A survey on mutation testing methods, fault classifications and automatic test cases generation

R Jeevarathinam1* and Antony Selvadoss Thanamani2
1Department of Computer Science, SNR Sons College, Coimbatore 641 006
2Department of Computer Science, NGM College, Pollachi 642 001

Received 13 January 2010; revised 13 January 2011; accepted 17 January 2011

Mutation testing is a fault based testing strategy to measure quality of testing. It measures how good the tests are by inserting faults into the program under test. This review presents a survey on various mutation testing tools available. Basic concepts and notations of mutation testing are described along with working mechanisms.

Keywords: Automatic test cases, Mutation testing, Software testing

Introduction

Mutation testing (MT), though very expensive, is reported1 as an effective measurement for quality of a test suite and superior to common place assessments such as coverage metrics2-3. Surviving mutations, not found by test suite, mixes most valuable and least valuable mutations in one set. Therefore, when one assesses surviving mutants, one must first eliminate equivalent mutants. In an experiment on 20 random mutations on 12,000-line JAXEN program4, 40% of non-detected mutations have been found to be equivalent. On an average, it took 30 min to assess one single mutation for equivalence. If all 4,110 non-detected mutations had been assessed, this task would have cost 2,055 h, or two person-years; actually, it can be assumed that ratio of equivalent mutants increases further as test suite improves. A number of efforts have been attempted to reduce cost of equivalent mutants. Various heuristics based on mutant similarity have been suggested5. Static program analysis can detect many cases of equivalent behavior6 in particular path constraints. Program slicing can assist in narrowing down impact of a mutation7. Genetic algorithms have been suggested to specifically evolve mutants detected by at least one test case8. None of these techniques has yet been shown to scale to large programs.

MT measures quality of testing by examining whether test set, test input data used in testing, can reveal certain type of faults. MT methods generate simple syntactic deviations, mutants of original program representing typical programming errors such as a mutation methods replaces an arithmetic operator says replace a + with -, *, /, which is intended to represent programmer using a wrong operator. If a test case distinguishes mutant program from original program in terms of output then it is said that mutants are killed otherwise mutants are alive. There can be two possibilities in case of live mutant: i) introduced mutants are equivalent mutants and produce same result and cannot be killed; and ii) if test case is unable to differentiate between original symbol and nonequivalent mutant then it can be improved by adding new test cases.

This review presents mutation testing methods for automatic test cases generation.

Mutation Analysis (MA)

MA9 introduces faults in software under test. It is assumed that test cases are good if they detect these faults. MA is a white-box testing technique, in which quality of a test set is related to ability of that test set to differentiate program being tested from a set of marginally different, and presumably incorrect, alternate programs. MAC10,11 gives programmers useful feedback on fault-revealing power of test cases. It also offers an estimate of how many new test cases are needed to better test a given software component. Test cases that testers generally provide easily cover 50-70% of introduced faults, but improving this score to 90-100% is time consuming and therefore expensive. So, automating test
Developments in Mutation Testing (MT)

A test case differentiates two programs if it causes two programs to produce different outputs. Process of performing MA on some test set T, relative to a given program P, begins by running P against every test case in T. If program computes an incorrect result, test set has fulfilled its obligation and program must be changed (determining correctness of these results is “Oracle” problem, which is common to all testing techniques).

Syntactic change is called mutation. 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 rules (G), called mutagenic operators or mutagens. Original program plus 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 original program, it is said that test case has killed mutant, indicating that test case is able to detect faults represented by mutant. Once killed, these dead mutants are not run against any additional test cases. Although syntactically different, some mutants are functionally identical to original program and called equivalent mutants. Although some progress has been made in automatically identifying equivalent mutants, this remains a time-consuming manual task. Since no test case can kill equivalent mutants, which must be removed from consideration in assessing test data quality. Ratio of dead mutants to remaining undifferentiated live mutants is an indicator of test set quality. In MA, mutation adequacy score (MS) measures test set quality and MS is percentage of potentially killable mutants that actually have been killed by T, given as

\[
\text{MSG} (P,T) = \left( \frac{\#\text{Dead}}{\#\text{Mutants} - \#\text{Equivalent}} \right) \times \frac{100}{1}
\]

where \#Mutants is total number of mutants in program neighborhood. Subscript MS by set of mutagens G to reflect their influence on the number and type of mutants produced.

In practice, however, a standard set of mutagens is used and it is common for this subscript to be omitted. Major computational cost of MA is incurred when running the mutant programs against test cases. Number of mutants generated for a program is roughly proportional to the number of data references times the number of data objects.

Research in Mutation Testing (MT)

MT has been suggested for vector processors, single-instruction-multiple-data (SIMD) machines, and multiple-instruction-multiple-data (MIMD) machines. Choi & Krauser gave a general method for scheduling mutant executions on nodes of a hypercube. Krauser et al. suggested that mutants of the same type be grouped together and that groups be handled by different processors in SIMD system. Mathur & Krauser suggested that vectorizable programs be created, each one incorporating several mutants of the same type, and only scalar variable replacement (SVR) type mutants are suitable for unification.

According to Choi & Mathur method, each mutant program is separately compiled on host processor and resulting executable programs are scheduled for execution on node processors. Implementation of strategy (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 single processor, interpretive version of Mothra. It was suggested to remove compilation bottleneck from PMothra through compiler integrated testing, wherein original program is compiled once and mutant programs are created by making simple code patches to original executable program. Main difference between PMothra and HyperMothra is the way how systems process mutants. In PMothra, each mutant is compiled separately, and mutant executables are distributed to and executed by node processors. HyperMothra distributes MDRs to node processors, which then apply changes to intermediate code and interprets each mutant. This method is directly based on Mothra’s interpretive approach, and allows for a more direct comparison of HyperMothra with Mothra than does PMothra. Other differences between PMothra and HyperMothra are that HyperMothra has built-in efficiency improvements over...
Mothra and is implemented on an Intel iPSC/2, a second-
generation hypercube.

Girgis & Woodward\textsuperscript{21} implemented a system for
Fortran-77 programs that integrates weak mutation (WM)
and data flow analysis. Their system instruments a source
program to collect program execution histories, which are
then evaluated to measure completeness of test data
with respect to WM and several data flow path selection
criteria. WM system is analytical rather than execution-
based as Mothra. It examines execution history, and if
test case would have caused a mutant to produce an
internal program state that differed from original
program’s internal state, the mutant is killed. Hamlet\textsuperscript{22}
presented an early testing system that was embedded in
a compiler and performed a version of instrumented WM.
Analytical approach, although computationally less
expensive than execution-based approach, suffers from
two problems: i) whether a mutant can be killed can only
be obtained for a few kinds of mutants; and ii) since no
separate executions are being done for mutants,
components must have a much localized extent,
precluding several components that have been
implemented (including those found to be most effective).
Girgis & Woodward\textsuperscript{21} system also only considers four
of Howden’s five elementary program components
(variable assignment, variable reference, arithmetic and
relational expression), and only applies three types of
mutations (wrong-variable, off-by-a-constant, and wrong-
relational-operator). These transformations seem to
 correspond to Mothra’s SVR, unary operator insertion
(UOI), and relational operator replacement (ROR)
operators.

Mothra\textsuperscript{23} is a complete, flexible software test
environment that supports mutation-based testing of
software systems. It was implemented in C programming
language under 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
‘plug-compatible’\textsuperscript{24} 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. This portion of Mothra is referred as
language system, which must satisfy several unusual
requirements. In particular, some techniques can be useful
in program analysis systems such as debuggers, testing
systems, and development environments.

An extensive tool set is contained in Mothra software
system\textsuperscript{24}. With advanced user interface, a tester can
specify testing goals, automatically generate test cases
to satisfy test criteria, execute program and determine
input/output pair correctness or equivalence of mutants,
manipulate or fine tune test cases, and debug program
when errors are revealed. These capabilities require
extensive support from underlying language system.
Mothra intermediate code (MIC) is simple and efficient.
MIC instructions have been used for interpretation,
various types of symbolic analysis\textsuperscript{25}, data flow analysis\textsuperscript{26},
decompilation, automatic generation of test data\textsuperscript{27}, and
are currently being used to develop a debugger\textsuperscript{24}. There
is much information stored in MIC instructions and in
Mothra symbol table, yet information is simple to
understand and easy to access. This combination of
power and simplicity is important to integrated program
analysis systems and other research software that is
expected to grow over time. As bottlenecks in the testing
process shift in response to new capabilities, additional
tools will be built and integrated into Mothra, allowing it
to serve as an experimental software testing vehicle for
new applications. Design and implementation techniques
that Mothra team developed to satisfy particular goals
and requirements are useful in applications other than
MA and software testing.

Since Leonardo is based on Mothra, it uses all 22
mutant operators. Mothra includes 19 other mutant
operators\textsuperscript{27,28}, which allow for considerably more fault
detection power than systems that use a small subset.
Additionally, Leonardo uses several different definitions
of component, including that extend beyond mutated
statement and thus cannot be implemented analytically.
Woodward & Halewood\textsuperscript{29} introduced the idea of firm
mutation by pointing out that WM and strong mutation
(SM) represent extreme ends of what is actually a
spectrum of mutation approaches. Mutants are killed in
MT, by comparing the state of mutant program with the
state of original program on same test case. WM and
SM differ principally in when they compare states; SM
compares final outputs of programs and WM compares
intermediate states after execution of the component.
Woodward & Halewood\textsuperscript{29} compared states of two
programs at any point between first execution of mutated
statement and the end of program, yielding firm mutation.
A local extent technique demonstrates that a fault has a local effect on computation, and a global extent demonstrates that a fault will cause a program failure. Firm mutation is similar to Morell’s concept of extent in fault-based testing\textsuperscript{25,30}. WM is a local extent technique and SM is a global extent technique. Morell also points out that fault affects program’s execution at any point between local and global extents, depending on how far one requires incorrect program state to propagate. Marick implemented a WM system\textsuperscript{31,32} and reported results from using test data generated for WM and SM to find faults that were injected into programs. Richardson & Thompson\textsuperscript{33} have used a path analysis approach to extend these ideas to require that a fault transfer from its origination point (corresponding to the location of mutation) to some point later in program’s execution. An analytical study of WM\textsuperscript{34} has shown that under certain conditions, test sets generated to satisfy WM can also be expected to also satisfy SM. Constraint-based testing\textsuperscript{35,36} provides indirect support for WM. Test cases that are generated to cause a mutant to have an incorrect intermediate state (essentially satisfying weak mutation) kill their target mutants at a large percentage of the time.

Automatic mutant generator MuJava runs mutants against tests supplied by tester, and reports mutation score of test suite. MuJava can be obtained from two mirrored websites at KAIST and GMU\textsuperscript{37}. Kim et al\textsuperscript{38}, and Chevalley & Thevenod-Fosse\textsuperscript{39,40} first reported class mutation operators. Offutt et al\textsuperscript{41} developed a categorization of OO programming faults, which is used to design a more comprehensive collection of class mutation operators\textsuperscript{42,43}. There are two goals for mutation operators: i) to address all OO programming faults; and ii) to ensure all OO language features are tested. Mutation operators tested language features of inheritance, polymorphism, dynamic binding, and access control, and introduced operators to test type conversion features. In addition to new mutation operators, some existing operators have been refined for MuJava. New version contains three new mutation operators for type conversion, merges two operators from old version into one, and splits three different operators into two a piece (3 operators became 6). These changes make operator definitions and implementations more consistent. Thus, MuJava old version had 24 mutation operators and new version has 29. 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. Automated Java mutation system MuJava was used to investigate characteristics of class-level mutants generated from 866 classes drawn from six open-source Java programs. In MuJava, more than 70\% of class-level mutants were equivalent, and far more than 5-15\% found with unit-level mutants. Results on the open source software show that there were many fewer class-level mutants than unit-level mutants (an average of 46 per class). Mutation testing, which was considered to be too expensive for practical use in unit testing, was found more practical at inter-class testing level.

Conclusions

This paper reviewed application of MT techniques to software testing. Traditionally, MA has been a method for modifying programs according to specific rules to help create high quality tests. Mutation operators can be defined to create alternate versions of mutants, which can be created directly from grammar or by modifying a program. This re-definition of mutation has three benefits: i) it allows mutation to be described in a more simple way and understood more readily; ii) it is easier to develop new applications of MA; and iii) third benefit is left for future work, ensuring that existing techniques are complete according to generic criteria.

References

26 Craft W M, Detecting equivalent mutants using compiler optimization techniques, Master’s Thesis, Department of Computer Science, Clemson University, 1989.


34 Horgan J R & Mathur A P, Weak mutation is probably strong mutation, Technical report SERC-TR-83-P (Software Engineering Research Center, Purdue University, West Lafayette IN) December 1990.


