An Oversight on Mutation Testing

A.Ramya
Research Scholar,
Department of Computer Science,
Sri Ramakrishna College of Arts and Science for Women,
Coimbatore.

S. Preetha,
Assistant Professor,
Department of Computer Science,
Sri Ramakrishna College of Arts and Science for Women,
Coimbatore.

ABSTRACT:

Software Testing is the process of executing a program or system with the aim of finding errors. 50% of the total development time is spent on testing the software and correcting them. Tests are commonly generated from program source code, graphical models of software (such as control flow graphs), and specifications / requirements. Creating test cases that efficiently checks for faults in software is always a problem. To solve this problem, mutation testing, a fault - based testing technique, used to find the effectiveness of test cases. This paper provides an oversight on mutation testing and also discusses various surveys on mutation testing .It also describes the tools ,used to build them effectively and helps in reaching a state of maturity and applicability.

INTRODUCTION:

Software testing is an important phase of software development life cycle. **Software testing** is an investigation conducted to provide end-users with information about the quality of the product or service under test. Software testing can also provide an objective, independent view of the software to allow the business to appreciate and understand the risks of software implementation.

Software testing can be stated as the process of validating and verifying that a computer program/application/product:

- meets the requirements that guided its design and development,
- works as expected,
- can be implemented with the same characteristics,
- and satisfies the needs of end-users

The scope of software testing includes examination of code and execution of that code, in various environments and conditions. It examines the two aspects of code which is:

- ✓ does it do what it is supposed to do
- ✓ Do what it needs to do.

However, for any other program, faults may occur in any development phase of a software .A **Fault** is a structural weakness in a software system that may lead to the systems eventually failing. To eradicate those faults in software system, an efficient test case is needed. The more efficient the test cases are, the more testing can be performed in a given time and therefore the more confidence, can be kept in the software. To solve this problem, mutation testing is introduced. Mutation Testing is a fault-based testing technique, for evaluating, the quality of software.

Mutation testing measures how "good" our tests are, by inserting faults into the program under test.

The rest of the paper is enlisted as follows: operators and tools used for mutation testing,
Background of testing, and summarizes researches made on mutation testing, in several decades.

MUTATION TESTING:

The best way to find a test case, that efficiently works on software faults is, mutation testing.

Seeding defects into a program and checking whether the test suite finds them. **Mutation testing** (or *Mutation analysis* or *Program mutation*) is used to design new software tests and evaluate the quality of existing software tests. Mutation testing involves modifying a program's source code or byte code in small ways. Each mutated version is called a *mutant* and tests detect and reject mutants by causing the behavior of the original version to differ from the mutant. This is called *killing* the mutant.

Test suites are measured by the percentage of mutants that they kill. New tests can be designed to kill additional mutants. Mutants are based on well-defined *mutation operators*, that either imitate typical programming errors (such as using the wrong operator or variable name) ,or force the creation of valuable tests (such as driving each expression to zero). The purpose is to help the tester develop effective tests or locate weaknesses in the test data used for the program or in sections of the code that are seldom or never accessed during execution.

Such defects can be created automatically, using a set of mutation operators to change ("mutate") random program parts. A mutation that is not detected ("killed") by the test suite indicates that the test suite was unable to detect the seeded defect and it is likely to miss similar, true defects in the code. Test managers can use such results, to improve their test suites, such that they detect these mutants.

Mutation testing has been shown to be an effective measurement, for the quality of a test suite and superior to commonplace assessments, such as coverage metrics. A well-known issue is its large usage of computing resources. A less known, but far more significant cost, though, comes from the problem of equivalent mutants. These are mutants that leave the program's overall semantics unchanged and therefore cannot be caught by any test suite: The result of mutation testing "surviving" mutations, not found by the test suite ,thus mixes the most valuable and the least valuable mutations in one set.

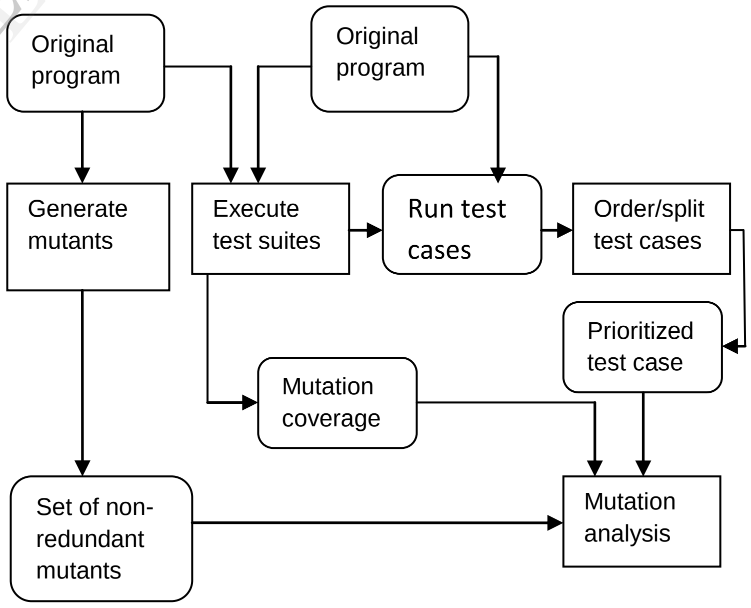


Fig: 1 Mutation testing process.

Various heuristics based on mutant similarity have been suggested. Static program analysis, in particular path constraints, can detect many cases of equivalent behaviour . Program slicing can helps in narrowing down the impact of mutation. Genetic algorithms have been suggested to specifically evolve mutants detected by at least one test case.

One approach to building confidence in test cases is *mutation analysis*, which introduces faults in the software under test. It is assumed that test cases are good, if they detect these faults. This approach, which has been successfully applied to qualify unit test cases, for object-oriented classes, gives programmers useful feedback on the "fault revealing power" of their test cases.

The test cases that testers generally provide easily cover 50–70 percent of the introduced faults, but improving this score to 90–100 percent is time consuming and therefore expensive. So, automating the test optimization process could be extremely helpful.

Mutation testing is done by selecting a set of mutation operators, and then applying them to the source program, one at a time for each applicable piece of the source code. The result of applying one mutation operator to the program is called a *mutant*. . If a test cases distinguish the mutant program from the original program in term of output, then we say mutant are killed, otherwise mutants are alive.

For instance:

Table: 1

Program p	Mutant p
if (a >0 && b > 0)	if (a > 0 || b > 0)
return 1;	return 1;

In this example, program p, is a test case given. Mutant p is a modified form of test case. One mutant operator || is applied to a piece of source program p, instead of &&.

MUTATION OPERATORS:

Mutation Operator	Description
AAR	Array reference for array reference replacement
ABS	Absolute value insertion
ACR	Array reference for constant replacement
AOR	Arithmetic operator replacement
ASR	Array reference for scalar variable replacement
CAR	Constant for array reference replacement
CNR	Comparable array name replacement
CRP	Constant replacement
CSR	Constant for scalar variable replacement
DER	DO statement alterations
DSA	DATA statement alterations
GLR	GOTO label replacement
LCR	Logical connector replacement
ROR	Relational operator replacement
RSR	RETURN statement replacement
SAN	Statement analysis
SAR	Scalar variable for array reference replacement
SCR	Scalar for constant replacement
SDL	Statement deletion
SRC	Source constant replacement
SVR	Scalar variable replacement
UOI	Unary operator insertion

TOOLS:

The development of Mutation Testing tools is an important enabler for the transformation of Mutation Testing from the laboratory into practical and widely used testing technique. Without fully automated mutation tool, Mutation Testing is unlikely to be accepted by industry. In this section, it summaries development work on Mutation Testing tools. These tools are classified into three classes: academic, open sources and industrial.

Name	Application	Year	Character
PIMS	Fortran	1977	General
EXPER	Fortran	1979	General
CMS.1	COBOL	1980	General
FMS.3	Fortran	1981	General
Mothra	Fortran	1987	General
Proteum 1.4	C	1993	Interface Mutation, FNS
TUMS	C	1995	Mutant Schemata Generation
Insure++	C/C++	1998	Source Code Instrumentation

Proteum/IM 2.0	C	2001	Interface Mutation, FNS.
Jester	Java	2001	General (Open Source)
Pester	Python	2001	General (Open Source)
TDS	CORBA IDL	2001	Interface Mutation
Nester	C#	2002	General (Open Source)
JavaMut	Java	2002	General
MuJava	Java	2004	Mutant Schemata, Reflection Technique.
Plextest	C/C++	2005	General (Commercial)
SQLMutation	SQL	2006	General
Certitude	C/C++	2006	General (Commercial)
SESAME	C, Lustre,Pascal	2006	Assembler Injection
ExMAn	C, Java	2006	TXL
MUGAMMA	Java	2006	Remote Monitoring
MuClipse	Java	2007	Weak Mutation, Mutant Schemata, Eclipse plug-in.
CSAW	C	2007	Variable type optimization.
Heckle	Ruby	2007	General (Open Source)
Jumble	Java	2007	General (Open Source)
Testool	Java	2007	General
ESPT	C/C++	2008	Tabular
MUFORMAT	C	2008	Format String Bugs
CREAM	C#	2008	General
MUSIC	SQL(JSP)	2008	Weak Mutation, SQL Vulnerabilities
MILU	C	2008	Higher Order Mutation, Search-based tech, Test harness embedding.
Javalanche	Java	2009	Invariant and Impact analysis.
GAmera	WS-BPEL	2009	Genetic algorithm.
MutateMe	PHP	2009	General (Open Source).
AjMutator	AspectJ	2009	General.

MUTATION-TESTING BACKGROUND:

The main interest of mutation analysis, is to provide an estimate of the quality of a test dataset, with the proportion of faulty programs it detects. To be effective, the mutation analysis, must create mutant programs, which correspond to realistic faults. A test set, is related to the ability of that test set, to differentiate the program being tested, from a set of marginally different, and presumably incorrect, alternate programs. A test case differentiates two programs, if it causes the two programs to produce different outputs.

The process of performing mutation analysis on some test set T, relative to a given program P, begins by running P against every test case in T. If the program computes an incorrect result, the test set has fulfilled its duty and the program must be changed. Assuming P, computes correct results, for every test case in T, a set of alternate programs is produced. Each alternate program, Pi, known as a mutant of

P, is formed by modifying a single statement of P, according to some predefined modification rule. Such modification rules, G, are called mutagenic operators or mutagens.

The syntactic change itself is called the mutation. The original program, plus the mutant programs, are collectively known as the program neighborhood, N, of P. Each mutant is run against the test cases in T. If for some test case in T, a mutant produces a result different than that of the original program, we say that test case has "killed" the mutant indicating that the test case is able to detect the faults, represented by the mutant.

Once killed, these dead mutants are not run against any additional test cases. Some mutants, who are called equivalent mutants, that are syntactically different, are functionally identical to the original program.

An EXAMPLE OF EQUIVALENT MUTATION:

Table : 2

Program *p*	**Equivalent Mutant *m***
for (int i = 0; i < 10; i++)	for (int i = 0; i ! = 10; i++)
{	{
...(the value of i ...	(the value of i
is not changed)	is not changed)
}	}

An equivalent mutant is generated by changing the operator < into operator! = . If the statement within the loop does not change the value of i, program p and mutant m will produce same output.

MSG (P, T) = (#Dead/ (#Mutants-#Equivalent)) * 100%

#Mutants = total number of mutants in the program neighborhood.

The mutation adequacy score MS by the set of mutagens G to reflect their influence on the number and type of mutants produced.

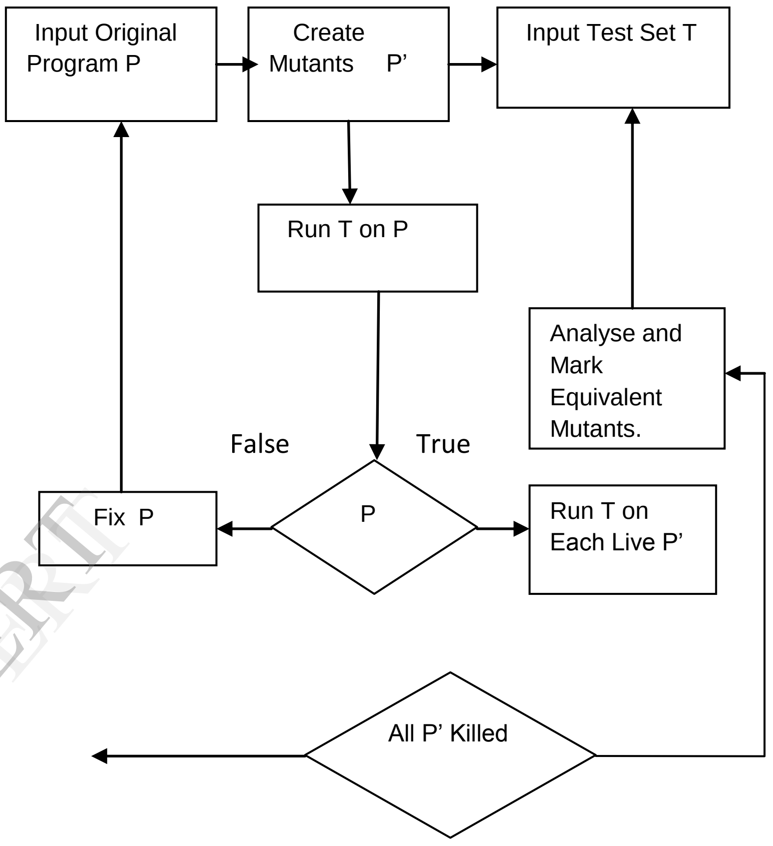
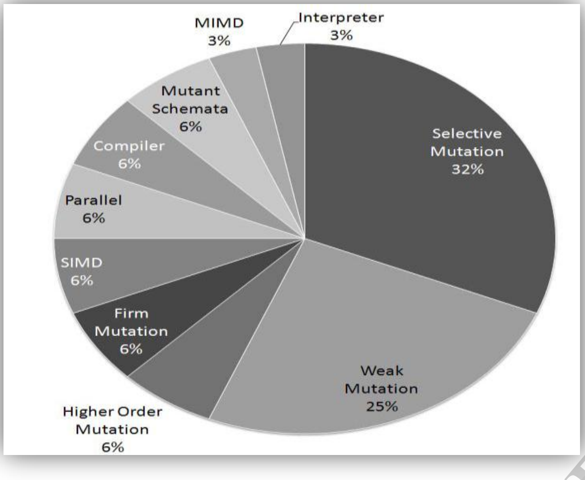


Fig : 2 Process of mutation Analysis.

RESEARCH IN MUTATION TESTING:

Empirical study is an important aspect in the evaluation and dissemination of any technique. Empirical results on the evaluation of Mutation Testing are then reported in detail.



Work in parallel mutation testing has been suggested for vector processors, single-instruction-multiple-data (SIMD) machines, and multiple-instruction-multiple-data (MIMD)machines. Mutant unification was proposed by **Mathur and Krauser** . Their hope is that a vector processor could then execute the unified mutant programs and achieve a significant speedup over a scalar processor .But only only scalar variable replacement (svr) type mutants are suitable for unification. A later paper by **Mathur Krauser,, and Rego** suggests a strategy for efficient execution of mutants on SIMD machines. As in mutant unification, the authors suggest that mutants of the same type be grouped together and that the groups be handled by different processors in the SIMD system. This strategy also has not been implemented.

Choi and Mathur give a general method for scheduling mutant executions on the nodes of a hypercube . In this strategy, each mutant program is separately compiled on the host processor and the resulting executable programs are scheduled for execution on the node processors. The implementation of the strategy, called PMothra, runs on a 128 processor NCUBE/7 hypercube. Unfortunately, because of the cost of separately compiling each mutant program, PMothra actually ran slower than the single processor, interpretive version of Mothra. In their paper, Choi and Mathur suggested removing the compilation bottleneck from PMothra through a method called compiler integrated testing. In this method, the original program is compiled once and the mutant programs are created by making simple code patches to the original executable program. The principle difference between PMothra and HyperMothra is the way that the systems process mutants. In PMothra, each mutant is compiled separately, and the mutant executables are distributed to and executed by the node processors. HyperMothra distributes the MDRs to the node processors, which then apply the changes to the intermediate code and interprets each mutant.

Hamlet presented an early testing system that was embedded in a compiler and performed a version of instrumented weak mutation.Hamlet's system seems to be the first mutation-like testing system.

Girgis andWoodward implemented a system for Fortran-77 programs that integrates weak mutation and data flow analysis. Their system instruments a source program to collect program execution histories. These execution histories are then evaluated to measure the completeness of test data with respect to weak mutation and

several data flow path selection criteria. The system examines the execution history, and if the test case would have caused a mutant to produce an internal program state that differed from the original program's internal state, the mutant is killed.
It suffers from two problems.
First, whether a mutant can be killed can only be obtained for a few kinds of mutants. Second, since no separate executions are being done for the mutants, the components must have a very localized extent, precluding several of the components that we have implemented.These transformations seem to correspond to Mothra's scalar variable replacement (svr), unary operator insertion (uoi), and relational operator replacement (ror) operators.
.

The **Mothra testing project** was initiated in 1986 by members of the Georgia Institute of Technology's Software Engineering Research Center . Mothra is a complete, flexible software test environment that supports mutation based testing of software systems. It was implemented in the C programming language under the Ultrix-32 operating system and has been ported to a variety of BSD and System V UNIX environments. Mothra was designed as a collection of “plugcompatible" tools based on shared data structures that are stored as files and treated as abstract objects. This design has allowed Mothra to evolve to a remarkable degree as a growing group of researchers continues to add new tools and capabilities, implement different user interfaces that allow for novel styles of interaction, and modify the system for special purpose experimentation. At the core of this collection of tools is a set of programs and objects that enable Mothra to translate, execute, and modify programs. They refer to this portion of Mothra as the language system.This paper provides valuable insights for building language systems for special-purpose applications. In particular, some techniques that can be useful in program analysis systems such as debuggers, testing systems, and development environments.

The Mothra software testing environment consists of an extensive tool set. With help and guidance from an advanced user interface, a tester can specify testing goals, automatically generate test cases to satisfy test criteria, execute the program and determine input/output pair correctness or equivalence of mutants, manipulate or fine tune the test cases, and debug the program when errors are revealed. Mothra intermediate code to be simple and efficient. MIC instructions have been used for interpretation, various types of symbolic analysis, data flow analysis [26], decompilation, automatic generation of test data , and are currently being used to develop a debugger . There is much information stored in the MIC instructions and in the Mothra symbol table, yet the information is simple to understand and easy to access. The design and implementation techniques the Mothra team developed to satisfy particular goals and requirements are useful in applications other than mutation analysis and software testing. The way they designed and implemented the Mothra language system has been instrumental in the continuing success of the software. They expected that these techniques to be useful for building largescale program analysis systems, research software, and educational tools.

Woodward and Halewood introduced the idea of firm mutation by pointing out that weak and strong mutation represent extreme ends of what is actually spectrum of mutation approaches. In mutation testing, mutants are killed by comparing the state of the mutant program with the state of the original program on the same test case. Weak and strong mutation differ principally in when they compare the states; strong mutation compares the final outputs of the programs and weak mutation compares the intermediate states after execution of the component. Woodward and Halewood point out that they can compare the states of the two programs at any point between the first execution of the mutated statement and the end of the program, yielding what they called firm mutation..

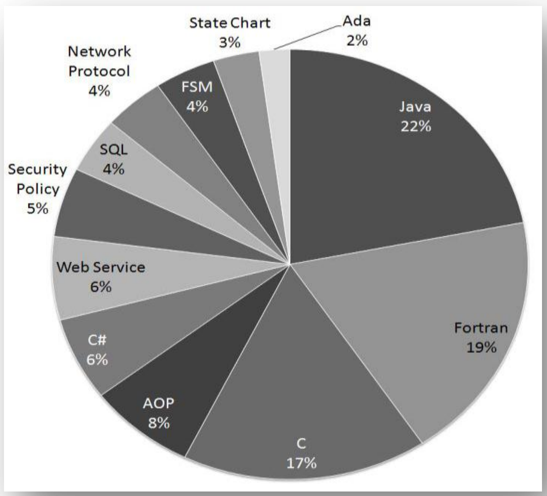
Firm mutation is similar to Morell's concept of extent" in fault-based testing. A local extent technique demonstrates that a fault has a local effect on the computation, and a global extent demonstrates that a fault will cause a program failure. Weak mutation is a local extent technique and strong mutation is a global extent technique. Morell also points out that we could require that the fault affect the program's execution at any point between the local and global extents, depending on how far we require the incorrect program state to propagate.

Richardson and Thompson have used a path analysis approach to extend these ideas to require that a fault transfer from its origination point to some point later in the program's execution.. **Marick** has also implemented a weak mutation system and reported results from using test data generated for weak and strong mutation to find faults that were injected into programs.

OO programs have many characteristics that differ from traditional programs. They are often structured differently and they contain new features such as encapsulation, inheritance, and polymorphism. These differences and new features in OO programs change the requirements for mutation testing. A major difference for testers is that OO software changes the levels at which testing is performed. In OO software, unit and integration level testing can be classified into four levels:
(1) *intra-method*, (2) *Inter-method*, (3) *intra-class*, and (4) *inter-class*.

Kim, Clark and McDermind, and Chevalley and Thevenod-Foss. Offutt developed a categorization of OO programming faults , which is used to design a more comprehensive collection of class mutation operators . There are two goals for the mutation operators, first to address all the OO programming faults and second to ensure all OO language features are tested. The mutation operators tested the language features of inheritance, polymorphism, dynamic binding, and access control. In addition to new mutation operators, some existing operators have been refined for MuJava. The new version contains three new mutation operators for type
conversion, merges two operators from the old version into one, and splits three different operators into two a piece (three operators became six). These changes make the operator definitions and implementations more consistent.
MuJava presents a method for determining equivalent class-level mutants, data on the number of equivalent mutants found, and data on the number of mutants created. The automated Java mutation system MuJava was used to investigate the characteristics of class-level mutants generated from 866 classes drawn from six open-source Java programs. The *equivalency conditions* described in MuJava was found that more than 70% of the class-level mutants were equivalent, far more than the 5% to 15% found with unit-level mutants. Results on the open source software show that there were many fewer class-level mutants than unit-level mutants.

Percentage of publications addressing each language to which Mutation Testing has been applied
Is shown below:



CONCLUSION:

This paper has provided a detailed survey, and analysis of trends, and results on Mutation Testing. The paper covers mutation testing, its background, and empirical evaluations of the mutation testing. Recent trends also include the provision of new open source, and industrial tools. Mutation Testing, is now reaching, a mature state. Mutation has three benefits.

- First, it allows mutation to be described in a more simple way and understood more readily.
- Second, it is easier to develop new applications of mutation analysis. It makes it easier, to apply mutation analysis, to new contexts.
- The third benefit is left for future work, that of ensuring that existing techniques, are complete according to the generic criteria.

Previous researchers have tried to enhance the strength of mutation, by adding more mutation operators, taking the "more is better" philosophy. But the later researchers found that "less is more", or at least, that "less is nearly as good". Previous research, focused on improving the strength of mutation testing, but without a clear metric for strength.

Future mutation testing tools will be developed faster than previous research systems and will require significantly less human involvement.

REFERENCES:

[1] J. Offutt and R. H. Untch, "Mutation 2000: Uniting the Orthogonal," in *Proceedings of the 1st Workshop on Mutation Analysis (MUTATION' 00)*, published in book form, as *Mutation Testing for the New Century*. San Jose, California, 6-7 October 2001, pp. 34-44.

[2] K. N. King and A. J. Offutt, "A Fortran Language System for Mutation-Based Software Testing," *Software: Practice and Experience*, vol. 21, no. 7, pp. 685–718, October 1991

[3] T. A. Budd and D. Angluin, "Two Notions of Correctness and Their Relation to Testing," *Acta Informatica*, vol. 18, no. 1, pp. 31–45, March 1982.

[4] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In ICSE '05: Proceedings of the 27th International Conference on Software Engineering, pages 402–411, New York, NY, USA, 2005. ACM.

[5] P. J. Walsh. A measure of test case completeness (software, engineering).

PhD thesis, State University of New York at Binghamton,Binghamton, NY, USA, 1985

[6] P. G. Frankl, S. N. Weiss, and C. Hu. All-uses versus mutation testing: An experimental comparison of effectiveness. Journal of Systems and Software, 38:235–253, 1997

[7] J. Choi and A. P. Mathur. Use of fifth generation computers for high performance reliable software testing. Technical report SERC-TR-72- P, Software Engineering Research Center, Purdue University, West Lafayette IN, April 1990.

[8] E. W. Krauser, A. P. Mathur, and V. Rego. High performance testing on SIMD machines. In Proceedings of the Second Workshop on Software.

[9] Aditya P. Mathur and E. W. Krauser. Modeling mutation on a vector processor. In Proceedings of the 10th International Conference on Software Engineering, pages 154{161, Singapore, April 1988. IEEE Computer Society Press.

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

[11] M. R. Girgis and M. R. Woodward. An integrated system for program testing using weak mutation and data flow analysis. In Proceedings of the Eighth International Conference on Software Engineering, pages. 313{319, London UK, August 1985. IEEE Computer Society.

[12] R. G. Hamlet. Testing programs with the aid of a compiler. IEEE Transactions on Software.

[13] R. A. DeMillo and E. H. Spafford, `The Mothra software testing environment', Proceedings of the 11th NASA Software Engineering Laboratory Workshop, Goddard Space Center, December 1986.

[14] R. A. DeMillo, E.W. Krauser, R. J. Martin, A. J. Offutt and E. H. Spafford, `The Mothra tool set', Proceedings of the Hawaii International Conference on System Sciences, Kailua-Kona, HI, January 1989.

[15] Offutt, `An extended overview of the Mothra software testing environment', Proceedings of the IEEE Second Workshop on Software Testing, Verification and Analysis, Banff Alberta, July 1988.

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

[17] 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

[18] P. Chevalley and P. Thevenod-Fosse. A mutation analysis tool for Java programs. Journal on Software Tools for Technology Transfer (STTT), pages 1{14, December 2002.

[19] J. Offutt, R. Alexander, Y. Wu, Q. Xiao, and C. Hutchinson. A fault model for subtype inheritance and polymorphism. In Proceedings of the 12th International Symposium on Software Reliability Engineering, pages 84{93, Hong Kong China, November 2001. IEEE Computer Society Press.

[20] Y. S. Ma, Y. R. Kwon, and J. Offutt. Inter-class mutation operators for Java. In 13th International Symposium on Software Reliability Engineering, pages 352{363, Annapolis MD, November 2002. IEEE Computer Society Press.