Extension of Mutation Testing for the Requirements and Design Faults

Boyuan Li, Carol Smidts

Department of Mechanical and Aerospace Engineering, The Ohio State University

Scott Laboratory W382, 201 W. 19th Ave. Columbus, OH 43210

li.4935@ osu.edu; smidts.1@osu.edu

ABSTRACT

Mutation Testing is a fault–based software testing technique that is considered to be the most efficient in detecting faults. It focuses on the software code level and follows a set of rules which are called mutation operators to systematically seed faults into the source code. Each seeded fault results is a new version of the software, which is called a mutant. Mutation testing promises to be effective in identifying adequate test data which can be used to find faults in the source code. However, traditional mutation testing is not geared towards the detection of requirements and design faults. To cover the full spectrum of possible faults, the mutation framework has been extended from the code level to the requirements and design level. Twenty-nine defects that appear in requirements and design documents were identified to define appropriate mutants. Then corresponding mutation operators that model the defects were developed. For each mutation operator, the strategy to generate a mutant and the evaluation of the number of possible mutants were defined. Existing code level cost reduction techniques were tailored for requirements and design mutation operators that result in a large number of mutants. With the extended mutation testing framework, the full spectrum of possible faults will be tested and identified to ensure the safety and functionality of the software.

Key words: Mutation Testing, Requirements Faults, Design Faults.

1. Introduction

Mutation Testing is a fault–based software testing technique that has been widely studied for about four decades [1]. The purpose of mutation testing is to generate an effective test set, which can detect faults, by repeatedly improving the test cases.

Two basic hypotheses underlie mutation testing [2]. The first is the competent programmer hypothesis. This hypothesis states that most software faults introduced by experienced programmers are due to small syntactic errors. The second hypothesis is called the coupling effect. The coupling effect asserts that simple faults can cascade or couple to form other emergent faults. Mutation Testing is considered to be the most efficient among existing software testing strategies in detecting faults. It follows a set of rules which are called mutation operators to systematically seed faults into the source code. Each seeded fault results in a new version of the software, which is called a mutant. Test suites are then developed to distinguish the mutants from the original program. The test suites are considered to be capable of finding indigenous faults if the test suite can find all the seeded faults. Mutation testing research focuses on reducing its high cost which consists of the high computational cost of executing the large number of mutants against a test set and the amount of human effort involved in using Mutation Testing.

The concept of Mutation Testing was proposed in 1971 by Richard Lipton [3]. The birth of the field can be traced back to the late 1970s by DeMillo et al. [4] and Hamlet [2]. To reduce the high cost of mutation testing, various cost reduction techniques have been developed including: Mutant Sampling [5], [6], Mutant Clustering [7], Selective Mutation [8]–[12], Higher Order Mutation [13], Weak Mutation [14], [15], Strong Mutation [4], Firm Mutation [16], Run-time Optimization [17]–[19], etc.

Traditional mutation testing focuses on the software code level and promises to be effective in identifying adequate test data which can be used to find all faults in the source code. However, traditional mutation testing does not address requirements and design faults. To cover the full spectrum of possible faults, it is necessary to extend the mutation framework from the code level to the requirements and design level.

In this paper, traditional mutation testing technique is extended to cover requirements and design faults. Twenty-nine defects that appear in requirements and design documents were identified to define appropriate mutants. Then corresponding mutation operators that model the defects were developed. For each mutation operator, the strategy to generate a mutant and the evaluation of the number of possible mutants were defined. For mutation operators that have a large number of mutants, existing code level cost reduction techniques were extended and applied. With the extended mutation testing framework, the full spectrum of possible faults will be tested and identified to ensure the safety and functionality of the software. Since the cost to fix a fault grows significantly as the development life cycle proceeds, the total development cost is expected to decrease as faults are identified and fixed in the early phases of the software life-cycle.

2. Mutation Testing Extension to Requirements and Design Level

Mutation operators are the rules which are followed to generate mutants. A number of mutation operations have been developed for the code level. Table 1 shows some examples of traditional mutation operators.

Table 1 Examples of traditional mutation operators

Operand Replacement Operators	Definition
AAR	Array reference for array reference 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
LCR	Logical connector replacement
ROR	Relational operator replacement
RSR	RETURN statement replacement
SDL	Statement deletion
SRC	Source constant replacement
UOI	Unary operator insertion

To extend code-level mutation testing to the requirements and design level, it is necessary to identify possible requirements and design defects and define new mutation operators. In this section, we introduce our approach to identifying and classifying requirements and design defects. Then mutation operators were developed to model the defects identified.

2.1 Defects classification and mutation operators for requirements and design defects

Defects that appear in requirements and design documents were studied by Li et al. based on the structure of Software Requirements Specifications (SRS) and Software Design Description (SDD) [20]. Twenty-nine defects were identified. We will model the SRS and SDD using Extended Finite State Machines (EFSM). Therefore, we will express the defects using the EFSM as well.

A typical SRS outline based on the IEEE 830-1998 standard is provided in Figure 1. Section 1 in the outline is an introduction section which provides rationale for developing the software. Section 2 provides a general description and describes the high level logic of the software system to be built. Section 3 defines the specific requirements and therefore specifies the software functions at the SRS level as shown in the frame on the right.

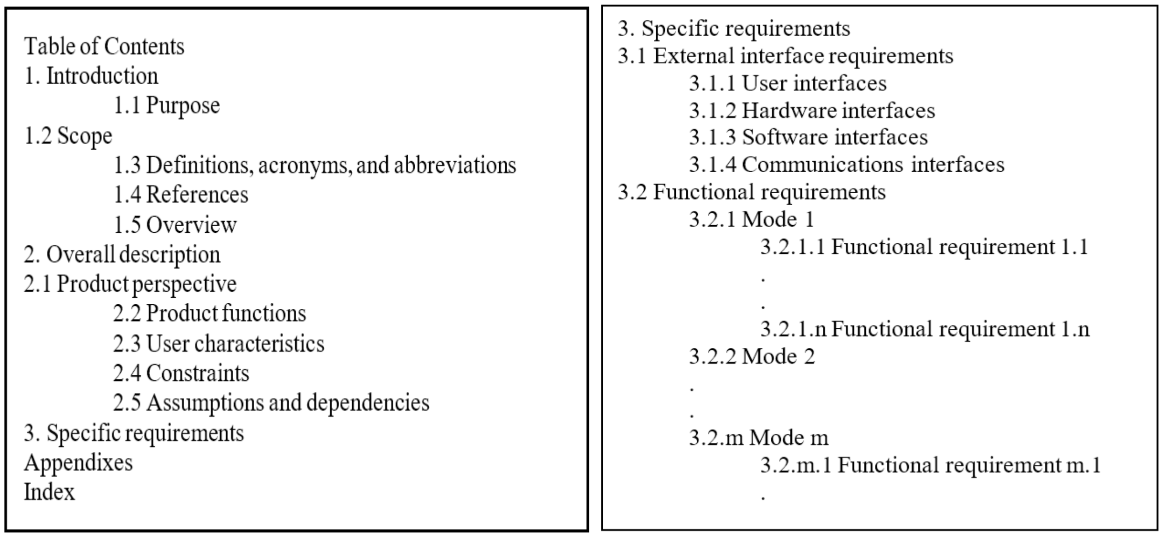

Table of Contents
1. Introduction
    1.1 Purpose
1.2 Scope
    1.3 Definitions, acronyms, and abbreviations
    1.4 References
    1.5 Overview
2. Overall description
2.1 Product perspective
    2.2 Product functions
    2.3 User characteristics
    2.4 Constraints
    2.5 Assumptions and dependencies
3. Specific requirements
Appendixes
Index

3. Specific requirements
3.1 External interface requirements
    3.1.1 User interfaces
    3.1.2 Hardware interfaces
    3.1.3 Software interfaces
    3.1.4 Communications interfaces
3.2 Functional requirements
    3.2.1 Mode 1
        3.2.1.1 Functional requirement 1.1
        .
        .
        3.2.1.n Functional requirement 1.n
    3.2.2 Mode 2
    .
    .
    3.2.m Mode m
        3.2.m.1 Functional requirement m.1
        .

Figure 1 A typical SRS outline based on the IEEE 830-1998 standard

A snippet of functional requirements found in section 3's template consists of the following information:

1. Function name.
2. Input variables: an input variable is a tuple with four elements:
    a. Input name: the name of the input
    b. Type: can be integer, decimal, string or Boolean
    c. Cross application boundary: Boolean variable indicating if an input is directly provided by the user, or any other external entity.
    d. Range: The range of this input
3. Output variables: an output variable is a tuple defined in a similar way as an input variable.
4. Variables: A variable is a tuple defined similarly to input and output variables. Some are global to the entire program, and some are local. In this paper only variables important to the logic of the SRS and SDD are considered.
5. Function logic: This information describes what the function does and how it achieves its goals.

The EFSM model of the SRS is hierarchical. The highest level (i.e., level 0) of the hierarchy is always the SRS itself, and is modeled as one function. The function in specific requirements section has inputs, outputs, variables and a logic. It calls other functions defined in the SRS document, and these functions are at level 1 (i.e., they are SRS level 1 functions).

Twenty-nine defects were identified based on the structure and functionality of the SRS as shown in Table 2. The 29 defects can be classified into 5 categories which are defects for the logic of level 1 functions, defects related to inputs, defects related to outputs, defects related to variables and defects for the logic of the level 0 function. An SRS captures the software requirements that will drive the design and design constraints to be considered or observed [21]. The defects in the SDD are assumed to follow the same classification.

Table 2 Defects and mutation operator

Defect category	Defect name	Mutation Operator
Category 1: Defects for the definition of level 1 functions	1. Missing (definition of) function: The entire definition of a function is missing from the SRS/SDD.	MF
	2. Extra (definition of) function: The entire function definition is extraneous.	EF
	3. Incorrect/ambiguous function name: The name of the function is incorrect/ambiguous.	IAFN
	4. Function with incorrect logic: The functionality is valid but the logic is erroneous.	FIL
	5. Function with Incorrect functionality: The functionality is not valid.	FIF
Category 2: Defects related to inputs	6. Missing input: The definition of an input is missing from the SRS/SDD.	MI
	7. Extra input: The definition of an input is extraneous.	EI
	8. Incorrect/ambiguous input name: The name of the input is incorrect or ambiguous when it is defined	IAIN
	9. Input with incorrect type: The type of the input is erroneously defined.	IIT
	10. Input with incorrect range: The range of the input is erroneously defined.	IIR
Category 3: Defects related to outputs	11. Missing output: The definition of an output is missing from the SRS/SDD.	MO
	12. Extra output: The definition of an output is extraneous.	EO
	13. Incorrect/ambiguous output name: The name of the output is incorrect or ambiguous when it is defined.	IAON
	14. Output with incorrect type: The type of the output is erroneously defined.	OIT
	15. Output with incorrect range: The range of the output is erroneously defined.	OIR
Category 4: Defects related to variables	16. Missing variable: The definition of a variable is missing from the SRS/SDD.	MV
	17. Extra variable: The definition of a variable is extraneous.	EV
	18. Incorrect/ambiguous variable name: The name of the variable is incorrect or ambiguous when it is defined.	IAVN
	19. Variable with incorrect type: The type of the variable is erroneously defined.	VIT
	20. Variable with incorrect range: The range of the variable is erroneously defined.	VIR
Category 5: Defects for the logic of the level 0 function	21. Missing instance of function: The definition of a function is correct, but a call to that function is missing.	MIF
	22. Extra instance of function: The definition of a function is correct, but there is an extra call to that function.	EIF
	23. Incorrect/ambiguous function call: The definition of a function is correct, but when it is called by another function, the name is incorrect or ambiguous.	IAFC
	24. Missing predicate: A predicate of the function is missing.	MP
	25. Extra predicate: A predicate of the function is extraneous.	EP
	26. Incorrect/ambiguous predicate: The logical expression of a predicate is incorrect or ambiguous.	IAP

	27. Missing Loop: A loop of the function is missing.	ML
	28. Extra Loop: A loop of the function is extraneous.	EL
	29. Incorrect/ambiguous Loop condition: The logical expression of the condition of a loop is incorrect or ambiguous.	IALC

3. Mutation Operator Implementation: A Case Study for Missing Input (MI)

For each mutation operator, detailed rules following which mutants can be generated can be developed based on the corresponding defect being modeled. The rules include mutant generation strategy, mutant quantification method and cost reduction technique. The mutation operator MI modeling the defect Missing Input is used in this paper to provide examples for each in turn.

Missing input models the occurrence of a defect whereby an input is missing from the SRS/SDD. The defect template for a function with Missing Input(s) is found in the Figure 2. The defective function may report an error or provide an incorrect output resulting in a faulty behavior of the system.

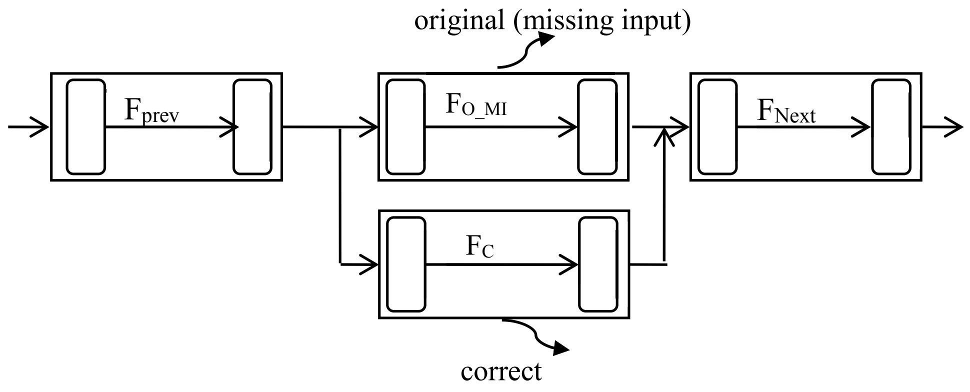

Figure 2 Missing Input defect template

3.1 Mutant Generation and Quantification

Mutation testing aims to estimate the efficiency of test suites and therefore to guarantee the quality of the software. Based on mutation testing hypotheses, all possible mutants should be generated for each mutation operator.

Assume there are *N* functions in the SRS/SDD which are $\{F_1,\ F_2, \dots, F_N\}$ and function F_k has a total number of I_k inputs. A Missing Input mutant M_{ij} can be defined as removing the j^{th} Input I_j from the i^{th} function F_i. The mutant M_{ij} contains the same information as the original SRS/SDD besides the removed input.

There is one mutant corresponding to each input *I*, the total number of mutants n_{MI} generated based on the mutation generator MI can be denoted as follows:

$$n_{MI} = \sum_{k=1}^{N} I_k$$

The total number of mutants generated based on the mutation generator MI is the total number of inputs in all the functions. If the number of inputs is large, a cost reduction technique is necessary.

3.2 Cost Reduction Technology

To reduce the cost of mutation testing, various cost reduction techniques were developed for traditional mutation testing. One such technique, mutants sampling is applied to the extended mutation testing method discussed in this paper.

Mutants sampling randomly selects a subset of mutants. This subset is then tested against the test set to determine its sufficiency (provide a mutation score close enough to the original mutation testing). If sufficiency is not satisfied, a new sample will be selected randomly and added to the subset. This process is continued until the test set is sufficient. By selecting only 10% of the mutant samples, Budd found that the test cases generated from this sample overlooked only 1% of the nonequivalent mutants[5]. In Wong and Mathur's studies [22], the authors suggested that a random selection of 10% of mutants is only 16% less effective than a full set of mutants in terms of mutation score. This study implied that Mutant Sampling is valid for an *x*% value no less than 10%.

To apply this strategy to requirements and design mutation testing, 10% of mutants can be randomly selected for quantifiable mutation operators. We assume that the accuracy of mutant sampling will remain the same as that observed by Budd, Wong and Mathur when the strategy is applied to requirements and design defects.

The total number of mutants after reduction n_{MI}^{reduce} will be

$$n_{MI}^{reduce} = 10\% \times \sum_{k=1}^{N} I_k$$

4. Conclusion

In this paper, traditional mutation testing techniques which have been applied to the code level till now were extended to requirements and design defects. The relevant mutation operators were defined. Mutation strategies including mutant generation, quantification and cost reduction were developed and applied to the mutation operator MI as an example. With the extended mutation testing framework, the full spectrum of possible faults will be tested and identified to ensure the safety and functionality of the software

Future work will focus on execution of the mutants generated, and the automation of the mutation process as well as the validation of the cost reduction strategies proposed.

5. Reference

[1] Y. Jia, S. Member, and M. Harman, "An Analysis and Survey of the Development of Mutation Testing," vol. 37, no. 5, pp. 649–678, 2011.

[2] R. G. Hamlet, "Testing Programs with the Aid of," no. 4, 1977.

[3] R. J. Lipton, "Fault Diagnosis of Computer Programs," 1971.

[4] R. A. Demillo, R. J. Lipton, and F. G. Sayward, "Hints on Test Data Selection : Help for the Practicing Programmer," 1978.

[5] T. A. Budd, "Mutation analysis of program test data," Yale University, 1980.

[6] A. Acree, "On Mutation," Georgia Institute of Technology, 1980.

[7] S. Hussain, "Mutation Clustering," King's College London, 2008.

[8] E. S. Mresa and L. Bottaci, "Efficiency of mutation operators and selective mutation strategies: An empirical study," *Softw. Test. Verif. Reliab.*, vol. 9, no. 4, pp. 205–232, 1999.

[9] a. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, "An experimental determination of sufficient mutant operators," *ACM Trans. Softw. Eng. Methodol.*, vol. 5, no. 2, pp. 99–118, 1996.

[10] A. Offutt and G. Rothermel, "An experimental evaluation of selective mutation," *15Th Int. Conf.*, pp. 100–107, 1993.

[11] W. E. Wong, "On Mutation and Data Flow," Purdue University, 1993.

[12] A. P. Mathur, "Performance, Effectiveness, and Reliability Issues in Software Testing," in *Proceedings of the Fifteenth Annual International*, 1991, pp. 604–605.

[13] Y. Jia and M. Harman, "Constructing Subtle Faults Using Higher Order Mutation Testing," *2008 Eighth IEEE Int. Work. Conf. Source Code Anal. Manip.*, pp. 249–258, 2008.

[14] W. E. Howden, "Weak Mutation Testing and Completeness of Test Sets," *IEEE Trans. Softw. Eng.*, vol. 8, no. 4, pp. 371–379, 1982.

[15] A. J. Offutt and S. D. Lee, "Empirical evaluation of weak mutation," *IEEE Trans. Softw. Eng.*, vol. 20, no. 5, pp. 337–344, 1994.

[16] M. R. Woodward and K. Halewood, "From weak to strong, dead or alive? An analysis of some mutation testing issues," in *Software Testing, Verification, and Analysis*, 1988.

[17] B. Bogacki and B. Walter, "Evaluation of Test Code Quality with Aspect-Oriented Mutations," *Proc. 7th Int. Conf. Extrem. Program. Agil. Process. Softw. Eng.*, vol. 4044, no. 1, pp. 202–204, 2006.

[18] R. H. Untch, "Mutation-based Software Testing Using Program Schemata," *Proc. 30th Annu. Southeast Reg. Conf.*, pp. 285–291, 1992.

[19] R. A. DeMillo, E. W. Krauser, and A. P. Mathur, "Compiler Integrated Program Mutation," *Proc. 5th Annu. Comput. Softw. Appl. Conf.*, pp. 351–356, 1991.

[20] X. Li and J. Gupta, "ARPS: An Automated Reliability Prediction System Tool for Safety Critical Software," *PSA*, vol. 2013, pp. 22–27, 2013.

[21] IEEE Computer Society, *IEEE Std 1016-2009, IEEE Standard for Information Technology—Systems Design—Software Design Descriptions*. 2009.

[22] I. Moore, "Jester and Pester," 2001. [Online]. Available: http://jester.sourceforge.net/.