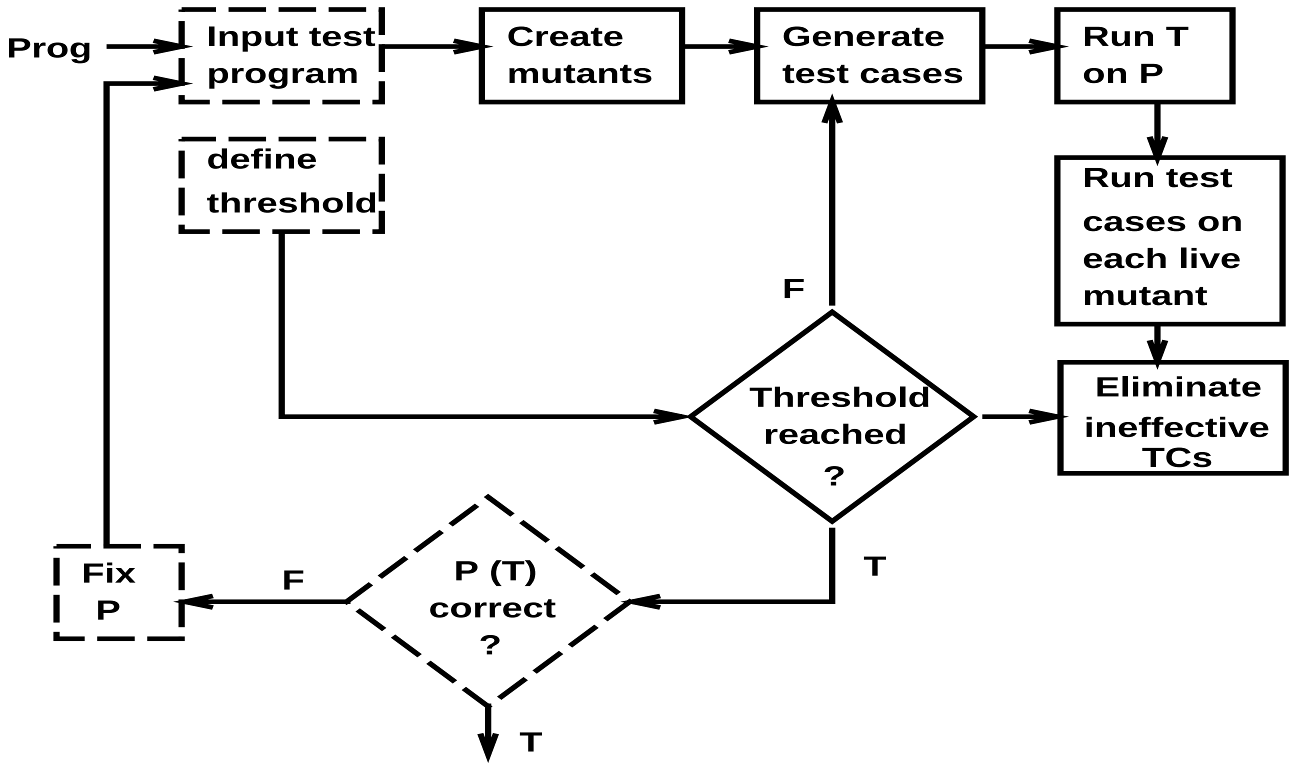

Figure 3: **New Mutation Testing Process.**
Solid boxes represent steps that are automated and dashed boxes represent steps that are manual.

experience has shown that building mutation systems using program schemata instead of interpretation is much easier and faster, which decreases the cost of building a mutation system. Schema-based systems have the additional advantage of being able to use the standard development compiler and runtime environment for executing the mutants, instead of the special-purpose tools included in the mutation system.

We envision a system that provides almost complete automation to the tester. This type of system would allow a programmer to submit a software module, and after a few minutes of computation, respond with a set of test cases that are assured of providing 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.

References

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

[2] 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.

[3] 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.

[4] R. A. DeMillo and A. J. Offutt. Experimental results from an automatic test case generator. *ACM Transactions on Software Engineering Methodology*, 2(2):109–127, April 1993.

[5] W. E. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, 8(4):371–379, July 1982.

[6] K. N. King and A. J. Offutt. A Fortran language system for mutation-based soft-

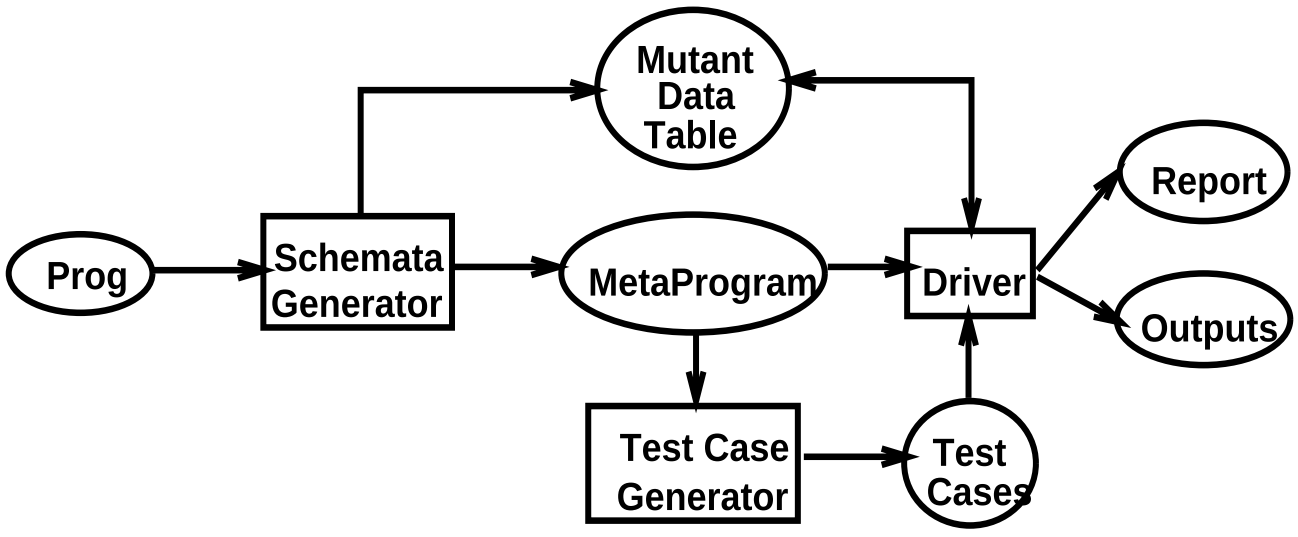

Figure 4: **Architecture of a Practical, Efficient Mutation Testing System**

ware testing. *Software–Practice and Experience*, 21(7):685–718, July 1991.

[7] 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.

[8] L. J. Morell. *A Theory of Error-Based Testing*. PhD thesis, University of Maryland, College Park MD, 1984. Technical Report TR-1395.

[9] A. J. Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering Methodology*, 1(1):3–18, January 1992.

[10] A. J. Offutt, Ammei Lee, Gregg Rothermel, Roland Untch, and Christian Zapf. An experimental determination of sufficient mutation operators. Technical report ISSE-TR-94-100, Department of Information and Software Systems Engineering, George Mason University, Fairfax VA, 1994. Under revision.

[11] A. J. Offutt and S. D. Lee. An empirical evaluation of weak mutation. *IEEE Transactions on Software Engineering*, 20(5):337–344, May 1994.

[12] 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.

[13] R. Untch. Mutation-based software testing using program schemata. In *Proceedings of the 30th ACM Southeast Regional Conference*, Raleigh, NC, April 1992.

[14] 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.