JKSCI

한국컴퓨터정보학회논문지
Journal of The Korea Society of Computer and Information
Vol. 29 No. 6, pp. 89-100, June 2024
https://doi.org/10.9708/jksci.2024.29.06.089

Main causes of missing errors during software testing

Young-Mi Kim*, Myung-Hwan Park**

*Ph.D. Student, Dept. of Computer & Radio Communication Engineering, Korea University, Seoul, Korea
**Associate Professor, Dept. of Computer Science, Korea Air Force Academy, Cheongju, Korea

[Abstract]

The primary goal of software testing is to identify and correct errors within software. A key challenge in this process is error masking, where errors disappear internally before reaching the output. This paper investigates the causes and characteristics of error masking, which complicates software testing. The study involved injecting artificial errors into three software programs to examine the extent of error masking by various test cases and to explore the underlying reasons. The experiment yielded four major findings. First, about 50% of the error masking occurred because the errors were not executed. Second, among various operators, logical and arithmetic operators masked errors less frequently, while relational and temporal operators tended to mask errors more extensively. Third, certain test cases demonstrated exceptional effectiveness in propagating errors to the output. Fourth, the type of error injected influenced the masking effect.

▸ **Key words**: Error masking, Error propagation, Test cases, Test suite, Testing efficiency, Mutant

[요 약]

소프트웨어 테스팅의 궁극적인 목표는 소프트웨어의 에러를 찾아내고 수정하는 것이다. 소프트웨어 에러를 발견하기 어렵게 만드는 요인 중에는 소프트웨어의 에러가 출력에 도달하기 전에 내부에서 마스킹 되어 사라지는 것이다. 이 논문의 목적은 소프트웨어 테스팅을 어렵게 만드는 에러 마스킹의 원인 및 특성을 조사하는 것이다. 이를 위해 3개의 소프트웨어를 대상으로 인위적인 에러를 주입하여 그 에러가 다양한 테스트 케이스들에 의해서 얼마만큼 마스킹 되는지, 그리고 그 원인은 무엇인지를 조사하였다. 실험 결과 4가지 주요 발견이 도출 되었다. 첫째, 약 50% 정도의 에러 마스킹은 에러가 실행되지 않았기 때문에 발생하였다. 둘째, 여러 연산자들 중에서 논리연산자와 산술연산자는 에러를 상대적으로 적게 마스킹하고, 관계연산자와 시간 연산자는 에러를 상대적으로 많이 마스킹하였다. 셋째, 테스트 케이스들 중에 에러를 출력까지 전파시키는데 특별한 성능을 보이는 테스트 케이스의 존재를 확인할 수 있었다. 넷째, 주입한 에러의 종류에 따라서 마스킹 효과가 다르다는 것을 확인할 수 있었다.

▸ **주제어**: 에러 마스킹, 에러 전파, 테스트 케이스, 테스트 수트, 테스팅 효율성, 뮤턴트

• First Author: Young-Mi Kim, Corresponding Author: Myung-Hwan Park
*Young-Mi Kim (myjulietkim@gmail.com), Dept. of Computer & Radio Communication Engineering, Korea University
**Myung-Hwan Park (pigisum@gmail.com), Dept. of Computer Science, Korea Air Force Academy
• Received: 2023. 05. 08, Revised: 2024. 05. 27, Accepted: 2024. 06. 11.

Copyright © 2024 The Korea Society of Computer and Information
http://www.ksci.re.kr pISSN:1598-849X | eISSN:2383-9945
www.kci.go.kr

I. Introduction

The primary goal of software testing is to find errors in the software and fix them before the program is released [1]. Thus, much research has focused on improving the testing process to achieve this goal. For example, many studies have attempted to generate more rigorous test data to more thoroughly execute the construct of the software. Recently, a test oracle, a mechanism that determines the success or failure of testing, has also drawn the attention of researchers as a factor to increase the fault-finding capability of testing [2,3,4].

However, previous work has found that many errors in the software cannot be caught even using the most rigorous test data and test oracle [5,6]. Nevertheless, the testing community has not found explicit evidence for why the errors cannot be caught during testing. The inability to detect errors during testing can often be linked to not meeting the specific criteria set forth in the RIP (Reachability, Infection, Propagation) model [7,8,9,10]. This model outlines essential conditions for error detection in software testing, suggesting that successful error detection is contingent upon fulfilling these conditions. In the RIP model, Reachability means that a faulty construct in the software is executed during test execution and Infection means that a fault in the software is changed to an erroneous state, so at least one variable in the software is infected by a fault. Finally, Propagation means that the erroneous value of a variable is propagated to output. If any one of these three conditions fails to be satisfied, errors cannot be revealed during testing.

In the test community, there is no clear understanding of what factors are predominant in not discovering errors during testing. In addition, to the best of our knowledge, no previous study has examined when an error is masked out before it propagates to the monitored variable, and what construct in the program plays a pivotal role in this masking.

In this paper, we introduce experimental results to investigate the causes of missing errors during testing. For the experiment, we use three software that are all in the form of Lustre language [11,12]. We generate test suites (set of test cases) satisfying three different structural coverage criteria: condition, decision, and MC/DC. We also generate 400 mutants per software by seeding a simple typo fault into each software. Test execution is performed using test cases with mutants and the original model (oracle, hereafter). During test execution, we monitor the value of each variable of the mutant and oracle and compare the values of the mutant and oracle variables. We finally determine if the error propagates to the output variable. If the error does not propagate to the output variable, we investigate the reason for missing the error.

The results of our experiment draw four key conclusions. First, the main cause of missing errors is due to the reachability failure. Second, among various operators, logical and arithmetic operators masked errors less frequently, while relational and temporal operators tended to mask errors more extensively. Third, certain test cases demonstrated exceptional effectiveness in propagating errors to the output. Fourth, the type of error injected influenced the masking effect.

II. Background and Related Work

In this section, we present the definition of terms that we used in this paper and explain the Lustre language. We also discuss some related works regarding error masking and propagation issues during software testing.

1. Terminology

- **Fault:** it refers to mistakes, like a typo, in the source code, a faulty instruction or missing data [13].

www.kci.go.kr

- **Error:** it refers to an erroneous state of the program when a fault is executed, which results in an erroneous value in at least one variable [13].
- **Reachability failure:** In Lustre, every variable is evaluated at each time step, distinguishing it significantly from other languages. However, despite this evaluation, not all constructs in Lustre necessarily execute. For instance, within the construct "if C then S1 else S2," both S1 and S2 are evaluated, but only the branch selected by the conditional C executes. Therefore, in our experiments, reachability failure only occurs in scenarios involving this specific conditional construct.
- **Infection failure:** This failure happens when there is a fault in the software, but an error is not manifested in any variable in the software. For example, if the original program “A OR B” has been replaced with “A AND B” and the test input is True for A and B, the error cannot be manifested even though a fault exists in the program.
- **Propagation failure:** This failure happens when an error is manifested but disappears during the operation of operators. For example, suppose variable A is an error. In an expression, “C = A AND B,” if B is false, then the value of C becomes false regardless of the value of A. In this case, the error in A is masked out by the AND operator. We further classify masked cases into the following: logical operator masking, relational operator masking, temporal masking, arithmetic operator masking.
  - **Logical operator masking:** This masking occurs by the logical operator.
  - **Relational operator masking:** This masking occurs by a relational operator. For example, suppose the correct value of variable B is 3, but it has 4 instead. In expression “C = 5 >= B,” the error in B will be masked out in C.
  - **Temporal masking:** This masking occurs by a delay operator. In the expression “C = 1 → PRE(B) ,” C will have a stream of values. The first value is 1 followed by a one-step previous value of B. The error in B will propagate to C at time step 2 since the value 1 will be assigned to C at time step 1. If the test terminates at time step 1, then the error is masked out in C.
  - **Arithmetic operator masking:** This masking occurs by an arithmetic operator. In expression “C = B / 10,” suppose the correct value of B is 3, but it has 4 instead. The evaluation of the expression becomes 0 for both cases since the type of the expression is an integer. Thus, the error in B is masked out in C

2. Lustre Language

Lustre [11,12] is a declarative and synchronous dataflow programming language proposed for the design of reactive systems. Lustre is the core language of the SCADE tool [14], developed by Esterel Technologies, and is commonly used for critical control software in aircraft and nuclear power plants. This language is based on a synchronous paradigm, and the behavior of a system is a sequence of reactions. Each reaction is meant to read current inputs, update the value of the variables and evaluate the output value. The synchronous paradigm assumes that the reaction of the system is instantaneous. The Lustre program consists of nodes, which models the subprogram in the modular language.

3. Related works

Study on the error flow characteristic of a system can be classified into two areas. The first area is the study about how the productive test cases, which can enforce the error to propagate to output, can be generated. Most works to improve the quality of test cases in terms of structural coverage criteria fall into this area.

www.kci.go.kr

Much research focuses on improving the quality of test case satisfying the traditional coverage criteria such as condition, decision, branch and MC/DC [15,16]. Recently, Whalen et al. [13,17] suggested a new structural coverage criterion, called Observable MC/DC (OMC/DC), that combines the MC/DC coverage metric with a notion of observability. This coverage criterion ensures that the error in the program has to propagate to a monitored variable.

The second area is the study about investigating the error propagation characteristic of the program. Several previous works on measuring the error propagation probability of the system are related to this area [18,19,20,21,22].

Goraddia [18] suggested a dynamic impact analysis to measure the impact strength of error propagation from a variable to the other variable. The goal of this approach was to estimate the pervasiveness of errors through program construct.

Voss [19] proposed a sensitivity analysis to rank program locations based on their ability (sensitivity) to propagate an error in their location to output variables. The sensitivity of a location depends on the execution probability, infection probability, and propagation probability in the location.

Abdelmoez et al. [20] suggested an estimation technique of error propagation probability between software components. Error propagation matrix is computed from the error propagation probability among components.

Jahangirova et al. [21] carried out an empirical analysis on the characteristics of failed error propagation across six real-world Java open-source projects involving 386 actual faults. Their findings indicated that the impact of failed error propagation is minimal during unit-level tests but becomes significantly more pronounced in system-level testing.

Chan et al. [22] developed a framework named Invariant Propagation Analysis (IPA), which automatically generates dynamic invariants by instrumenting source code at the entry and exit points of functions. This approach aims to analyze error propagation in multi-threaded programs.

III. Experiment

We are interested in understanding how many errors in the software are missing (not revealed) during testing and what factors are attributed to the missing errors. To answer this question, we designed experiments.

For the experiment, we perform the following steps:

1. We choose 3 Lustre software as case examples to be used in the experiment (Section III.1).
2. We generate 400 mutants for each case examples, with each mutant containing a single fault (Section III.2)
3. We generate the test suite satisfying condition, decision, and MC/DC coverage (Section III.3)
4. We run the mutant and oracle with the test suites to investigate the cause of the missing error (Section III.4).

1. Case Examples

For the case example in this study, we use 3 software in the Lustre language form. Two of these systems, Infusion and Alarm, are medical systems designed for medical research [23]; Infusion is a prescription management system and Alarm is an alarm-induced system of an infusion pump device. The other system is a Microwave system that controls microwave ovens developed by Rockwell Collins [17]. All models are non-proprietary and developed for research and teaching purposes. Table 1 provides details about the case examples.

www.kci.go.kr

Table 1. Case example information

	Line of code	# input variable	# output variable	# internal variable
Infusion	6485	20	5	825
Alarm	6117	42	5	934
Microwave	1444	13	4	46

2. Mutant Generation

We generate 400 mutants for each case example by replacing a program construct of correct model (oracle) into a different construct. The seeded fault (replaced construct) is type-compatible to original one so that it does not trigger grammar errors during compile time. Each fault falls into one of 4 different types and 100 mutants was created for each type (totally 400 mutants for a case example).

- **Logical operator fault:** A logical operator is randomly chosen and replaced by a different one.
- **Arithmetic operator fault:** An arithmetic operator is randomly chosen and replaced by a different one.
- **Relational operator fault:** A relational operator is randomly chosen and replaced by a different one.
- **Literal fault:** A literal is randomly chosen and replaced by a larger or less literal one. For example, 3 is replaced by 2 or 4 in a random manner.

3. Test Case Generation

In this study, we generate test suites (a set of test cases) satisfying three structural coverage criteria: condition coverage, decision coverage, and modified condition/decision coverage (MC/DC). Before explaining the coverage criteria, we first introduce the terms: "condition" and "decision". Condition is a Boolean expression containing no Boolean operators. Decision is a Boolean expression composed of conditions and zero or more Boolean operators.

- **Condition coverage:** This coverage criterion is achieved when the condition is evaluated as both true and false at least once during the execution of the test suite.
- **Decision coverage:** This coverage criterion is achieved when the decision is evaluated as both true and false at least once during the execution of the test suite
- **MC/DC coverage:** This coverage criterion is achieved when every condition in a decision in the program is evaluated as both true and false at least once, every decision in the program is evaluated to true or false at least once, and each condition in a decision has been shown to independently affect that decision's outcome.

A more detailed explanation about the structural coverage criteria can be found in the [16]. Table 2 shows the size of the test suites (number of test cases in the test suite) achieving the three structural coverage criteria for the case examples.

Table 2. Number of test cases for case examples

	Condition coverage	Decision coverage	MC/DC coverage
Infusion	91	69	104
Alarm	232	196	227
Microwave	850	392	724

4. Test Execution

We run the experiment to examine the number of missing errors and the causes.

First, we run the oracle program with the test suites as explained above and record the values of every programming construct (including the variables) at every time step during test execution. For example, in the expression "C = A * B + 3 >= 10," we record the result of every subexpression during evaluation such as "A * B," and "A * B + 3," "A * B + 3 >= 10." To this end, we construct a parse tree like a structure (called an error propagation tree) for the Lustre program. During test execution, we trace the data flows from the input variable to the output variable in the error

www.kci.go.kr

propagation tree and record the value changes at every construct of the tree at every time step.

Second, we run the mutant program and record the value changes in the same way as the oracle program.

Third, we compare the recorded values of the oracle with those of the mutant and identify the error-missing points when the error does not propagate to a further construct in the error propagation tree.

Finally, we categorize the cause of each missing error based on the construct type of the error-missing points.

They are further categorized as reachability failure, infection failure, or propagation failure. Propagation failure is further classified into logical operator masking, relational operator masking, arithmetic operator masking, or temporal masking.

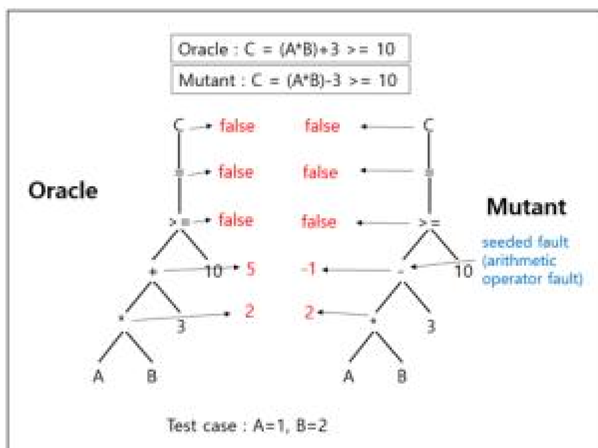

Fig. 1. Experiment process with oracle and mutant

Fig 1 illustrates the experiment process using an oracle and a mutant. In the oracle program, a '+' operator is replaced with a '-' to create a mutant. For the test case, the input variable A is set to 1, and input variable B to 2. Following the error propagation tree, both the oracle and the mutant yield a result of 2 at the '*' operator. However, at the oracle's '+' operator, the result is 5, whereas at the mutant's '-' operator, it results in -1, allowing for the detection of an error through comparison. Nonetheless, at the '>=' operator, both the oracle and the mutant produce a false value, leading to the error being masked. Consequently, a propagation failure occurs, primarily due to relational operator masking.

IV. Results and Implications

In this section, we present our results and discuss the implications. We first present our findings on the characteristics of the test suites, test cases, and mutant types on revealing errors in the mutants. Next, we discuss implications of the findings. Finally, we present threats to the validity of our study.

1. Experiment Results

Table 3 shows the ratio of mutants that are caught by at least one test cases in the test suite for each case example. The results indicate that in more than 60% of the mutants, the seeded fault was not executed, or was not manifested, or was masked out during the execution of all test cases in the test suite. Even the more rigorous test suite (MC/DC) did not show superior power to catch errors in our experiment. The cause of these results will be addressed in the Implications section.

Table 3. Percentage of mutants caught for each case example over the test suites

	Condition coverage	Decision coverage	MC/DC coverage
Infusion	22.5%	26.5%	31.3%
Alarm	30.0%	26.8%	28.2%
Microwave	40.0%	39.3%	37.8%

Table 4 illustrates the fault detection capabilities of each test case within a test suite. It highlights that the average percentage of mutants each test case could identify as faulty was below 10%. This indicates that individual test cases are significantly less effective in revealing faults compared to the

www.kci.go.kr

collective detection rate of the test suites. Additionally, the test cases within the MC/DC test suite exhibited slightly better error detection rate than those within the Condition or Decision test suites. While the average error detection capability of the test cases is quite low, there exist individual test cases with significantly superior error detection performance. Fig 2 demonstrates the number of mutants detected by each test case, revealing considerable variation in their error-catching abilities. As shown in Fig 2, in the Infusion, more than 50 test cases failed to detect any mutants, whereas one test case detected 79 mutants. In the Alarm, some test cases caught only 2 mutants, while another managed to detect 47 mutants. Similarly, in the Microwave, while some test cases caught 10 mutants, another was able to detect as many as 84 mutants. This indicates a wide disparity in performance among individual test cases within the same test suites.

Table 4. Average percentage of mutants detected by each test case in every given case example

	Condition coverage	Decision coverage	MC/DC coverage
Infusion	3.7%	3.9%	4.2%
Alarm	3.2%	2.9%	3.2%
Microwave	9.4%	9.6%	10.0%

Table 5 shows the average percentage of mutants caught by the test cases by mutant type. The types of mutants most frequently caught vary by case example. For instance, in the Infusion, relational operator faults were the most frequently detected, while in Alarm, literal faults predominated, and in Microwave, relational operator faults were again most prevalent. Although the most commonly caught mutant type differs across case examples, within each case example there is a significant disparity between the most and least caught mutant types. In Infusion, the ratio between the most and least caught mutant types is approximately 3.4 times; in Alarm, it is 3 times; and in Microwave, it reaches 3.7 times. This highlights substantial variations in detection effectiveness within individual case examples.

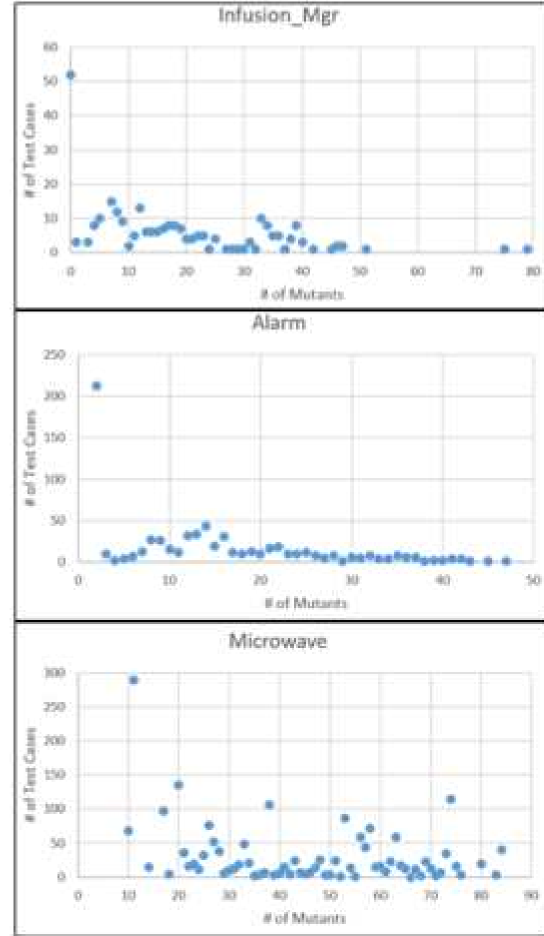

Fig. 2. Number of mutants revealed by the number of test cases for each case example

Table 5. Average percentage of mutants caught for each case example over mutant types

	Logical operator	Arithmetic operator	Relational operator	Literal fault
Infusion	3.6%	1.5%	5.7%	5.1%
Alarm	2.6%	1.5%	3.7%	4.5%
Microwave	7.6%	15.4%	11.5%	4.2%

Table 6. Number of mutants not caught by any test case

	Logical operator	Arithmetic operator	Relational operator	Literal fault
Infusion (269)	67 (24.9%)	74 (27.5%)	61 (22.7%)	67 (24.9%)
Alarm (268)	57 (21.3%)	81 (30.2%)	67 (25%)	63 23.5%)
Microwave (225)	52 (21.3%)	59 (26.2%)	45 (20%)	69 (23.5%)

www.kci.go.kr

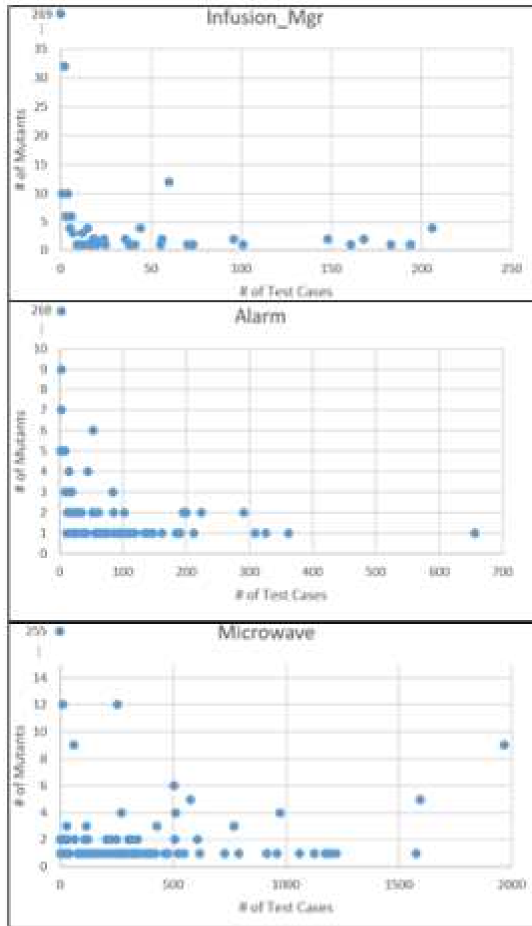

Fig. 3. Distribution of number of mutants caught and number of test cases

Table 6 provides another insight about the error-revealing characteristics of mutant types. The table shows the number of mutants that were not caught by any test cases. It is interesting that over 50% of the mutants were not caught by any test cases regardless of their coverage criteria. Fig 3 plots the distribution of the number of mutants caught and number of test cases. As seen in Fig 3, while some mutants are caught by a large number of test cases, others are caught by only a few. This issue will be further discussed in the Implications section.

Table 7 presents the underlying causes for errors not detected in our study. Within the various case examples, the predominant factor for missing errors is a failure in reachability, accounting for approximately 50% of all undetected errors. Subsequent prevalent causes vary by case examples; however, masking by relational operators consistently exerts a significant influence on propagation failures across all cases.

Table 8 delineates the relationship between the causes of undetected errors and the test suite coverage criteria. Notably, the MC/DC test suite, which adheres to the most stringent coverage criteria, exhibits the highest incidence of reachability failures. While this will be further addressed in the Implications section, it can be observed that the MC/DC test cases frequently miss errors in IF-then-else statements. For other types of missing errors, no significant differences were observed across different coverage criteria.

Table 9 presents the causes of errors not detected in relation to the types of mutants. A notable observation is that literal mutants consistently show a higher incidence of reachability failures across all case examples. This is likely because the location of literal faults is predominantly within If-then-else statements, which were not executed. Infection failures for literal mutants are uniformly 0% across all cases, suggesting that if a literal fault is executed, it invariably transitions the software to an erroneous state. Moreover, logical and relational operator faults exhibit a relatively higher rate of missing errors due to infection failures. This indicates that

Table 7. Causes of missing errors for the case examples (LOM: logical operator masking, ROM: relational operator masking, TM: temporal masking, AOM: arithmetic operator masking)

	Reachability failure	Infection failure	Propagation failure			
			LOM	ROM	TM	AOM
Infusion	48.2%	1.4%	2.3%	24.2%	23.9%	0%
Alarm	49.3%	11.1%	1.4%	30.6%	7.0%	0.4%
Microwave	57.0%	22.7%	2.1%	8.2%	2.4%	7.6%

www.kci.go.kr

Table 8. Cause of missing errors for the case examples by coverage criteria (LOM: logical operator masking, ROM: relational operator masking, TM: temporal masking, AOM: arithmetic operator masking)

	Coverage Criteria	Reachability failure	Infection failure	Propagation failure			
				LOM	ROM	TM	AOM
Infusion	Condition	41.4%	1.7%	2.2%	24%	30.6%	0%
	Decision	38.5%	1.4%	2.4%	24.2%	33.5%	0%
	MC/DC	57.7%	1.2%	2.4%	24.2%	14.6%	0%
Alarm	Condition	49.4%	11.3%	1.4%	30.2%	7.1%	0.4%
	Decision	47.5%	11.4%	1.3%	31.3%	8.0%	0.5%
	MC/DC	50.8%	10.7%	1.4%	30.5%	6.2%	0.4%
Microwave	Condition	55.8%	23.0%	3.0%	8.6%	2.6%	7.0%
	Decision	57.1%	22.7%	1.4%	8.3%	1.8%	8.8%
	MC/DC	58.1%	22.5%	1.5%	7.7%	2.6%	7.6%

Table 9. Cause of missing errors for the case by mutant type (LOM: logical operator masking, ROM: relational operator masking, TM: temporal masking, AOM: arithmetic operator masking)

	Mutant type	Reachability failure	Infection failure	Propagation failure			
				LOM	ROM	TM	AOM
Infusion	Logical	48.0%	7.4%	3.2%	13.5%	27.8%	0%
	Arithmetic	37.4%	0%	1.8%	39.6%	21.3%	0%
	Literal	53.3%	0%	1.9%	21.5%	23.3%	0%
	Relational	46.6%	4.6%	4.5%	17.0%	27.3%	0%
Alarm	Logical	50.9%	24.5%	0.8%	13.3%	10.5%	0%
	Arithmetic	44.6%	3.7%	1.6%	44.1%	4.8%	1.2%
	Literal	52.9%	0%	0.9%	41.0%	4.8%	0%
	Relational	51.8%	19.8%	2.1%	17.1%	9.2%	0%
Microwave	Logical	53.8%	33.7%	6.5%	2.9%	1.1%	2.0%
	Arithmetic	56.6%	27.7%	0.9%	6.1%	0.7%	8.1%
	Literal	61.7%	0%	0%	15.6%	1.6%	21.2%
	Relational	56.2%	26.6%	1.4%	8.6%	5.5%	1.8%

even if these operators are replaced by others, the output may not change depending on the input, meaning that errors can remain masked. For example, if the expression "A and B" is altered to "A or B," the output remains unchanged when both A and B are either true or false, hence masking the error.

2. Implications

The experiment in this study reveals several insights about factors that influence test effectiveness as follows.

2.1 Most influential factor for missing errors

As indicated in Table 7, it is very interesting that over 50% of the errors disappear due to reachability failure regardless of the coverage criteria. This is a counter-intuitive result from two perspectives. First, much research has attempted to generate productive test cases to enforce test cases to comb every program structure under test. However, our results show that a significant number of errors are not propagated to a further program construct due to not being chosen from the “If C then S1 else S2” construct. Second, we believe the test suite with more rigorous coverage criterion will be superior to overcome the masking effect of a program construct. As noted in Table 8, however, the MC/DC test suite shows less capability to cope with the masking effect of the “If C then S1 else S2” construct. The analysis of the source code from the case examples reveals that they were initially created using a graphical tool called SCADE, and then mechanically translated into the Lustre language by a compiler. The source code heavily utilizes nested If-then-else statements, where a then or else clause contains another If-then-else statement,

www.kci.go.kr

often nested dozens of times. Even the most robust MC/DC test suites ensure only that each individual If-then-else statement executes both then and else clauses but do not guarantee the execution of all embedded statements within nested structures. This is identified as a contributing factor to the problem. Specifically, when If-then-else statements are nested, there are segments that the MC/DC test suite cannot execute. With increased nesting, the unexecuted segments become more prevalent. Developing a test suite that ensures the execution of every decision in such nested If-then-else statements will be further research topic.

2.2 Least influential factor for missing errors

As indicated in Table 7, logical operator masking and arithmetic operator masking has the least influence on missing errors during testing. This is also a somewhat counter-intuitive result to the most common belief. The chance of errors being masked in a logical operator is 50%. Thus, we expect that the logical operator will be an influential factor in error masking. However, our experiment showed that a logical operator has a much lower impact on error masking than that of a relational operator and temporal masking.

2.3 Critical test cases

As shown in Table 4, the fault-finding capability of most individual test cases is very disappointing. Among 400 mutants, the average percentage of mutants that the individual test case caught was below 10%. More surprisingly, among the 262 test cases in the Infusion case example, 52 test cases did not catch any mutant, as shown in Fig 1. However, the other single test case caught 79 mutants in the same case example. We call this exceedingly productive test case a critical test case. The other case examples include a test case that caught 47 mutants in the Alarm case example and a test case that caught 84 mutants in the Microwave case example. Including a critical test case in the test suite can significantly affect the fault-finding capability of test suites. If we can generate as many critical test cases as possible, the testing efficiency will be further enhanced. However, there is still little understanding on what makes a critical test case and how.

2.4 Pervasive vs. insulant mutant

As shown in Tables 5 and 6, some mutants are very insulant to propagate errors to other constructs of the program. For example, the Infusion case example has 269 mutants that were not caught by any test cases. The errors in these mutants are very difficult to find. On the other hand, some mutants are very pervasive in propagating errors to other constructs of the program. As an example, four mutants in Infusion were caught by 206 test cases. Further research should focus on what makes a mutant pervasive or insulant in propagating errors in the software.

3. Threats to Validity

- **Mutant generation:** We generate four different fault type to simulate the programmers' faults during programming. The faults generated were just a replacement of an operator or literal. This simple fault scheme does not fit with programmers' real faults such as replacing variables or omitting instructions. The reason we use this restricted fault type is due to the limitation of our tool. If the program structure of mutants is far different from the original program (oracle), comparing variables between the oracle and mutant becomes impossible or much more challenging. However, we believe that the causes of missing errors are similar to our results when even complicated faults are used. This is because the missing errors due to reachability failure or propagation failure will be equally applied to complicated faults.
- **Test suite coverage:** We generate a test suite using the open source program released by

www.kci.go.kr

Gregory Gay, Assistant Professor in the Department of Computer Science & Engineering, at the University of South Carolina. However, we are not sure if the generated test suite fully achieves the intended coverage criteria. Depending on the program structure, it is not possible to generate a test suite to satisfy a certain coverage criteria. Thus, the test suite generated in this study may not fully achieve the intended coverage criteria. Nevertheless, the main purpose of this study is not to investigate the quality of the test suite, but to investigate the characteristics of error propagation and masking. Thus, even with the current test suite, the contribution of this study will not be compromised.

- **Language choice:** We used Lustre as the basic language for the experiments and analysis. Although Lustre is not a popular language compared to more common languages, it has a similar structure to systems written in C or C++. There are also translation tools from Lustre program to C or C++ program. Thus, we believe our results are also applicable to programs written in those languages.

V. Conclusions

In this study, we examine the cause of missing errors during testing. The experimental results indicate that more than 50% of the missing errors occur because they do not have a chance to execute. If an error is not executed, it cannot further propagate to other places. Relational operator has a relatively higher impact on error masking. In contrast, logical operator and arithmetic operator have a very small impact on error masking. This finding suggests that much work remains to be done to determine a way to execute programs more thoroughly and elaborately.

We also found that a large portion of test cases showed a very poor capability to catch errors in mutants while a few test cases showed a very impressive capability to catch these errors. In addition, some mutants are very pervasive in propagating errors while others were very insolent to transmit errors. Further research is needed to address these findings to seek answers on what causes these differences.

ACKNOWLEDGEMENT

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP; Ministry of Science, ICT & Future Planning) (No. 2018R1D1A1B07050181).

REFERENCES

[1] G. J. Myers, “Art of Software Testing”, John Wiley & Sons, Inc., New York, NY, 1979

[2] G. Gay, M. Staats, M. Whalen, and M. Heimdahl, “Automated oracle data selection support,” IEEE Transactions on Software Engineering, 2015. DOI: 10.1109/TSE.2015.2436920

[3] E. Barr, M. Harman, P. McMinn et al., "The oracle problem in software testing: a survey", IEEE Trans. Softw. Eng., vol. 41, pp. 507-525, 2015 DOI: 10.1109/TSE.2014.2372785

[4] R. A. P. Oliveira, U. Kanewala, and P. A. Nardi. “Automated test oracles: State of the art, taxonomies, and trends”. Advances in Computers, 95:113-199, 2015. DOI: 10.1016/B978-0-12-800160-8.00003-6

[5] G. Gay, M. Staats, M. Whalen, and M. Heimdahl, “The risks of coverage-directed test case generation,” IEEE Transactions on Software Engineering, vol. 41, no. 8, pp. 803-819, 2015. DOI: 10.1109/TSE.2015.2421011

[6] S. Rayadurgam and M. P. E. Heimdahl, “Coverage based test-case generation using model checkers,” in Proceedings of the Eighth Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems, 2001, pp. 83-91. DOI: 10.1109/ECBS.2001.922409

[7] R. A. DeMillo and J. Offutt. “Constraint-based automatic test data generation. IEEE Transaction on Software Engineering,

www.kci.go.kr

17(9):900-910, September 1991.

[8] N. Li and J. Offutt. “Test Oracle Strategies for Model-Based Testing”. IEEE Transactions on Software Engineering, January 2016.

[9] L. J. Morell. “A theory of error-based testing”. IEEE Transactions on Software Engineering, 16(8):844-857, August 1990.

[10] J. Offutt. Automatic Test Data Generation. PhD thesis, Georgia Institute of Technology, Atlanta, GA, USA, 1988.

[11] N. Halbwachs, L. Fabienne, and R. Christophe. "Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE." IEEE transactions on software engineering Vol. 18, No. 9, 1992.

[12] N. Halbwachs et al. “The Synchronous Data Flow Programming Language LUSTRE.” In Proc. IEEE Vol. 79, No. 9, 1991.

[13] M. Whalen, G. Gay, D. You, M. P. E. Heimdahl, and M. Staats, “Observable modified condition/decision coverage,” in Proceedings of the 35th Internationa l Conferenceon Software Engineering, 2013, pp.102-111. DOI: 10.1109/ICSE.2013.6606556

[14] J. L. Colaço, B. Pagano and M. Pouzet, "SCADE 6: A formal language for embedded critical software development (invited paper)," 2017 International Symposium on Theoretical Aspects of Software Engineering (TASE), 2017.

[15] J. J. Chilenski and S. P. Miller, “Applicability of modified condition/decision coverage to software testing,” Software Engineering Journal, vol. 9, no. 5, pp. 193-200, 1994. DOI: 10.1049/sej.1994.0025

[16] M. Pezzè and M. Young, “Software Testing and Analysis: Process, Principles, and Techniques.” Wiley, 2008.

[17] Y. Meng, G. Gay and M. Whalen, “Ensuring the Observability of Structural Test Obligations”, IEEE Transactions on Software Engineering, Vol 46, Issue 7, 2018.

[18] T. Goradia, "Dynamic Impact Analysis: A Cost-Effective Technique to Enforce Error-Propagation", Proc. ACM Int'l Symp. Software Testing and Analysis, pp. 171181, June 1993. DOI: 10.1145/174146.154269

[19] J. Voas, "PIE: A Dynamic Failure-Based Technique", IEEE Trans. Software Eng., vol. 18, no. 8, pp. 717727, Aug. 1992. DOI: 10.1109/32.153381

[20] W. Abdelmoez, D.M. Nassar, M. Shereshevsky, N. Gradetsky, R. Gunnalan,H.H. Ammar, Yu Bo, A Mili, "Error propagation in software architectures", Proceedings of International Symposium on Software Metrics, pp. 384-393, 2004. DOI: 10.1109/METRIC.2004.1357923

[21] G. Jahangirova, D. Clark, M. Harman, and P. Tonella, “An empirical study on failed error propagation in java programs with real faults,” 2020

[22] A. Chan, S. Winter, H. Saissi, K. Pattabiraman, and N. Suri, “Ipa: Error propagation analysis of multi-threaded programs using likely invariants,” in 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST), pp. 184-195, 2017.

[23] A. Murugesan, M. Whalen, S. Rayadurgam and M. P. E. Heildahl, “ Compositional Verification of a Medical Device System”, Proceedings of the 2013 ACM SIGAda Annual Conference on High Integrity Language Technology, 2013.

Authors

Young-Mi Kim received her M.S. degree from the Department of Computer Science at Korea University in 2001. Young-Mi Kim is currently pursuing a Ph.D. degree with the Department of Computer & Radio Communication Engineering at Korea University, Seoul, Korea. Her research interests include formal methods, networks, SDN security, and software testing.

Myung-Hwan Park received the B.S. degree from Republic of Korea Air Force Academy in 1994, the M.S. degree from Korea University in 2000, and the Ph.D. degree from University of Minnesota in 2010, all majoring Computer Science. Dr. Park is currently an associate professor of computer science at Republic of Korea Air Force Academy, South Korea. His research interests include software testing and safety critical system.

www.kci.go.kr