A Mutation Carol: Past, Present and Future

Jeff Offutt

Software Engineering
George Mason University
Fairfax, VA 22030, USA
offutt@gmu.edu

Abstract

Context: The field of mutation analysis has been growing, both in the number of published papers and the number of active researchers. This special issue provides a sampling of recent advances and ideas. But do all the new researchers know where we started?
Objective: To imagine where we are going, we must first know where we are. To understand where we are, we must know where we have been. This paper reviews past mutation analysis research, considers the present, then imagines possible future directions.
Method: A retrospective study of past trends lets us the ability to see the current state of mutation research in a clear context, allowing us to imagine and then create future vectors.
Results: The value of mutation has greatly expanded since the early view of mutation as an expensive way to unit test subroutines. Our understanding of what mutation is and how it can help has become much deeper and broader.
Conclusion: Mutation analysis has been around for 35 years, but we are just now beginning to see its full potential. The papers in this issue and future mutation workshops will eventually allow us to realize this potential.

Keywords: Mutation testing

1. Introduction

The concept of mutation analysis was invented nearly 40 years ago and the first paper was published in 1977. My personal involvement with mutation analysis began as a graduate student in the mid-1980s. I started by reading most of the early papers, then helped write technical sections of a proposal that would

lead to one of the most exciting and dynamic research projects of my career, the Mothra project.

Since those early days, the field of mutation testing has grown remarkably. Jia and Harman [1] documented growth in theoretical work, empirical work, papers published, and tool development over the past 25 years. They found the field to be continuing to expand with more papers published and more scientists working on more projects every year. This paper is based on a keynote talk at Mutation 2009 and offers a retrospective on how the field developed in the past, where it is today, and where it might go in the next decade.

2. Mutation in the Past

Legend has it that the first ideas of mutation analysis were postulated in 1971 in a class term paper by Richard Lipton [2]. Depending on who we ask, his professor, Dave Parnas, either thought mutation was a bad idea or a reasonably clever idea that was not worthy of a PhD dissertation. The first research project was started in the late 1970s by DeMillo (Georgia Tech), Lipton (Princeton) and Sayward (Yale). The first research papers were published by Budd and Sayward [3], Hamlet [4], and DeMillo, Lipton, and Sayward [5]. DeMillo, Lipton, and Sayward's 1978 paper [5] is most often cited as the seminal reference. Mutation has primarily been applied to software by creating mutant versions of the source, but has also been applied to formal software specifications and many other artifacts.

Many fundamental theoretical concepts were laid out in 1980 PhD dissertations by Budd [6] at Yale University and Acree [7] and Hanks [8] at the Georgia Institute of Technology, and in a related technical report [9]. These papers included the original analysis of the number of mutants generated for a program, which Budd found to be roughly proportional to the product of the number of variable references times the number of data objects ($O(Refs * Vars)$). A later analysis [7] claimed that the number of mutants is $O(Lines * Refs)$–assuming that the number of data objects in a program is proportional to the number of lines. This was reduced to $O(Lines * Lines)$ for most programs; the figure that usually appears in the literature.

A 1996 statistical regression analysis of actual programs by Offutt et al. [10] showed that the number of lines did **not** contribute to the number of mutants, but that Budd's figure was accurate. The selective mutation approach eliminates the number of data objects so that the number of mutants is proportional to the number of variable references ($O(Refs)$).

Budd's thesis also established theoretical results on recognizers and generators. A *recognizer* is an algorithm that decides whether a set of tests satisfies mutation (or some other test criterion) and a *generator* is an algorithm that creates a set of tests to satisfy a test criterion. Budd showed that both problems are undecidable, but that there are far more specific instances of the recognizer problem that can be decided than of the generator problem. In practical terms, coverage analysis tools provide approximate solutions to the recognizer problem. They do well except for equivalent mutants, which cannot be determined automatically. The problem of detecting equivalent mutants is a specific instance of the more general feasible path problem [11]. Automatic test data generators provide approximate solutions to the generator problem [12, 13]. Confirming Budd's theoretical results, researchers and tool builders have found this a much harder practical problem to address.

The first mutation research project in the late 1970s and early 1980s focused exclusively on program mutation at the unit level. Mutation operators were developed for Fortran [14, 15], Cobol [16, 17], and Lisp [18]. Several working systems were built, including PIMS (for Fortran) [14, 3], CPMS (for Cobol) [19, 16], and EXPER (for Cobol) [20].

This multi-year, multi-university, project developed many of the fundamental concepts and theory behind program-based mutation analysis. In addition to the above theoretical concepts, the paper by DeMillo, Lipton, and Sayward [5] defined the *coupling effect*, which claims that "simple faults" and "complex faults" are *coupled* in such a way that tests that detect "simple errors" can usually detect "complex errors." The problem of equivalent mutants was also identified, both in theoretical terms (Budd's recognizers) and in practical terms (the tools). The important concept of *program neighborhoods* was also identified by Budd [6]. A *neighborhood* of a program is a set of programs that are very similar to the original program; this is a key theoretical basis for how to define mutation operators. These basic concepts have been explored, refined, evaluated, and updated in dozens of papers through the 1980s and 1990s.

Perhaps most importantly, the working systems built during the original mutation research project established that tools could be built to support mutation. However, these tools were fairly limited in their functionality. They were also not distributed beyond the researchers involved in the initial project, and not used by anybody but the developers.

2.1. The Mothra Project

The Mothra project was established at the Georgia Institute of Technology in the 1980s to demonstrate practical feasibility of mutation[1]. This Air Force-funded project had a goal of solving the many engineering problems associated with using mutation in practice. The resulting tool, Mothra, was the first widely used working mutation system. In addition to the researchers on the Mothra project, Mothra was installed at hundreds of universities and research labs, and used by dozens of researchers in their own projects [21, 22, 23, 24]. Mothra also gained wide popularity as a fault-insertion tool to support experimental comparisons of other test criteria [25, 26]. When requested, the source (over 100 KLOC of C) was provided to other researchers who went on to create numerous modifications that were used to demonstrate research solutions and to support fundamental mutation-related research questions. Some have called Mothra an early precursor to modern open source software.

The Mothra project resulted in dozens of papers during the project, and more after [27, 12, 22, 28, 29, 21, 30, 31, 23, 13, 11, 10, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 24] (this is far from a complete list). Four PhD theses were a direct result of the Mothra project: Offutt (1988) at Georgia Tech [46], and Agrawal (1991), Krauser (1991), Wong (1993) at Purdue University [47, 48, 49]. Follow-on PhD theses after the primary Mothra project finished were by Choi in 1991 and Ghosh in 2000 at Purdue University [50, 51] and Untch at Clemson University in 1995 [52]. More than half a dozen MS theses were also completed after the primary Mothra project finished, including Craft (1989), Seaman (1989), Lee (1991), Pressley (1992), Zapf (1993) at Clemson University and Pan (1994) at George Mason University [53, 54, 55, 56, 57, 58].

The Mothra research project addressed several technical and engineering problems. Some required fundamental theoretical advances, while others required new design, engineering, or programming solutions. One of the most novel aspects of this project was that it was one of the first large-scale software engineering research projects that included the philosophy "We build tools." Most software engineering research at the time was qualitative in nature and few involved building tools that were robust enough to distribute to other researchers. Of course, building tools is now the expected norm in software engineering.

[1] In some early documents, Mothra was given as the acronym *Mutation Oriented Test Harness for Reliable Ada*, but that acronym was very seldom used, partly because Mothra itself tested Fortran programs, not Ada. Rumors that the name Mothra was based on an old Japanese monster movie are greatly exaggerated.

- **Architecture of a mutation system:** An important philosophy behind the architecture of Mothra was to view it as a *laboratory* for future research efforts. Thus, Mothra should contain an infrastructure that was adaptable and expandable. The idea was to make it easy to add new tools, modify existing tools, and inspect and modify all data stores. The primary designers of Mothra were Rich DeMillo (the project PI), and Jeff Offutt (PhD student). This philosophy led us to an architecture of a collection of separate programs, which were integrated around a common set of data stores with standardized APIs. In modern terms, this would be called an object-oriented, component-based system, although those terms were barely known in 1984 when the design was created. Parts of the design were documented in technical reports and early papers [59, 23, 24]. Figure 1 illustrates the overall architecture.

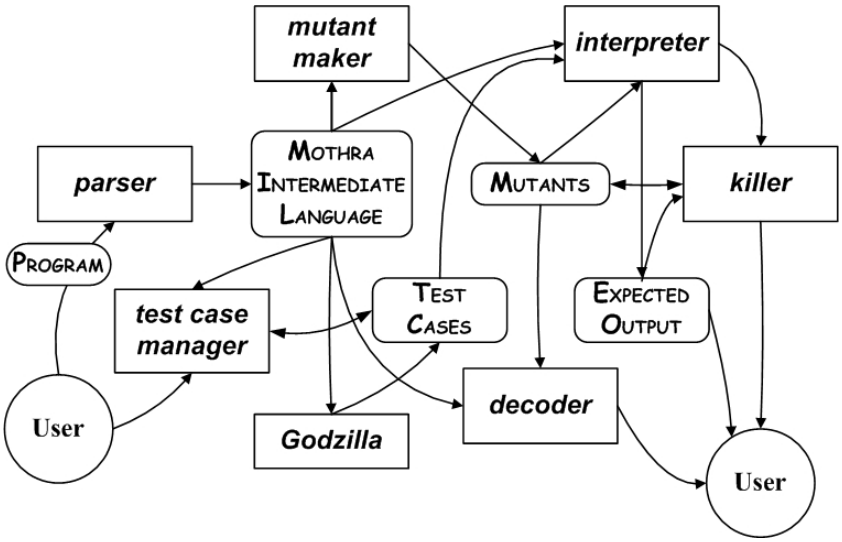

Figure 1: The Mothra Tool Set

The four rounded rectangles in the middle of the figure represent major data stores. The eight squared rectangles represent separate tools. Each tool performed a separate mutation task, and they communicated through the data stores. The *parser* accepted a program from the user and translated it to a special-purpose intermediate language, the **Mothra Intermediate Language** (**MIL**). The **MIL** was at a fairly high level, and included lots of operators specifically designed to support mutation[2]. The *interpreter*

[2]The **MIL** looks fairly similar to Java ByteCode, at least in terms of format and amount of

read the **MIL**, read test inputs from **Test Cases**, ran the program, and saved the results in **Expected Output**. The *mutant maker* read the **MIL**, then applied a number of rules (mutation operators), and saved the results in **Mutants**. The *interpreter* also read the **Mutants** data store, applied operators to modify the **MIL** to create mutants, and passed the results to the *killer*. The *killer* tool read the **Expected Output**, compared it with the actual output of each mutant, and then marked dead mutants in the **Mutants** data store.

The *test case manager* accepted inputs from the user and saved them in a form the *interpreter* could read. The *decoder* read the **MIL** and the **Mutants**, then showed the mutants to the user in source code, human readable, form. The automatic test data generator (*Godzilla*) generated tests automatically and is described below. Other tools were also created as the Mothra project continued, including tools to mark mutants as equivalent by hand, then automatically.

The team designed and built several user interfaces (*UIs*) for Mothra. One of the goals of the tool-based architecture was to allow maximum flexibility in how Mothra was used. In development, testing, and much of the experimentation, the different tools were run directly from the command line (in Unix). Options were built into the tools to support different types of use. For example, the *interpreter* had options to run a single test, a set of tests, or all tests; a single mutant, a set of mutants, or all mutants; to run only live mutants or to run all live, dead, and equivalent mutants; and to display more or less information. The *killer* tool could kill one mutant or a set of mutants; and could make a mutant alive again. The design surrendered efficiency of integration in favor of flexibility of operation. Most of the execution time was spent in the interpreter, so the lost of efficiency elsewhere was relatively unimportant. The interpreter developers (Offutt and Kim King) spent a lot of effort in optimizing the interpreter.

The first non-command line UI was a menu system in the Bourne shell (*sh*), with menu options mapped to tools. More sophisticated interfaces were built in the C-shell (*csh*), the Korn shell (*ksh*), and the Bourne-Again shell (*bash*); the Bourne-Again shell version was most commonly distributed with Mothra. Later, a graphical user interface was built with the X-library. Unfortunately, it depended on local modifications to the X-library that

information kept.

were made at Purdue University, so few were able to successfully install it elsewhere.

- **Creation and storage of mutants:** Mutation has more test requirements (that is, mutants) than most other test criteria, so handling the mutants was a major issue in the design and development of Mothra. The most obvious way to implement mutation is to create a complete copy of the source for every mutant. Of course, this requires lots of space and lots of compilation, costs that were viewed as being prohibitive for Mothra. The solution was based on the **MIL**, which can be described as halfway between assembly language and source code. The **MIL** was designed specifically so that each mutant required a change to one and only one **MIL** statement. Thus, most **Mutants** were stored as simple records that had a replacement **MIL** statement and an index into the **MIL**. A few mutants required a new **MIL** statement to be inserted after the indexed statement. This simple and efficient storage mechanism allowed the *interpreter* to read the entire **Mutants** description file into an in-memory array, then run a mutant by making a single change to the original **MIL**.

  Another early decision was to be able to view the data stores and hand-modify them. Thus, each data store was a flat text file. This slowed down I/O, so the tools were designed to read data into memory before execution and only access the files at the beginning and end of execution.

- **Efficient execution of mutants:** A related issue was how to most efficiently execute large numbers of mutants. The simple design of the **Mutants** file helped a lot. Interpreting the **MIL** was slower than running a compiled program and speed comparisons showed that the *interpreter* was between five and ten times slower. Not having to read and write files when changing mutants or tests saved significant execution time. The *killer* program was adapted to run both as a standalone program that read the **Mutants** file, changed it, then wrote it back out, and as a function call from the *interpreter* to avoid reading and writing the **Mutants** file. In unit-level program-based mutation, the first few tests tend to kill lots of mutants. So the normal execution of the *interpreter* was to run each test on every live mutant, as opposed to running each mutant against each test. The first test ran against all mutants, the second usually ran against only about two thirds of the mutants, and by the fourth or fifth test, Mothra was typically running 20% or fewer of the mutants.

  Another issue with the *interpreter* was that it was running (almost) the

same program dozens or hundreds of times in the **same process**. Thus it was important that state from one run not interfere with subsequent runs. This was done by zeroing out the entire memory array in the interpreter (a technique now implemented much more elegantly by Java's thread mechanism). Another issue that came up is that some mutants would cause an infinite loop. Unix will happily run a process forever or until the user forcibly stops it, but that is not an option when the same process needs to manage hundreds of executions. Mothra's simple solution was the "*10X*" rule–if a mutant executed 10 times as many **MIL** statements as the original program on the same test, the mutant was stopped and marked killed. Some mutants will also lead to run time exceptions (we are, after all, trying to cause failures!), which in Unix C programs usually cause the process to terminate. Thus the *interpreter* had to catch all exceptions, kill the mutant, and continue execution.

- **Definition and storage of test cases:** Mothra was built for unit testing of Fortran functions, which has an advantage of making the definition of a test very simple. A Mothra test was comprised of only primitive type variables in either parameters or global statements. The *test case manager* identified the inputs and gave users a simple interface to add values for each input. If a program had a read statement, during original execution the read statement prompted the users for inputs, then saved them and used them again when executing mutants (a very primitive form of a capture-replay tool).

- **Automatic test data generation:** One of the most difficult and expensive tasks in testing is finding input values to test software with. This was the subject of Offutt's dissertation [46, 12]. The first version of the Godzilla tool simply created random values that were correctly formatted to be stored in the **Test Cases** file. This led to a fascinating result–random values killed from 40% to 50% of the mutants, indicating that roughly half of all mutants are *trivial*, that is, easily killed. Even though mutation generates a lot of test requirements when compared to other criteria, it does **not** require significantly more tests.

  The theory of having to **reach** the mutated statement (*reachability*), **infect** the state with an incorrect value (*necessity*), and then **propagate** to an incorrect output (*sufficiency*) led to a symbolic evaluation approach combined with rules for creating tests to kill individual mutants. Satisfying reachability (the next version of the tool) killed 70% to 75% of the mu-

tants, implying that the statement coverage criterion would kill most mutants. Satisfying infection yielded tests that killed 85% to 90% of the non-equivalent mutants [12, 28, 22, 29, 44, 43, 41]. However, these results only held for program units that did not have loops, arrays, or pointers (generally, programs that did not need to place data objects on the stack). Later research resulted in a dynamic symbolic approach [60, 61] that, combined with domain reduction procedures, killed over 95% of non-equivalent mutants in the presence of loops and arrays, and got near 100% coverage on data flow and branch coverage [13].

- **Most effective mutation operators:** An important question about mutation is what operators to use. Which are redundant with respect to other operators? Which lead to the strongest tests? Mothra was used for numerous experimental studies on individual operators, many of which were never published. The most important were probably the *constrained*, or *selective*, approaches [62, 63, 10, 32], which led to a conclusion that only five of Mothra's 22 operators are needed. Moreover, those five operators yielded mutants that were linear in the number of variable references (as mentioned above).

- **Process and user interface:** Another issue that Mothra helped consider was how best to structure the mutation process. The tool-based architecture allowed any process to be imposed on Mothra, and two important concepts emerged from the automatic test data generation research. First, if test inputs are generated automatically, they become cheap, throwaway, resources. Second, most tests are *ineffective* in the sense that they do not kill new mutants that previous tests did not already kill. This led to the realization that the step of evaluating the results of tests should be saved to the end. That is one of the most human-intensive and thus expensive parts of mutation, thus this process change greatly reduces the cost of using mutation [64].

  Figure 2 shows two different views of a mutation process. Figure 2(A) on the left shows the mutation process as it was usually applied before the Mothra project. The Mothra project led to advances in automation, including test data generation, execution methods, and equivalent mutant detection. The key part of Figure 2(A) is that the manual steps of determining if the program is correct on the tests and analyzing and marking equivalent mutants is **inside** the main loop. Figure 2(B) shows the mutation process after the Mothra project and some of its following research. (The Mothra

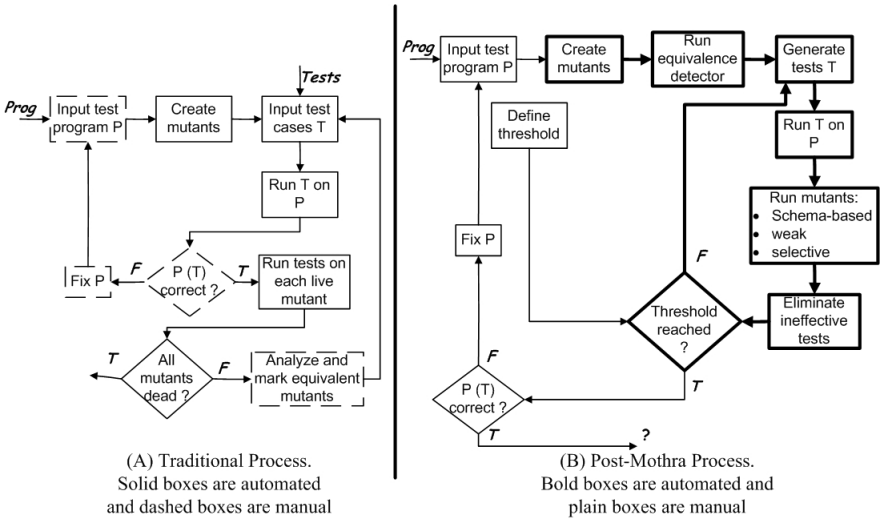

Figure 2: Two Mutation Processes

project officially ended in 1990, and followon work continued throughout the '90s. These figures are from Offutt and Untch's paper from the first Mutation workshop in 2000, and reflects results from Untch's 1995 dissertation, as well as other papers.) Figure 2(B) includes automation for generating tests and determining equivalent mutants as well as advances in mutation execution such as the schema-based approach. Most importantly, figure 2(B) has **no** manual steps inside the main loop. Most tests are thrown away in the "Eliminate ineffective tests" step, and the results of tests' execution on the original program are only considered for tests that are kept. Instead of expecting to find all equivalent mutants and kill all non-equivalent mutants, the tester supplies a practical "threshold" mutation score. When that score is reached, mutation testing should stop. This is a pragmatic engineering process that is not perfect, but that helps testers create good tests.

- **Parallelization of mutation execution:** Another issue that was looked at extensively with Mothra was parallelization of mutant execution. Krauser and Mathur first investigated parallelization with an NCube in 1986 [65]. Offutt, Pargas, Fichter, and Khambekar modified the *interpreter* to run on a hypercube in 1992 [40]. Zapf modified the *interpreter* to distribute across a collection of loosely connected Sun 4 Unix workstations [58]. Choi and Mathur also tried mutation on a hypercube in 1993 [27], and

in Choi's 1999 dissertation with three architectures, a MIMD, a vector processor, and a MIMD with vector processes [50]. Because mutation involves executing hundreds of almost identical programs independently, it is a natural for parallelization, and all studies reported successful results.

- **The coupling hypothesis:** The seminal paper by DeMillo, Lipton, and Sayward [5] introduced the notion of the *coupling effect*. The Mothra system was used to carry out an experimental evaluation of the coupling effect by modifying the *interpreter* to introduce two, then three, mutants at a time [66, 39]. Wah subsequently verified the positive experimental results with analytical proofs [67, 68].

- **Weak mutation:** Howden suggested the concept of *weak mutation* in 1982 [69]. Weak mutation stops execution of a mutant immediately after the mutated software "component," essentially satisfying reachability and infection, but not propagation. Girgis and Woodward evaluated weak mutation by hand in 1985 [70], and Woodward and Halewood suggested a compromise called *firm mutation* in 1988, in which execution is stopped at some point after the mutated software component but well before the end of the program [71]. Horgan and Mathur published theoretical support for weak mutation in 1990 [72]. In separate studies in 1991, Marick [73] and Offutt and Lee [42, 37] built weak mutation systems and evaluated the idea experimentally. Offutt and Lee's system was built by modifying the Mothra interpreter. This building tool effort revealed that evaluating a mutant after a mutated software "component" was ambiguous, and in fact, all weak mutation has to be in some sense firm. They found the most appropriate place to check mutants was after the basic block that enclosed the mutant.

- **Equivalent mutants:** A mutant that always exhibits the same behavior as the original program is *equivalent*. Equivalent mutants cannot be killed so represent noise in the mutation process. They are specific examples of the "feasible path" problem [74, 75], also known as the infeasible test requirement problem. Deciding whether a mutant is equivalent is generally undecidable, so various approximation procedures have been proposed.

  The first suggestion was by Baldwin and Sayward in 1979, who proposed using compiler optimization strategies to determine equivalence [76]. In 1981, Tanaka suggested using data flow analysis [77]. Offutt and Craft investigated Baldwin and Sayward's suggestion by adding an equivalence checker tool to Mothra [53, 36], which found around 15% of equivalent

mutants. This research suggested a stronger technique of using the constraints developed for test data generation to detect equivalent mutants. Subsequently, Pan and Offutt used algebraic techniques to detect infeasible constraints, which represented equivalent mutants [78, 11, 56]. This technique found all equivalent mutants that compiler optimization found, plus more for a total of around 45%. In her thesis, Pan suggested that program slicing could subsume the infeasible constraint method, and probably find more equivalent mutants. This idea was picked up by Hierons, Harman and Danicic, who explored the use of program slicing and program dependence in test data generation and equivalent mutant detection [79].

The Mothra project can be viewed as one of the field's most successful software engineering research projects. While funding only lasted a few years, its influence is still felt twenty years later. It demonstrated the strength and potential of mutation research. Before Mothra, most testing researchers doubted whether mutation had any value at all. After Mothra, mutation was widely recognized as the strongest test criterion known, although many still questioned its practical feasibility. Dozens of papers were a direct result of the Mothra project, and hundreds of papers were written after the project, but based on the product. The Mothra project also established the careers of several scientists who are still active researchers today. A goal of the Mothra system was to create a laboratory that could be used in many ways by many researchers, and by this measure, the product was enormously successful. Perhaps most importantly, it stimulated the field of software testing, engaging many young scientists in many ways. Jia and Harman [1] called Mothra "the most widely studied mutation testing tool." Virtually all current mutation systems can trace their lineage back to Mothra in one way or another.

2.2. Mutation Through the 1990s

Mutation research expanded in several directions during the 1990s. More researchers joined the field, more problems were addressed, and mutation began to be used for problems other than program-based unit testing. The Mothra project identified two problems for practical (industry) adoption of mutation: (1) Mutation analysis was too slow, and (2) Mutation was too hard for testers to use without becoming experts on the theory.

As Untch [52] put it, research into speeding up mutation fell into three categories; (1) Do fewer (that is, fewer mutants), (2) Do smarter, and (3) Do faster. *Do fewer* approaches include selective mutation [80, 26, 10, 32, 81, 62, 63] and mutant sampling [82]. *Do smarter* approaches include weak mutation [83, 70,

69, 73, 84, 42, 37, 71], distributed execution [27, 65, 40, 58], and different processes [64]. *Do smarter* approaches include schema-based analysis [85, 52, 86, 87] and compiler-integration [30, 48]. Most research into making mutation easier to use during the 1990s tried to eliminate manual labor, including automatic test data generation [28, 12, 29, 43, 41, 46, 13, 44] and equivalent mutant detection [76, 53, 79, 36, 78, 11, 56, 77].

Most of this work focused on program-based unit testing. Tools used mutation operators defined for traditional programming languages to modify individual statements, one at a time. Languages for which researchers defined mutation operators included Fortran [23], Ada [88, 89], C [90, 91, 92], and Lisp [18]. This use of mutation can be considered to be "traditional mutation."

One of the first "out of the box" ideas for mutation was interface mutation [93, 94, 95, 96, 81]. Interface mutation mutates function calls, moving the application of mutation beyond the unit testing level to integration testing.

Another higher level application of mutation that started in the 1990s was specification mutation, which mutates formal specifications [97, 98, 99, 100, 101, 102]. Mutation analysis was also applied to concurrent software [103], used in many experimental studies [104, 105, 25, 26, 33, 106, 107, 63], applied to finite state machine testing [108] used to measure reliability [31], finding security vulnerabilities [109], testing network protocols [110], and integration testing between classes [111, 112].

In 2000, I thought mutation research was finished [64] and the only thing left was industry adoption. But I completely missed the significance of these novel, non-unit level, applications of mutation.

3. Mutation in the Present

The 2000s saw an explosion of new results, papers, and tools. muJava [113, 114, 115] incorporates statement-level and class-level mutation operators and has been widely used in research studies since its first release in 2003. Other tools presently in use include Proteum [91, 93, 94] (built in the 1990s), Csaw [116], Certitude [117], Mu Dynamics [118], Jumble [119], PlexTest [120], Heckle [121], and many more [1]. Mutation has been applied to a variety of program levels and issues including interface mutation [94], class testing [122, 112, 113, 111], multi-class testing [123], object-oriented software [113], web applications [124], real-time software [125], and concurrency [126]; and several new languages, including Python, C#, SQL, Lustre, Ruby, SQL, PHP and AspectJ.

Whereas in the past mutation was applied to programming languages, mutation in the present is being applied to other software artifacts and models, such as

XML, statecharts, activity diagrams, input languages, SQL, HTML, and spreadsheet formulas; not to mention problems other than testing such as security, reliability, and complexity measurement.

These new applications have led to a redefinition of mutation based on an *aggregating realization*: Mutation applied to program units is only one instance of a general class of applications. A modern definition of mutation analysis is:

> **Mutation Analysis:** *The use of well defined* **rules** *defined on* **syntactic descriptions** *to make* **systematic changes** *to the* **syntax** *or to* **objects** *developed from the* **syntax**.

In program-based unit testing, the rules are mutation operators, the syntactic description is the grammar of the programming language, and the objects are the methods or functions under test. This can also be described as a *diversity view*, where mutation is one version of *syntax-directed testing*, which finds tests that cover a space defined by a grammar

This generic interpretation of mutation allows it to be applied in many situations. Essentially, any software artifact that can be described as a grammar can be mutated [127].

Another great step forward in mutation analysis was when mutation entered the mainstream through successful commercialization. All respectable software engineering research should have the eventual goal of helping real programmers build better software. In a purely theoretical area of research, having no path to practical use can be acceptable. However, a field that includes the word "engineering" cannot be purely theoretical. Some advances may take years or decades, but the path should be there. At the beginning, some certainly had doubts whether mutation would ever be practical, but the possibility was always clearly there. The advances in the 1990s made practical application possible in the 2000s.

The product *Certitude* by Certess (now sold by SpringSoft) tests integrated circuit designs in VHDL or Verilog by using mutation and includes many of the engineering advances that were invented during the 1990s. A more limited tool is *PlexTest* by ITRegister, which tests C++ using a limited version of mutation. Mutation has also been used in ad-hoc ways by many non-researchers, indicating that it is moving out of the research stage[3]. Perhaps the strongest evidence

[3]My favorite example is by a former student, who hand-crafted mutants to test a large embedded real-time control program, and found over a dozen software faults that had been deployed for years. One had cost the company tens of millions of dollars, and my student was rewarded with a $750,000 bonus. Unfortunately, this company strongly prefers to remain nameless for fear that its customers may think its software "has bugs."

of widespread acceptance of mutation is very 21st century; it now has its own Wikipedia entry[4]. Thus, mutation is truly becoming a story of a long-term effort by numerous researchers around the world leading to great practical success.

4. Mutation in the Future

> **Prediction is difficult – especially about the future.** - *Niels Bohr*

But prediction is a lot of fun! Before any thinker in a computing discipline dares to discuss the future, it is essential to caution ourselves with the world wide web. It did not exist in 1990, yet by 2000 the web had changed the world. In retrospect, once we had fast computers, a mature knowledge of hypertext, and a far-reaching network, wasn't the web inevitable? With that caution, the following discussions are partly predictions and partly wishes. Only time will tell which wishes become truth, and which predictions will be obviated by new changes in our field.

4.1. Mutation Must be Integrated With Development

Over time, I have realized that developers do not want to understand mutation. They just want good tests! In fact, developers do not want to understand testing. They just want to find problems with their software.

I first heard about the *compile-debug-edit* cycle as a student and remember thinking "I don't want to compile or debug or edit, I want to **write** a program and **run** it!" That was naive; programmers will always need to debug. But modern IDEs eliminate the need to explicitly ask for a compilation. After all, programmers do not care about compiling, they just want to run their programs.

Later, as I learned about testing, I wondered why do we separate syntax errors from semantic errors so strongly? IDEs find syntax errors and either correct them or tell the programmer what they are. Then once we compile cleanly, the IDEs give up and make us do the rest of the work by hand!

Figure 3 illustrates this idea. The compiler finds syntax errors and reports them to the developer, who then uses an editor to modify the program. Once the compiler finds no syntax errors, it turns to a test engine. Mutants are created, tests are automatically generated, and the programmer is presented with a list of tests and their results. The tests are demonstrably effective (having achieved a high mutation score), and when the programmer marks test results as being incorrect, those tests are automatically sent to a debugger.

[4] *http://en.wikipedia.org/wiki/Mutation_testing*

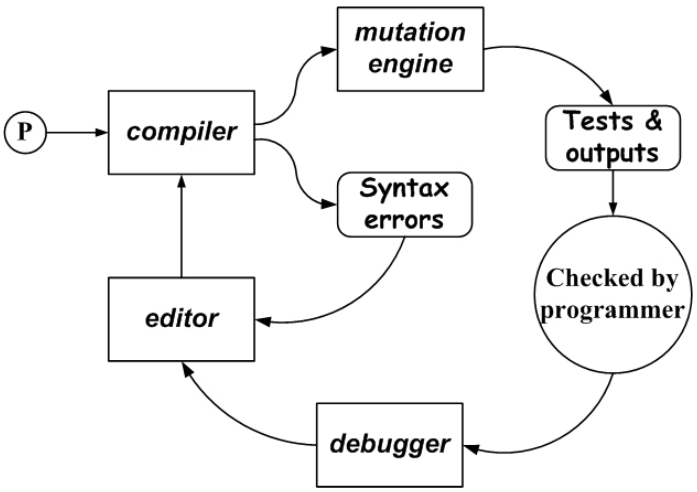

Figure 3: Integrating Mutation into an IDE

This kind of application requires certain relaxations of theoretical concerns. A mutation tool integrated into an IDE must ignore problems of completeness and infeasibility. Primarily, some mutants will not be killed and equivalent mutants should be ignored. While this is uncomfortable for theoretically-minded researchers, the truth is that developers do not care about equivalent mutants or the mutation score. They just want good tests; more precisely, they just want to know when software fails. If the developer has strong concerns about reliability, the tool could provide options to inspect individual mutants and add additional tests or mark them as equivalent. But even ignoring these issues, the tests this tool could generate would be much stronger than the hand-crafted tests the vast majority of programmers currently create during developer testing.

4.2. What Kinds of Faults Can Mutation Detect?

Although we have many theoretical results and a wealth of knowledge for how to use mutation, we have very little knowledge about what specific kinds of faults mutation is good at finding. This requires more experimental studies, preferably on real software with real faults. A related question is what kind of faults do we really care about?

An interesting example can be taken from Java. If a program overrides *equals()*, for the program to behave correctly, it must also override *hashcode()*. Will mutation help us detect that kind of fault? Can we design mutation operators to help detect this kind of fault?

A more esoteric example comes from the features of object-oriented software, specifically, inheritance and polymorphism. Many class-level mutants in Java [115] cannot be killed unless the tester creates a subclass of the mutated class, and the subclass has specific characteristics. How hard is it to create tests to kill these mutants? How useful are these mutants in testing real software?

4.3. Deeper Theory of Mutation Operators

Another likely subject of future exploration is for a deeper theory for mutation operators. The most common goal of mutation operators has been to model software faults. But this is based on a somewhat narrow view of mutation; program-based mutation for functional correctness. What else can be modeled? What kinds of changes or differences **cannot** be modeled by mutants?

Some data indicate that our mutation operators are inefficient. Although small programs tend to create a lot of mutants, experimentalists have found that only a few tests will kill most mutants [128, 129]. This implies that we may not need many of the overlapping mutants. Although experimental studies can help us reduce the number of mutation operators we use [10], a theoretical approach to mutation operator subsumption or dominance could lead to a more comprehensive theory to what mutants are needed. Ultimately, what is a minimalist approach to mutation?

4.4. Mutate for Improvement Instead of Correctness

The advances in mutation allows for many novel applications of mutation. Whereas the 1990s saw us moving from mutating programs to mutating other software artifacts, the generic view of mutation allows us to use mutation analysis for problems other than assessing functional correctness. One novel idea is to mutate an artifact, then measure the new version to see if it is in some way "better" than the original.

Figure 4 illustrates this process. The figure describes the original artifact as a "model of the software," perhaps some sort of design model. The model is then mutated, perhaps with existing mutation operators or possibly with specially-designed operators, to create N mutants. Then the mutants are measured through some sort of "assessment" process. For example, the mutant could be measured as to whether it exhibits improved maintainability, performance, size, or some other qualitative measure. Then tradeoffs among the many mutants could be evaluated, an some mutants may be merged to produce the next version. The process could repeat. This process is similar to traditional biological mutation and shares a lot with genetic algorithms.

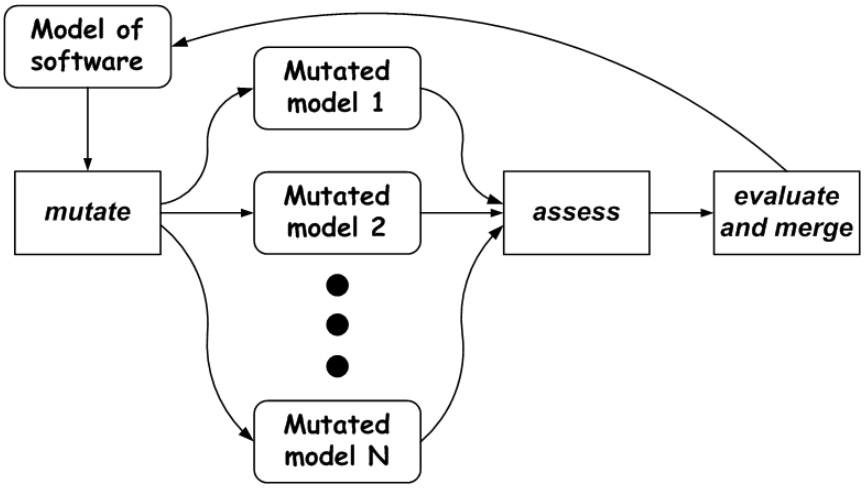

Figure 4: Mutation for Improvement

4.5. What Else Can We Mutate?

The novel generic view of mutation leads to some intriguing possibilities. Grammars can model many software artifacts. What other kinds of artifacts can we mutate? What additional problems can be solved with mutation? How else can we use mutants? I fully expect that this kind of out of the box thinking will lead to many new applications for mutation analysis in the future.

5. Conclusions

The generic definition of mutation in section 3 raises an intriguing possibility. All previous and existing mutation tools have been instantiated for a particular language and set of mutation operators. Even the flexible nature of Mothra did not allow new operators to be added easily. But a more abstract *mutation engine generator* (MEG) might be possible. A MEG would have two inputs: a grammar and a collection of mutation operators. It would need a flexible language for defining operators. The MEG could then create a mutation engine that uses the operators to create mutants on strings of the language defined by the grammar. The MEG could be a tool that allows many mutation analysis tools to be built fairly quickly, just like *yacc* allows compilers to be generated quickly.

6. Acknowledgements

This paper is based on a keynote talk at Mutation 2009. I want to thank the organizers for inviting me, as well as all the attendees for a fascinating workshop.

There is no way to fully acknowledge and thank my PhD advisor, Rich DeMillo. He was a co-inventor of mutation, convinced me to stay in testing by letting me help write the Mothra proposal, then gave me the enormous opportunity to take the technical lead on many aspects of the Mothra project. I also thank Dick Lipton for the initial idea behind mutation and Fred Sayward for co-leading the initial mutation research. I also want to thank all members of the Mothra team: Carolyn Budinger, Byoung-Ju Choi, John Flaspohler, William Hsu, Winny Hsu, Kim King, Ed Krauser, Rhonda Martin, Aditya Mathur, Mike McCracken, Hsin Pan and Gene Spafford (please forgive me if I left anyone out!); and my colleagues and students with whom I have collaborated on with mutation research since: Paul Ammann, Sten Andler, Mike Craft, Scott Fichter, Robert Geist, Fred Harris, Mary Jean Harrold, Zhenyi Jin, Yong-Rae Kwon, Ammei Lee, Stephen Lee, Lisa (Ling) Liu, Yu-Seung Ma, Robert Nilsson, Jie Pan, Roy Pargas, Gregg Rothermel, Kanupriya Tewary, Roland Untch, Jeff Voas, Wuzhi Xu, Christian Zapf, and Tong Zhang. Yue Jia and Mark Harman's excellent survey was a wonderful resource as I wrote this paper. And of course, I am grateful to all researchers in the area of mutation analysis, past, present and future.

References

[1] Y. Jia, M. Harman, An analysis and survey of the development of mutation testing, IEEE Transactions of Software Engineering To appear, dOI: http://doi.ieeecomputersociety.org/10.1109/TSE.2010.62.

[2] R. Lipton, Personal communication (September 2007).

[3] T. Budd, F. Sayward, Users guide to the Pilot mutation system, Technical report 114, Department of Computer Science, Yale University (1977).

[4] R. G. Hamlet, Testing programs with the aid of a compiler, IEEE Transactions on Software Engineering 3 (4) (1977) 279–290.

[5] R. A. DeMillo, R. J. Lipton, F. G. Sayward, Hints on test data selection: Help for the practicing programmer, IEEE Computer 11 (4) (1978) 34–41.

[6] T. A. Budd, Mutation analysis of program test data, Ph.D. thesis, Yale University, New Haven CT (1980).

[7] A. T. Acree, On mutation, Ph.D. thesis, Georgia Institute of Technology, Atlanta GA (1980).

[8] J. M. Hanks, Testing cobol programs by mutation, Ph.D. thesis, Georgia Institute of Technology, Atlanta GA (1980).

[9] A. T. Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton, 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).

[10] J. Offutt, A. Lee, G. Rothermel, R. Untch, C. Zapf, An experimental determination of sufficient mutation operators, ACM Transactions on Software Engineering Methodology 5 (2) (1996) 99–118.

[11] J. Offutt, J. Pan, Detecting equivalent mutants and the feasible path problem, Software Testing, Verification, and Reliability, Wiley 7 (3) (1997) 165–192.
[12] R. A. DeMillo, J. Offutt, Constraint-based automatic test data generation, IEEE Transactions on Software Engineering 17 (9) (1991) 900–910.
[13] J. Offutt, Z. Jin, J. Pan, The dynamic domain reduction approach to test data generation, Software-Practice and Experience 29 (2) (1999) 167–193.
[14] D. M. S. Andre, Pilot mutation system (PIMS) user's manual, Technical report GIT-ICS-79/04, Georgia Institute of Technology (April 1979).
[15] T. A. Budd, R. A. DeMillo, R. J. Lipton, F. G. Sayward, The design of a prototype mutation system for program testing, in: Proceedings NCC, AFIPS Conference Record, 1978, pp. 623–627.
[16] A. T. Acree, CPMS users guide, Technical report GIT-ICS-79/04, Georgia Institute of Technology (April 1979).
[17] J. M. Hanks, Testing COBOL programs by mutation: Volume I-introduction to the CMS.1 system, volume II - CMS.1 system documentation, Technical report GIT-ICS-80/04, Georgia Institute of Technology (February 1980).
[18] T. A. Budd, R. J. Lipton, Proving lisp programs using test data, in: Digest for the Workshop on Software Testing and Test Documentation, IEEE Computer Society Press, Ft. Lauderdale FL, 1978, pp. 374–403.
[19] A. Greece, Document cpms 1.1 in cpms users guide, Technical report GIT-ICS-79/04, Georgia Institute of Technology (April 1979).
[20] T. A. Budd, R. Hess, F. G. Sayward, EXPER implementor's guide, Tech. rep., Department of Computer Science, Yale University (1980).
[21] B.-J. Choi, A. Mathur, R. A. DeMillo, E. Krauser, R. Martin, J. Offutt, G. Spafford, The Mothra tool set, in: Proceedings of the 22nd Hawaii International Conference on System Sciences, Kailua-Kona HI, 1989, pp. 275–284.
[22] R. A. DeMillo, D. S. Guindi, K. N. King, W. M. McCracken, J. Offutt, An extended overview of the Mothra software testing environment, in: Proceedings of the Second Workshop on Software Testing, Verification, and Analysis, IEEE Computer Society Press, Banff, Alberta, 1988, pp. 142–151.
[23] K. N. King, J. Offutt, A Fortran language system for mutation-based software testing, Software-Practice and Experience 21 (7) (1991) 685–718.
[24] J. Offutt, K. N. King, A Fortran 77 interpreter for mutation analysis, in: 1987 Symposium on Interpreters and Interpretive Techniques, ACM SIGPLAN, St. Paul MN, 1987, pp. 177–188.
[25] P. G. Frankl, S. N. Weiss, C. Hu, All-uses versus mutation testing: An experimental comparison of effectiveness, Journal of Systems and Software, Elsevier 38 (3) (1997) 235–253.
[26] E. S. Mresa, L. Bottaci, Efficiency of mutation operators and selective mutation strategies: an empirical study, Software Testing, Verification, and Reliability, Wiley 9 (4) (1999) 205–232, december.
[27] B.-J. Choi, A. P. Mathur, High-performance mutation testing, The Journal of Systems and Software, Elsevier 20 (2) (1993) 135–152.
[28] R. A. DeMillo, J. Offutt, Experimental results of automatically generated adequate test sets, in: Proceedings of the Sixth Annual Pacific Northwest Software Quality Conference, Lawrence and Craig, Portland OR, 1988, pp. 209–232.

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

[30] R. A. DeMillo, E. W. Krauser, A. P. Mathur, Compiler-integrated program mutation, in: Proceedings of the Fifteenth Annual Computer Software and Applications Conference (COMPSAC' 92), Kogakuin University, IEEE Computer Society Press, Tokyo, Japan, 1991.

[31] R. Geist, J. Offutt, F. Harris, Estimation and enhancement of real-time software reliability through mutation analysis, IEEE Transactions on Computers 41 (5) (1992) 550–558, special Issue on Fault-Tolerant Computing.

[32] J. Offutt, G. Rothermel, C. Zapf, An experimental evaluation of selective mutation, in: Proceedings of the Fifteenth International Conference on Software Engineering, IEEE Computer Society Press, Baltimore, MD, 1993, pp. 100–107.

[33] J. Offutt, J. Pan, K. Tewary, T. Zhang, An experimental evaluation of data flow and mutation testing, Software-Practice and Experience 26 (2) (1996) 165–176.

[34] J. Offutt, J. Pan, J. M. Voas, Procedures for reducing the size of coverage-based test sets, in: Twelfth International Conference on Testing Computer Software, Washington, DC, 1995, pp. 111–123.

[35] J. Offutt, Practical mutation testing, in: Twelfth International Conference on Testing Computer Software, Washington, DC, 1995, pp. 99–109.

[36] J. Offutt, W. M. Craft, Using compiler optimization techniques to detect equivalent mutants, Software Testing, Verification, and Reliability, Wiley 4 (3) (1994) 131–154.

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

[38] J. Offutt, A practical system for mutation testing: Help for the common programmer, in: International Test Conference, IEEE Computer Society Press, Washington, DC, 1994, pp. 824–830.

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

[40] J. Offutt, R. Pargas, S. V. Fichter, P. Khambekar, Mutation testing of software using a MIMD computer, in: 1992 International Conference on Parallel Processing, Chicago, Illinois, 1992, pp. II–257–266.

[41] J. Offutt, An integrated automatic test data generation system, Journal of Systems Integration 1 (3) (1991) 391–409.

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

[43] J. Offutt, An integrated system for automatically generating test data, in: Proceedings of the 1990 Conference on Systems Integration, IEEE Computer Society Press, Morristown New Jersey, 1990, pp. 694–701.

[44] J. Offutt, E. J. Seaman, Using symbolic execution to aid automatic test data generation, in: Proceedings of the 1990 Annual Conference on Computer Assurance (COMPASS 90), IEEE Computer Society Press, Gaithersburg MD, 1990.

[45] J. Offutt, Using mutation analysis to test software, in: Proceedings of the Seventh International Conference on Testing Computer Software, ACM SIGSOFT, San Francisco CA, 1990, pp. 65–77.

[46] J. Offutt, Automatic test data generation, Ph.D. thesis, Georgia Institute of Technology,

Atlanta GA, technical report GIT-ICS 88/28 (1988).
[47] H. Agrawal, Towards automatic debugging of computer programs, Ph.D. thesis, Purdue University, (*also* Technical Report SERC-TR-103-P, Software Engineering Research Center, Purdue University, West Lafayette, IN) (August 1991).
[48] E. W. Krauser, Compiler-integrated software testing, Ph.D. thesis, Purdue University, (*also* Technical Report SERC-TR-118-P, Software Engineering Research Center, Purdue University, West Lafayette, IN) (December 1991).
[49] W. E. Wong, On mutation and data flow, Ph.D. thesis, Purdue University, (Also Technical Report SERC-TR-149-P, Software Engineering Research Center, Purdue University, West Lafayette, IN) (December 1993).
[50] B. CHoi, Software testing using high performance computers, Ph.D. thesis, Purdue University (December 1999).
[51] S. Ghosh, Testing component-based distributed applications, Ph.D. thesis, Purdue University, West Lafayette IN (2000).
[52] R. Untch, Schema-based mutation analysis: A new test data adequacy assessment method, Ph.D. thesis, Clemson University, Clemson SC, clemson Department of Computer Science Technical report 95-115 (1995).
[53] W. M. Craft, Detecting equivalent mutants using compiler optimization techniques, Master's thesis, Department of Computer Science, Clemson University, Clemson SC, technical Report 91-128 (1989).
[54] S. Lee, Weak vs. strong: An empirical comparison of mutation variants, Master's thesis, Department of Computer Science, Clemson University, Clemson SC (1991).
[55] D. L. Pressley, The path to godzilla, Master's thesis, Department of Computer Science, Clemson University, Clemson SC, technical Report 92-113 (1992).
[56] J. Pan, Using constraints to detect equivalent mutants, Master's thesis, Department of Information and Software Engineering, George Mason University, Fairfax VA, (Also released as technical report ISSE-TR-94-109, http://www.cs.gmu.edu/~tr_admin/) (1994).
[57] E. J. Seaman, Using symbolic evaluation to address the internal variable problem, Master's thesis, Department of Computer Science, Clemson University, Clemson SC (1989).
[58] C. N. Zapf, Medusamothra-A distributed interpreter for the mothra mutation testing system, M.S. thesis, Clemson University, Clemson, SC (August 1993).
[59] A. H. Agrawal, R. A. DeMillo, W. Hsu, W. Hsu, E. W. Krauser, J. Offutt, H. Pan, G. Spafford, Mothra internal documentation, version 1.5, Technical report SERC-TR, Software Engineering Research Center, Purdue University, West Lafayette IN (July 1989).
[60] B. Korel, Automated software test data generation, IEEE Transactions on Software Engineering 16 (8) (1990) 870–879.
[61] B. Korel, Dynamic method for software test data generation, Software Testing, Verification, and Reliability, Wiley 2 (4) (1992) 203–213.
[62] W. E. Wong, A. P. Mathur, Fault detection effectiveness of mutation and data flow testing, Software Quality Journal 4 (1) (1995) 69–83.
[63] W. E. Wong, A. P. Mathur, Reducing the cost of mutation testing: An empirical study, Journal of Systems and Software, Elsevier 31 (3) (1995) 185–196.
[64] J. Offutt, R. Untch, Mutation 2000: Uniting the orthogonal, in: Proceedings of Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries, San Jose, CA, 2000, pp. 45–55.
[65] E. W. Krauser, A. P. Mathur, Program testing on a massively parallel transputer based sys-

tem, in: Proceedings of the ISMM International Symposium on Mini and Microcomputers and their Applications, Austin TX, 1986, pp. 67–71.
[66] J. Offutt, The coupling effect: Fact or fiction?, in: Proceedings of the Third Symposium on Software Testing, Analysis, and Verification, ACM SIGSOFT 89, Key West Florida, 1989, pp. 131–140.
[67] K. S. H. T. Wah, Fault coupling in finite bijective functions, Software Testing, Verification, and Reliability, Wiley 5 (1) (1995) 3–47.
[68] K. S. H. T. Wah, A theoretical study of fault coupling, Software Testing, Verification, and Reliability, Wiley 10 (1) (2000) 3–46.
[69] W. E. Howden, Weak mutation testing and completeness of test sets, IEEE Transactions on Software Engineering 8 (4) (1982) 371–379.
[70] M. R. Girgis, 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, IEEE Computer Society Press, London UK, 1985, pp. 313–319.
[71] M. R. Woodward, 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, IEEE Computer Society Press, Banff, Alberta, 1988, pp. 152–158.
[72] J. R. Horgan, 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).
[73] B. Marick, The weak mutation hypothesis, in: Proceedings of the Fourth Symposium on Software Testing, Analysis, and Verification, IEEE Computer Society Press, Victoria, British Columbia, Canada, 1991, pp. 190–199.
[74] A. Goldberg, T. C. Wang, D. Zimmerman, Applications of feasible path analysis to program testing, in: Proceedings of the 1994 International Symposium on Software Testing, and Analysis, ACM Press, Seattle WA, 1994, pp. 80–94.
[75] R. Jasper, M. Brennan, K. Williamson, B. Currier, D. Zimmerman, Test data generation and feasible path analysis, in: Proceedings of the 1994 International Symposium on Software Testing, and Analysis, ACM Press, Seattle WA, 1994, pp. 95–107.
[76] D. Baldwin, F. Sayward, Heuristics for determining equivalence of program mutations, Research report 276, Department of Computer Science, Yale University (1979).
[77] 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).
[78] J. Offutt, J. Pan, Detecting equivalent mutants and the feasible path problem, in: Proceedings of the 1996 Annual Conference on Computer Assurance (COMPASS 96), IEEE Computer Society Press, Gaithersburg MD, 1996, pp. 224–236.
[79] R. Hierons, M. Harman, S. Danicic, Using program slicing to assist in the detection of equivalent mutants, Software Testing, Verification, and Reliability, Wiley 9 (4) (1999) 233–262.
[80] E. F. Barbosa, J. C. Maldonado, A. M. R. Vincenzi, Toward the determination of sufficient mutant operators for C, Software Testing, Verification, and Reliability, Wiley 11 (2001) 113–136.
[81] W. E. Wong, M. E. Delamaro, J. C. Maldonado, A. P. Mathur, Constrained mutation in C programs, in: Proceedings of the 8th Brazilian Symposium on Software Engineering,

Curitiba, Brazil, 1994, pp. 439–452.
[82] M. Sahinoglu, E. H. Spafford, A sequential statistical procedure in mutation-based testing, in: Proceedings of the 28th Annual Spring Reliability Seminar, IEEE Computer Society Press, Boston MA, 1990, pp. 127–148.
[83] V. N. Fleyshgakker, S. N. Weiss, Efficient Mutation Analysis: A New Approach, in: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 94), ACM SIGSOFT, ACM Press, Seattle, WA, 1994, pp. 185–195.
[84] A. C. Marshall, Static dataflow-aided weak mutation analysis (SDAWM), Information and Software Technology 32 (1) (1990) 99–104.
[85] R. Untch, Mutation-based software testing using program schemata, in: Proceedings of the 30th ACM Southeast Regional Conference, Raleigh, NC, 1992.
[86] R. Untch, M. J. Harrold, J. Offutt, TUMS: Testing using mutant schemata, in: Proceedings of the 35th Annual ACM Southeast Conference, ACM, Murfreesboro, TN, 1997, pp. 174–181.
[87] R. Untch, J. Offutt, M. J. Harrold, Mutation analysis using program schemata, in: Proceedings of the 1993 International Symposium on Software Testing, and Analysis, Cambridge MA, 1993, pp. 139–148.
[88] J. H. Bowser, Reference manual for Ada mutant operators, Technical report GIT-SERC-88/02, Georgia Institute of Technology (February 1988).
[89] J. Offutt, J. Payne, J. M. Voas, Mutation operators for Ada, Technical report ISSE-TR-96-09, Department of Information and Software Engineering, George Mason University, Fairfax VA, http://www.cs.gmu.edu/~tr_admin/ (March 1996).
[90] H. Agrawal, R. DeMillo, R. Hathaway, W. Hsu, W. Hsu, E. Krauser, R. J. Martin, A. Mathur, G. Spafford, Design of mutant operators for the C programming language, Technical report SERC-TR-41-P, Software Engineering Research Center, Purdue University, West Lafayette IN (March 1989).
[91] M. E. Delamaro, J. C. Maldonado, M. Jino, M. Chaim, Proteum: Uma ferramenta de teste baseada na análise de mutantes (proteum: A testing tool based on mutation analysis), in: 7th Brazilian Symposium on Software Engineering, Rio de Janeiro, Brazil, 1993, pp. 31–33, in Portuguese.
[92] M. E. Delamaro, 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, 1996, pp. 79–95.
[93] M. Delamaro, J. Maldonado, A. Vincenzi, Proteum/IM 2.0: An integrated mutation testing environment, in: Proceedings of Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries, San Jose, CA, 2000.
[94] M. Delamaro, J. C. Maldonado, A. P. Mathur, Interface mutation: An approach for integration testing, IEEE Transactions on Software Engineering 27 (3) (2001) 228–247.
[95] S. Ghosh, A. P. Mathur, Interface mutation, Software Testing, Verification, and Reliability, Wiley 11 (2001) 227–247.
[96] S. Ghosh, A. Mathur, Interface mutation, in: Proceedings of Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries, San Jose, CA, 2000.
[97] P. Ammann, P. E. Black, Abstracting formal specifications to generate software tests via model checking, in: Proceedings of the 18th Digital Avionics Systems Conference (DASC99), 1999.
[98] P. Ammann, P. E. Black, A specification-based coverage metric to evaluate test sets, in:

Proceedings HASE99: 4th IEEE International Symposium on High Assurance Systems, Washington, DC, 1999, pp. 239–248.

[99] P. Ammann, W. Ding, D. Xu, Using a model checker to test safety properties, in: Proceedings ICECCS 2001: Seventh IEEE International Conference on Engineering of Complex Computer Systems, IEEE, 2001.

[100] P. E. Ammann, P. E. Black, W. Majurski, Using model checking to generate tests from specifications, in: Second IEEE International Conference on Formal Engineering Methods (ICFEM'98), Brisbane, Australia, 1998, pp. 46–54.

[101] S. C. P. F. Fabbri, J. C. Maldonado, P. C. Masiero, M. E. Delamaro, E. W. Wong, Mutation analysis applied to validate specifications based on Petri nets, in: Proceedings of the 8th International Conference on Formal Description Techniques (FORTE'95), Quebec, Canada, 1995, pp. 329–337.

[102] S. D. R. S. D. Souza, J. C. Maldonado, S. C. P. F. Fabbri, W. L. D. Souza, Mutation testing applied to estelle specifications, Software Quality Control 8 (4) (1999) 285–301.

[103] R. H. Carver, Mutation-based testing of concurrent programs, in: Proceedings of the IEEE International Test Conference on Designing, Testing, and Diagnostics, Baltimore, Maryland, 1993, pp. 845–853.

[104] M. Daran, P. Thevenod-Fosse, Software error analysis: A real case study involving real faults and mutations, ACM SIGSOFT Software Engineering Notes 21 (3) (1996) 158–177.

[105] R. A. DeMillo, A. P. Mathur, On the use of software artifacts to evaluate the effectiveness of mutation analysis for detecting errors in production software, Technical report SERC-TR-92-P, Software Engineering Research Center, Purdue University, West Lafayette IN (March 1992).

[106] A. P. Mathur, W. E. Wong, An empirical comparison of data flow and mutation-based test adequacy criteria, Software Testing, Verification, and Reliability, Wiley 4 (1) (1994) 9–31.

[107] P. Thévenod-Fosse, H. Waeselynck, Y. Crouzet, An experimental study on software structural testing: Deterministic versus random input generation, in: Fault-Tolerant Computing: The Twenty-First International Symposium, IEEE Computer Society Press, Montreal, Canada, 1991, pp. 410–417.

[108] S. C. P. F. Fabbri, J. C. Maldonado, M. E. Delamaro, P. C. Masiero, Mutation analysis testing for finite state machines, in: 5th IEEE International Symposium on Software Reliability Engineering (ISSRE 94), Monterey, CA, 1994, pp. 220–229.

[109] A. K. Ghosh, T. OConnor, G. McGraw, An automated approach for identifying potential vulnerabilities in software, in: Proceedings of the IEEE Symposium on Security and Privacy, Oakland, California, 1998, pp. 104–114.

[110] R. L. Probert, F. Guo, Mutation testing of protocols: Principles and preliminary experimental results, in: I. Davidson, D. W. Litwack (Eds.), Protocol Test Systems, III, Elsevier Science Publishers B. V. (North-Holland), 1991, pp. 57–76.

[111] H. Yoon, B.-J. Choi, J. O. Jeon, Mutation-based inter-class testing, in: Proceedings of the Asian-Pacific Software Engineering Conference, Taipei, Taiwan, 1998, p. 174.

[112] S.-W. Kim, J. Clark, J. McDermid, Assessing test set adequacy for object-oriented programs using class mutation, in: Proceedings of Symposium on Software Technology (SoST'99), 1999, pp. 72–83.

[113] Y.-S. Ma, J. Offutt, Y.-R. Kwon, MuJava : An automated class mutation system, Software

Testing, Verification, and Reliability, Wiley 15 (2) (2005) 97–133.
[114] J. Offutt, Y.-S. Ma, Y.-R. Kwon, An experimental mutation system for Java, ACM SIGSOFT Software Engineering Notes 29 (5) (2004) 1–4, workshop on Empirical Research in Software Testing.
[115] J. Offutt, Y.-S. Ma, Y.-R. Kwon, The class-level mutants of muJava, in: Workshop on Automation of Software Test (AST 2006), Shanghai, China, 2006, pp. 78–84.
[116] M. Ellims, D. Ince, M. Petre, The Csaw C mutation tool: Initial results, in: Third Workshop on Mutation Analysis (Mutation 2007), Windsor, UK, 2007, pp. 185–192.
[117] M. Hampton, S. Petithomme, Leveraging a commercial mutation analysis tool for research, in: Third Workshop on Mutation Analysis (Mutation 2007), Windsor, UK, 2007, pp. 203–209.
[118] D. Kresse, Mu Dynamics home page, Online, http://www.mudynamics.com/, last access August 2010 (2005).
[119] SourceForge, Jumble home page, Online, http://jumble.sourceforge.net/ (2007).
[120] D. Kresse, PlexTest home page, Online, http://www.itregister.com.au/products/plextest.htm, last access August 2010 (2005).
[121] Rubyforge, Heckle home page, Online, http://seattlerb.rubyforge.org/heckle/ (2007).
[122] P. Chevalley, Applying mutation analysis for object-oriented programs using a reflective approach, in: Proceedings of the 8th Asia-Pacific Software Engineering Conference (APSEC 2001), Macau SAR, China, 2001.
[123] P. R. Mateo, M. P. Usaola, J. Offutt, Mutation at system and functional levels, in: Sixth Workshop on Mutation Analysis (Mutation 2010), Paris, France, 2010.
[124] U. Praphamontripong, J. Offutt, Applying mutation testing to web applications, in: Sixth Workshop on Mutation Analysis (Mutation 2010), Paris, France, 2010.
[125] R. Nilsson, J. Offutt, J. Mellin, Test case generation for mutation-based testing of timeliness, in: Proceedings of the 2nd International Workshop on Model Based Testing, Vienna, Austria, 2006, pp. 102–121.
[126] J. Bradbury, J. Cordy, J. Dingel, Mutation operators for concurrent java (J2SE 5.0), in: Second Workshop on Mutation Analysis (Mutation 2006), Raleigh, NC, 2006.
[127] J. Offutt, P. Ammann, L. L. Liu, Mutation testing `implements` grammar-based testing, in: Second Workshop on Mutation Analysis (Mutation 2006), Raleigh, NC, 2006, pp. 93–102.
[128] N. Li, U. Praphamontripong, J. Offutt, An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage, in: Fifth Workshop on Mutation Analysis (Mutation 2009), Denver CO, 2009.
[129] S. Wang, J. Offutt, Comparison of unit-level automated test generation tools, in: Fifth Workshop on Mutation Analysis (Mutation 2009), Denver CO, 2009.