MiLu : A Customizable, Runtime-Optimized Higher Order Mutation Testing Tool for the Full C Language

Yue Jia
King's College London
Strand, London
WC2R 2LS, UK
yue.jia@kcl.ac.uk

Mark Harman
King's College London
Strand, London
WC2R 2LS, UK
mark.harman@kcl.ac.uk

Abstract

This paper introduces MiLu*, a C mutation testing tool designed for both first order and higher order mutation testing. All previous mutation testing tools apply all possible mutation operators to the program under test. By contrast,* MiLu *allows customization of the set of mutation operators to be applied. To reduce runtime cost,* MiLu *uses a novel 'test harness' technique to embed mutants and their associated test sets into a single-invocation procedure.*

1. Introduction

Mutation testing (or mutation analysis) is a fault based testing technique used to measure the quality of a test set [8, 9]. In mutation testing, for a program p, a set of faulty programs p', called mutants, is generated by small changes to the original program p. A transformation rule that generates a mutant from the original program is known as a mutation operator. Mutants can be classified into two types: First Order Mutants (FOMs) and Higher Order Mutants (HOMs). FOMs are generated by applying mutation operators only once. HOMs are generated by applying mutation operators more than once.

In traditional mutation testing, only FOMs will be adopted. Each FOM p' will be run against a test set T. If the result of running p' is different from the result of running p for any test case in T, then the mutant p' is said to be "killed", otherwise it is said to have "survived". The adequacy level of the test set T can be measured by a mutation score that is computed in terms of the number of mutants killed by T.

In higher order mutation testing we use the full range of mutants, including higher order (where the order is greater than one) and first order (a special case of higher order in which the order is simply one). The HOMs used by MiLu in our approach to higher order mutation testing are called subsuming HOMs; ones that are harder to kill than the FOMs from which they are constructed. The full detail and motivation for higher order mutation testing can be found elsewhere [12].

This paper introduces a mutation testing tool called MiLu. MiLu is specially designed for higher order mutation testing of C Programs. However, though designed to support higher order mutation testing, MiLu is also an efficient and flexible tool for First Order Mutation Testing. MiLu adopts the 77 C mutation operators of Agrawal et al. [2]. Furthermore, it provides customized mutation operators. This paper concentrates on MiLu's approach to efficiency and flexibility. The tool is available from the MiLu website [1].

MiLu (in Chinese characters: 麋鹿) is the name of a deer composed of four other animals. It has a horse's head, a deer's antlers, a donkey's body and a cow's hooves. It is sometimes also known as Père David's Deer (Elaphurus davidianus) [3]. As a result, a MiLu is an example of a HOM; it applies the mutation operators of nature four times. Furthermore, because of its conservation status, MiLus are currently a critically endangered species, so the name also signifies the characteristics of a strongly subsuming HOM; rare but valuable [12].

The rest of the paper is organized as follows. In Section 2, work on previous mutation testing tools are briefly surveyed. Section 3 describes the architecture of MiLu. Section 4 introduces a scripting language to increase the flexibility by allowing users to customize mutation operators used by MiLu. Section 5 discusses the optimization techniques used by other mutation testing tools and introduces the 'test harness' approach that MiLu uses to optimize performance. The paper concludes with Section 6.

2. Background

Since mutation testing was proposed in 1987, a number of mutation testing tools have developed in academic world. Mothra is one of the most widely known mutation testing systems for FORTRAN 77 in the early historical development of mutation testing [6]. It was the first tool that implemented mutation analysis as a complete software testing environment. Following the introduction of mutation operators for the programming language C, proposed in 1989 [2], the first mutation testing tool for ANSI C program was developed, called Proteum [7]. From 1995, as Java became more popular, many mutation tools for Java were developed. One of most widely known one is MuJava, which not only supports traditional mutation operators but also provides class-level mutation operators [15].

In recent years, a number of open source mutation tools have also been implemented for many programming languages. For example, PesTer [16] is a mutation testing tool for Python and PyUnit tests. Nester [17] is a mutation testing tool for C# code. SQLMutation [20] is a mutation testing tool for database queries.

In the industrial setting, several companies have incorporated the idea of mutation testing into their commercial products. Plextest [11] is an automated mutation testing system for C++; one that implements the original idea of mutation with a unit testing framework. Insure++ [19] from Parasoft applies the idea of equivalent mutants to detect errors in C and C++ programs. Certitude [5] from Certess has taken the idea from mutation testing to develop a functional qualification; one that provides the ability to measure the quality of a verification environment.

Although both the academic and the industrial world have embraced mutation testing and have implemented mutation testing tools, the objective of these two worlds are different. The tools from academia often implement new ideas, or adapt new techniques, but tend to be less reliable. By contrast, the industrial tools are more stable, but do not provide the flexibility needed for experimentation. MILU is an academic tool for higher order mutation testing that also aims to provide both a reliable and a flexible environment for traditional mutation testing.

3. MILU Architecture

MILU provides two modes for mutation testing: traditional mode and higher order mode. Traditional mode is designed to support first order mutation testing. In this mode, users are able to use either pre-defined mutation operators or their own customized mutation operators. To automate the testing process, the user also needs to specify a comparison method, distinguishing the results between the mutants and the original program. MILU will take care of rest of the work; it will generate the mutants, execute each of them with the given test set and report the mutation score and other information that may be of use to an experimenter.

By contrast, higher order mode is designed to help researchers study subsuming HOMs. In higher order mode, users can either choose a pre-defined search-based optimization algorithm [10] or specify their own algorithm and fitness function to search for subsuming HOMs. The subsuming HOMs so-found can be applied in traditional mode to substitute for FOMs. MILU provides a user friendly GUI interface for users running this two modes, and it also provides a set of APIs for researches programming their own plug-in for the generation and evaluation of the mutants.

The structure of MILU is shown in Figure 1. The top level of MILU consists of three components: a Source Code Analyzer (SCA), a Mutating System (MTS), and a Testing and Evaluation System (TES). Each of these components also provides a utility interface to allow the users extending the features.

The SCA component works as a C parser. It takes a C program as input, and parses the source into a token list first. In order to support subsequent customization of mutation operators, not only syntactic, but contextual information will be preserved. Therefore, a simple Abstract Syntax Tree (AST) is also constructed from the token list by SCA.

MTS is the core component of MILU. It takes the AST from SCA and the mutation operators that user specified as input and generates a 'mutant template'; one that contains the possible position and type of all the FOMs to the program source. In MILU, each mutant is represented as a vector of integers called the 'Mutation Id'. The 'mutant template' can be used to generate the 'Mutant Id' of all the FOMs.

In traditional mode, the FOMs generated by SCA will be passed to TES directly. However, in higher order mode, MTS will use these FOMs as initial population to generate subsuming HOMs using the search-based algorithm that user specified. These so-generated subsuming HOMs will replace some of the FOMs first, then they will be passed to TES with rest of the FOMs.

TES is mainly used to run the mutants against a test set. It takes each mutant from MTS and passes it to the GCC compiler. Instead of compiling it into a executable program, the mutant is compiled into a shared library. In order to optimize the running cost, a 'test harness' is generated by the input test set. This 'test harness' is able to run all the test cases by invoking the mutant shared library dynamically.

In traditional mode, TES will execute the test harness to generate a detailed mutation report summarizing all running information during the testing process. Additionally, in higher order mode, for each generated HOM, TES will analyze the FOMs from which each HOM is constructed, and compute the fitness according the selected fitness function. A detailed report for these HOMs will be generated.

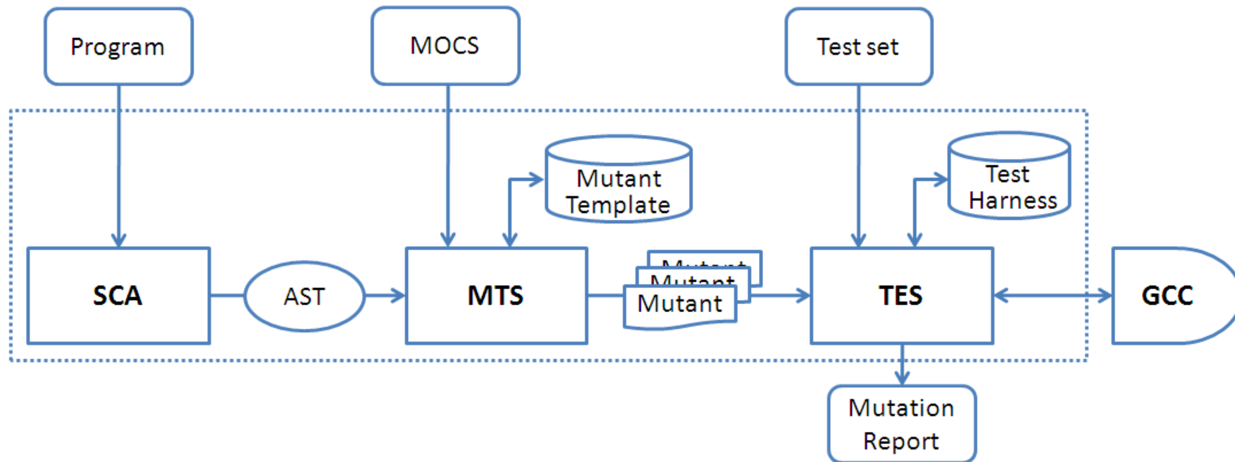

Figure 1. The Architecture of MiLu

4. Flexibility

Mutation operators are a set of transformation rules used to generate mutants. According to the structure and purposes of each program and each programming language, each of them has their own set of rules that represent the faults that programmers often make. However, in order to experiment with mutation testing, it is often convenient to select only a subset of operators to apply in order to explore the effect of this set in isolation from the large body of all operators [18]. Also, in the practical application of mutation testing, it may be convenient to select a set of mutation operators that denote a particular fault model [4]. Therefore, MiLu offers the user the opportunity to use a scripting language, the Mutation Operator Constraint Script (MOCS), to define the mutation operators to be applied. MOCS is able to add constraints to any mutation operators. It provides two types of constraints: direct substitutions and environmental conditions.

Direct Substitution Constraint: allows users to select a specific transformation rule that performs a simple replacement. For example, in the C programming language, programmers often misuse $=$ as $==$. Applying traditional mutation operators would also involve applying all rational operators, for example $>, <, ...,$ etc. However, by using MOCS, the rules can simply specify a single operator:

"==" $->$ "="

Although the idea of this substitution constraint is simple, it is highly flexible especially when the requirements are beyond the ability of the predetermined mutation operators. For example, Jones' research [13] is only interested in the impact of programmers who wrongly change the precedence of arithmetic operators by adding parentheses in the wrong place. Three different examples of scripts that capture different levels of choice with regard to this kind of specialization are:

1. "VAR AOP VAR" $->$ "(VAR AOP VAR)"
2. "VAR + VAR" $->$ "(VAR + VAR)"
3. "INT_VAR + INT_VAR" $->$ "(INT_VAR + INT_VAR)"

The first script will add parentheses to any expression with two variables and a arithmetic operator. The second script only affects the expression with $+$ operator while the third script only affects the expression with two integer variables and $+$ operator. None of these are 'standard' mutation operators, but MOCS makes it easy to add them.

Environmental Condition Constraint: sometimes, instead of applying a mutation operator to all source code, only a subset of the statements is interesting. The environmental condition constraint is used to specify a domain for applying mutation operator. For example, if we are only interested in applying rational operators in `if` statements, we can achieve this as follow:

[IF] ORRE

Where ORRE is the mutation operator for rational operator replacement. Environment condition constraints can also be used with direct substitution constraints together to provide more detailed control. For example, to check only whether the $\&\&$ is mistyped as $\&$ in an if predicate can be specified by

[IF] "&" $->$ "&&"

In order to let the users specify the constraints, SCA does not only parses the program into tokens at a syntactic level, but also builds a simple Abstract Syntax Tree (AST) to preserve structural contextual information. For example, the type of each variable and statement. However none of the expressions are evaluated during the analysis process.

5. Efficiency

From its inception, mutation testing has been considered expensive and so much engineering effort have concentrated upon efficiency. The interpreter-based technique is one of the optimization techniques used in first generation of mutation testing tools [14]. This technique implements a interpreter to interpret the result of a mutant from its source code directly, as shown in Figure 2. The total run time cost therefore is $it \times n \times m$. Where it is the interpreting time, m and n are the number of mutants and test cases respectively. This technique is efficient for small programs. However, due to the nature of interpretation, it becomes slower as the scale of programs increases.

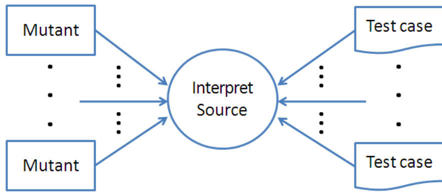

Figure 2. Interpreter-based Technique

To reduce the cost of interpretation, compiler-based techniques were subsequently introduced [7]. In compiler-based techniques, the mutant source is compiled into an executable program first, then each of them will be executed by a number of test cases, as shown in Figure 3. The total run time cost is $m(ct + rt \times n)$, where ct is the compilation time and rt is run time for each test set. This technique is faster than the interpreter-based, technique especially for large programs, because the run time, rt, for an executable program, is faster than the interpretation time it. However, this technique also introduces an extra overhead cost ct.

The Mutant schemata approach is designed to reduce the overhead cost of the traditional compiler-based techniques [21]. Instead of compiling each mutant separately, the mutant schemata technique generates a metaprogram. Just like a "super mutant" this metaprogram can be used to represent all possible mutants, as shown in Figure 4. Therefore, to run each mutant against the test set, only this metaprogram need be compiled. As this metaprogram is a compiled program, its running speed is faster than the interpreter-based technique. The total run time is $ct' + rt' \times n \times m$ where ct' and rt' are the compilation and run time of the metaprogram respectively. As the metaprogram is often a very large program, both ct' and rt' are slightly slower than compiling and running each mutant separately.

Most of the techniques introduced above focus on their optimization effort on reducing the compilation cost (left hand side of the figures). The reason is that in "20th Century mutation testing", this property is considered to be the main cost of the total cost of mutation testing. However, in "21st Century mutation testing tools", the bottleneck has migrated from compilation time to running cost, as explained in Figure 5. Therefore, MILU focuses on optimizing the right hand side of the figures; reducing the running cost.

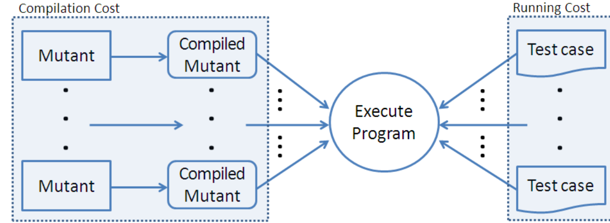

Figure 3. Compiler-based Technique

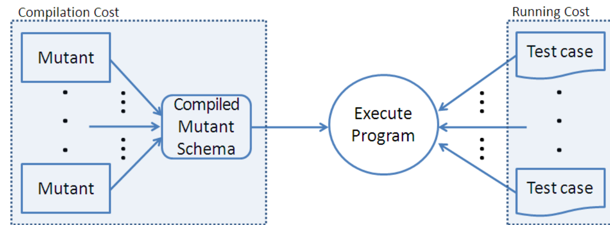

Figure 4. Mutant Schemata Technique

As the run time for a compiled mutant is already very fast, it is hard to improve the efficiency by reducing the single run time. Therefore, MILU reduces the running cost by reducing the totle execution times, as there are a number of overhead costs for running each mutant as a individual program. In MILU, a test harness is created containing all test cases and the settings for running each mutant as shown in Figure 6. Each mutant is compiled into a shared library that can be dynamically invoked by a test harness. During the testing process, only the test harness need to be executed, and each mutant runs as an internal function call. Compare this approach to the more expensive option of running each mutant on each test case separately which needs an operating system call to create a new process for each test case. Each such call will involve the allocation of many resources.

6. Conclusion

This paper has introduced a mutation testing tool called MILU for C programs. To satisfy the different purposes from both academia and industry, MILU not only implements a full set of C mutation operators, but also provides MOCS; a flexible scripting language to allow user-customization of mutation operators. To reduce the total cost of mutation testing, MILU introduces a novel 'test harness' technique to invoke the mutant efficiently as a shared library.

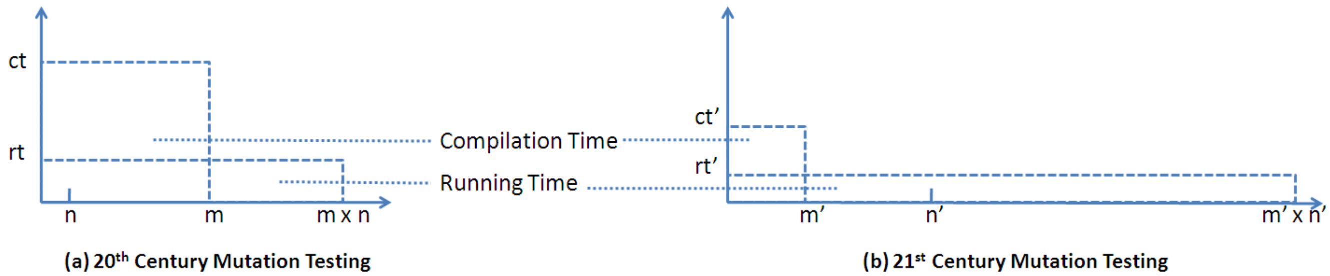

Figure 5. Changing Trends in Compilation and Running Cost for Mutation Testing. In the "20th Century", the number of mutants used in mutation testing were numerous. "21st Century mutation testing tools" use advanced mutation testing techniques, for example selective mutation [18] and mutant sampling [22], so the number of mutants to be considered has fallen from m to m'. However, more demanding testing means that the number of test cases has increased from n to n'. Furthermore, due to increasing computational power, both compilation and run time have reduced from ct and rt to cp' and rt' respectively. These changes in performance cost have tended to shift the dominant cost from complication to execution.

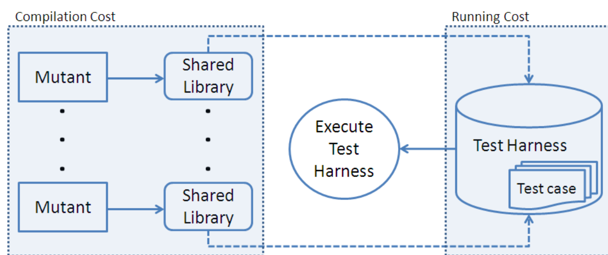

Figure 6. MILU**'s Test Harness Technique**

References

[1] Milu website. www.dcs.kcl.ac.uk/pg/jiayue/milu.

[2] H. Agrawal, R. Demillo, R. Hathaway, W. Hsu, W. Hsu, E. Krauser, R. J. Martin, A. Mathur, and E. Spafford. Design of mutant operators for the C programming language. Technical report, Mar. 1989.

[3] Brinklow. Gestation periods in the Pere David's Deer (Elaphurus davidianus): evidence for embryonic diapause or delayed development. *Reproduction, Fertility and Development*, 5:567–575, 1993.

[4] K. M. Butler. Stuck-at fault: a fault model for the next millennium. In *IEEE International Test Conference*, page 1166, 1997.

[5] Certess. Certitude delivering functional qualification. http://www.certess. com/product/, 2008.

[6] B. J. Choi, R. A. DeMillo, E. W. Krauser, R. J. Martin, A. P. Mathur, A. J. Offutt, H. Pan, and E. H. Spafford. The Mothra tool set (software testing). In *System Sciences*, volume 2, pages 275–284, Jan. 1989.

[7] M. E. Delamaro and J. C. Maldonado. Proteum - A tool for the assessment of test adequacy for C programs: User's guide. Technical report, Jan. 20 1996.

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

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

[10] M. Harman. The current state and future of search based software engineering. In *International Conference on Software Engineering, Future of Software Engineering*, pages 342–357, 2007.

[11] ITRegister. Plextest. http://www.itregister.com.au/products /plextest.htm, 2008.

[12] Y. Jia and M. Harman. Milu: A customizable, runtime-optimized higher order mutation testing tool for the full c language. In *8th IEEE International Working Conference on Source Code Analysis and Manipulation*, Beijing, China, September 2008 To appear.

[13] D. M. Jones. Developer beliefs about binary operator precedence. In *ACCU: C user group(UK)*, 2006.

[14] K. N. King and A. J. Offutt. A Fortran language system for mutation-based software testing. *Software—Practice and Experience*, 21(7):685–718, July 1991.

[15] Y.-S. Ma, J. Offutt, and Y. R. Kwon. Mujava: an automated class mutation system. *Software Testing, Verification & Reliability*, 15(2):97–133, 2005.

[16] I. Moore. Jester-the junit test tester. http://jester.sourceforge.net/, 2008.

[17] I. Moore. Nester: What is this? http://nester.sourceforge.-net/, 2008.

[18] A. J. Offutt, G. Rothermel, and C. Zapf. An experimental evaluation of selective mutation. In *International Conference on Software Engineering*, pages 100–107, 1993.

[19] Parasoft. Insure++. http://www.parasoft.com/jsp/products/-home.jsp?product=Insure, 2008.

[20] J. Tuya, M. J. Suarez-Cabal, and C. de la Riva. Sqlmutation: A tool to generate mutants of SQL database queries. In *MUTATION '06: Proceedings of the Second Workshop on Mutation Analysis*, page 1, Washington, DC, USA, 2006. IEEE Computer Society.

[21] R. H. Untch, A. J. Offutt, and M. J. Harrold. Mutation analysis using mutant schemata. In *International Symposium on Software Testing and Analysis*, pages 139–148, 1993.

[22] W. E. Wong and A. P. Mathur. Reducing the cost of mutation testing: An empirical study. *Journal of Systems and Software*, 31(3):185–196, 1995.