Poster: Test Case Prioritization Using Error Propagation Probability*

Jeonghyun Joo, Seunghoon Yoo, Myunghwan Park
Department of Computer Science, Korea Air Force Academy
Cheongju, Republic of Korea
Email: afajhjoo, shyoo.rokafa, pigisum@gmail.com

***Abstract*—A software test is to execute the program using a test case to examine whether it produces the intended output. For successful regression testing, it is very important to choose a relatively small amount, but productive test cases to maximize testing efficiency. Test case prioritization is a suggested technique for this purpose. This technique arranges the test cases in such a way that higher-order test cases are expected to outperform those on lower-order test cases in fault-finding capability. In this paper, we suggest a new metric for test case prioritization based on the error propagation probability of the test cases. This metric arranges test cases in order by means of the probabilistic fault finding capability of the test cases. Since our metric is based on mathematical probability, it can show statistically consistent and constant results for the fault-finding capability of test cases. The experiment results show that there is a high correlation between the test cases aligned by our metric and their fault-finding capabilities.**

***Index Terms*—Test Case Prioritization, Error Propagation Probability, Lustre Language, Regression Testing**

I. Introduction

Test case prioritization [1] [2] [3] [4] [5] is one of the most challenging tasks in regression testing. In regression testing, to verify that a change to the existing software does not destroy the function of the software, it is necessary to run all existing test cases in addition to the newly developed test cases, which is very expensive and time-consuming. Thus, there is a need to reduce the test cost and time while maintaining test quality during a regression test. Test case prioritization is a suggested technique for this purpose. This technique arranges the test cases in such a way that higher-order test cases are expected to outperform those on lower-order test cases in revealing errors in the software. Several metrics have been suggested to effectively order the test cases. For example, the dominant metrics used for prioritization include the average percentage of faults detected, the source code coverage of the test cases, and the coverage of requirement.

In this paper, we propose a new metric for test case prioritization based on the error propagation probability of the test cases. This metric arranges test cases in order by means of the probabilistic fault finding capability of the test cases. Simply put, our metric places the test cases in a higher order, which are more likely to propagate errors to the output in a software. In contrast, if the test cases cause errors to be masked out before they reach the outputs, the test cases are placed in a lower order. Since our metric is based on mathematical probability, it can show statistically consistent and constant results for the fault-finding capability of test cases. The experiment shows that there is a high correlation between the test cases aligned by our metric and their fault-finding capabilities.

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).

II. Background

This section introduces the definition of terms used in this paper and Lustre language, which is the target language of our research.

A. Term Definitions

In this section, we define the terms, which are used over this paper. **Fault** refers to mistakes in the source code like typos and faulty instructions. **Error** means an erroneous value of a variable caused by the fault execution. **Error propagation** is the transfer of an error to another variable causing the affected variable to become another error. **Error masking** is failure of error propagation causing an error to disappear in other variables that are computationally connected to the error.

B. Lustre Language

Lustre [6] is a declarative and synchronous data flow programming language, on which our analysis and experiment is based. Lustre is the core language of the SCADE tool, developed by Esterel Technologies and is popular for critical control software in aircraft and nuclear power plants. Lustre adopts a synchronous paradigm, in which a system behavior is a sequence of reactions. Each reaction is to read the current inputs, update the value of variables, and evaluate the output value. The synchronous paradigm assumes that the reaction of system is instantaneous. The Lustre program consists of nodes that model the subprogram in a modular language. Variables are defined in the node and evaluate their values at each time interval (time step). Thus, the variables are modeled as a stream of values in Lustre. The variable types include both combinatorial and delay variables. The combinatorial variable is used to change the system state (value of the variable) for each clock, and the delay variable is used to store the system state for one time step. PRE operator refers to the one-step previous value of a variable.

978-1-7281-5778-8/20/$31.00 ©2020 IEEE
DOI 10.1109/ICST46399.2020.00047

III. The Computation of Error Propagation Probability

In this section, we first describe the basic concept of our approach to compute the error propagation probability for test cases. We then present formula and algorithm for the computation.

A. Basic Concept

The basic concept to compute the error propagation probability of each test case and use it as a basis to prioritize test cases is as follows. Suppose the variable v has an error in the software under test. Let o be an output variable and t be the test case t. We define the propagation probability of the error of v to the output variable o with respect to the test case t as shown in Expression 1.

$$P(v, o, t) \tag{1}$$

Thus, a higher $P(v, o, t)$ value means that test case t is more likely to force the error in v to propagate to output variable o. Conversely, if $P(v, o, t)$ is low, the error in v is less likely to propagate to o by test case t.

Expression 2 represents the probability of the error in v being masked out before it reaches o when test case t is given.

$$1 - P(v, o, t) \tag{2}$$

Let $v_1, ..., v_n$ be all variables except the output variables in software, then Expression 3 represents the probability of the errors of all variables propagating to o with test case t.

$$\prod_{i=1}^{n} P(v_i, o, t) \tag{3}$$

Similarly, Expression 4 means the probability that no errors in any variables propagate to o, given test case t.

$$\prod_{i=1}^{n} (1 - P(v_i, o, t)) \tag{4}$$

Let $o_1, ..., o_m$ be all output variables in the software. Then, Expression 5 represents the probability of the errors of all variables propagating to all output variables with test case t.

$$\prod_{j=1}^{m} \prod_{i=1}^{n} P(v_i, o_j, t) \tag{5}$$

Expression 6 represents the probability that no errors in any variables propagate to any output variables with given test case t.

$$\prod_{j=1}^{m} \prod_{i=1}^{n} (1 - P(v_i, o_j, t)) \tag{6}$$

Finally, we present the expression that computes the error propagation probability of a test case. Expression 7 represents the probability that at least one error regardless of the location propagates to at least one output variable when test case t is given.

$$1 - \prod_{j=1}^{m} \prod_{i=1}^{n} (1 - P(v_i, o_j, t)) \tag{7}$$

If a test case has a higher error propagation probability in the software, it will have better performance in catching errors than test cases with a lower error propagation probability. Therefore, error propagation probability can be used to prioritize test cases and the quality of the test case will likely be proportional to the location of the test case in the order.

B. Translation from Lustre programs to Error Propagation Graph

As noted in Section II-B, the target language of our research is Lustre, which is suitable to model reactive systems. Lustre language itself, however, is not suited to compute error propagation probability since the syntactic structure of the program cannot be easily identified. Thus, we defined a mathematical model called the error propagation graph [7] which is suitable to represent the syntactic structure of the Lustre program. We also developed a tool to automatically translate a Lustre program to an error propagation graph. Error propagation graph G is a tuple $G = (V, E, C)$ where

- V is a finite set of nodes.
- $E \subseteq V \times V$ is the set of directed edges. An edge $(n_1, n_2) \in E$ represents an error propagation from node n_1 to n_2.
- C is a clock. The time sequence of C, $t = t_1 t_2 ...$, is an infinite sequence of time values $t_i \in N$ with $t_1 = 1$ and $t_i - t_{i-1} = 1$.

While the error propagation graph looks very similar to the abstract syntax tree of the Lustre language, the nodes have additional characteristics compared to an abstract syntax tree. Each node of the error propagation graph has five attributes: value, error probability, time, data type and formula for computing the error probability. The formula is explained in the next section. Figure 1 shows an example of a Lustre program and its error propagation graph. The error source node (a fault) in Figure 1 is '-' operator node. Thus, the computation of error propagation probability starts from this node.

C. Formula for Computation of Error Probability

This section introduces the computation of the error probability for a node in an error propagation graph. Figure 2 shows formula to compute the error probability for parts of operator nodes. The assignment operator node does not have any masking effects on errors. Thus, the error probability of the child node will be completely propagated to the parent node. Arithmetic operators such as $+, -, \div, \times$ have very lower masking effects on errors. If an error exists in at least one of the child nodes, the error will propagate to the parent node in most cases. Relational operator nodes such as $<, >, \leq, \geq$ have a different error probability compared to the arithmetic operator nodes. For example, suppose x node is an error in the expression $x > y$. Even if the value of x is wrong, we are

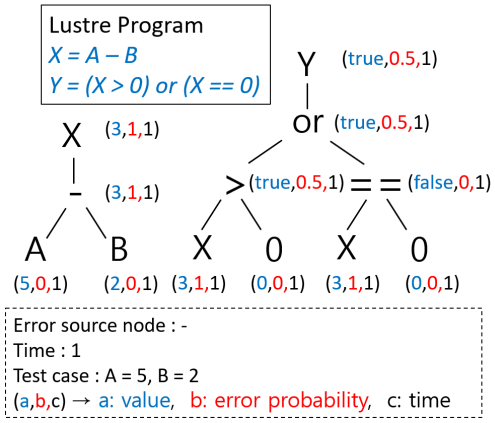

Fig. 1. Lustre Program and Error Propagation Graph

not sure that the evaluation result of the expression will be changed by the error. Thus, the probability of the evaluation result of the expression being changed (i.e., the probability of the error propagation) is 50%. Due to the space limitation, we do not cover the error propagation behavior of remaining types of nodes.

L: Left child node of A, R: Right child node of A, S: Single child node of A
$P_E(A)$: Error probability of A
$P_t(A)$: The probability that the value of node A is true
$P_f(A)$: The probability that the value of node A is false

Node A	Formula
=	$P_E(A) = P_E(S)$
+, −, ÷, ×	$P_E(A) = 1 - (1 - P_E(L))(1 - P_E(R))$
<, >, ≤, ≥	$P_E(A) = (1 - (1 - P_E(L))(1 - P_E(R))) * 0.5$
==, !=	※ L, R: numerical data type $P_E(A) = 0$ ※ L, R: Boolean data type $P_E(A) = P_E(L) + P_E(R) - P_E(L)P_E(R)$
AND	$P_E(A) = P_E(L)P_t(R) + P_t(L)P_E(R)$
OR	$P_E(A) = P_E(L)P_f(R) + P_f(L)P_E(R)$
NOT	$P_E(A) = P_E(S)$

Fig. 2. Parts of formula for computing the error probability of node A

D. Algorithm for Prioritizing Test Cases

Algorithm 1 shows the prioritization process of the test cases based on the error propagation probability of the test cases. The time complexity of the algorithm is $\mathcal{O}(A^2T)$, where A is the number of nodes in the graph and T is the number of test cases.

IV. Experiment

This section presents the goal, design, and results of the experiment we performed to check the effectiveness of our technique to prioritize test cases. We are interested in identifying how effective our approach is to predict the fault-finding capability of a test case. More specifically, we want to check whether there exists a statistically significant correlation

Algorithm 1 Prioritize test cases based on the error propagation probability of the test cases

⋆ EPG : Error Propagation Graph
⋆ EPP : Error Propagation Probability
Build EPG from a Lustre program
for each test case T_k **do**
  for each output node O_j in the EPG **do**
    for each node A_i (except output nodes) in the EPG **do**
      Set $P_E(A_i) \leftarrow 1$
      Compute $\prod_{i=1}^{n}(1 - P(A_i, O_j, T_k))$
    end for
    Compute $\prod_{j=1}^{m}\prod_{i=1}^{n}(1 - P(A_i, O_j, T_k))$
  end for
  Compute EPP of T_k : $1 - \prod_{j=1}^{m}\prod_{i=1}^{n}(1 - P(A_i, O_j, T_k))$
end for
Prioritize the test case T_k based on the EPP of T_k

between error propagation probability of a test case and fault-finding capability of the test case. To answer this question, we prepared an experiment with the following steps:

1) We chose three system models as case examples to be used in the experiment.
2) We generated mutants and test cases for each case example
3) We computed error propagation probability for each test case generated in the previous step.
4) We measured fault-finding capability of each test case by executing mutants with the test case and identifying the number of mutants caught by the test case.
5) We analyzed the degree of relationship between the error propagation probability and fault-finding capability of a test case.

A. Case Examples

For the case examples in this study, we used 3 system models written with the Lustre language. Two of these systems, Infusion and Alarm, are medical systems designed for medical research; Infusion is a prescription management system and Alarm is alarm-generating system of an infusion pump device . The other system is a Docking Example that was developed by NASA to describe the behavior of a space shuttle. All models are non-proprietary and were developed for research and teaching purposes.

B. Mutant and Test Cases Generation

We generated mutants for each case example by replacing a program construct of the correct model (oracle) into a different construct. The seeded fault (replaced construct) is type-compatible to the original one so that it would not trigger a grammar error. Each fault was categorized into one of 4 types: Logical operator fault, arithmetic operator fault, relational operator fault, and literal fault. The numbers of mutants generated for the case examples (Alarm, Docking Example, and Infusion) are 469, 435, and 451, respectively. We also generated test suites (set of test cases) satisfying MC/DC

coverage criterion over the case examples. The number of test cases generated for each case example was 656, 78, and 254, respectively.

C. Experiment Results

Figure 3, 4, 5 show the experiment result using case examples. The x-axis represents the error propagation probability of a test case and the y-axis represents a mutant kill ratio. The mutant kill ratio is the ratio of the number of mutants detected by the test case to the total number of mutants. The error propagation probability of a test case is computed with Expression 7. As shown in figures, there exists a very strong relationship between the fault-finding capability of a test case (i.e. the mutant kill ratio by a test case) and error propagation probability of the test case. Correlation coefficients between the two factors of the case examples are 0.8, 0.96, and 0.82, respectively.

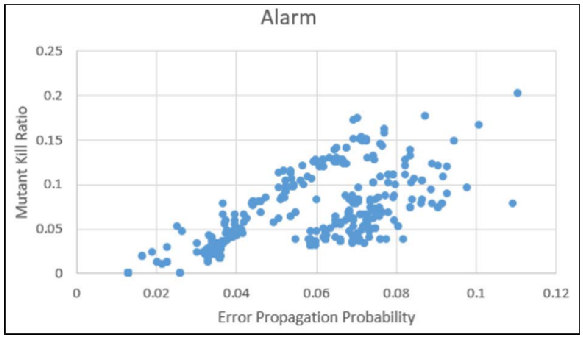

Fig. 3. Alarm

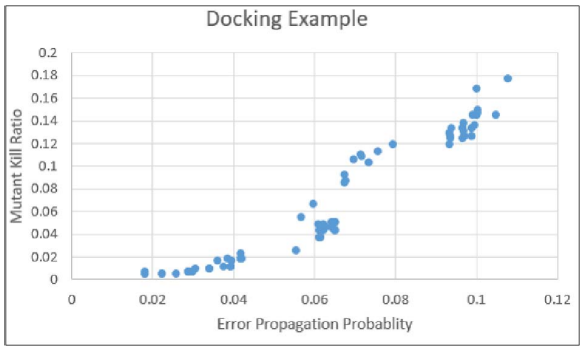

Fig. 4. Docking Example

V. Threat to Validity

We used Lustre as the basic language for the error propagation analysis. Although Lustre is not a popular language compared to more common languages, it has a similar syntactic structure with 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.

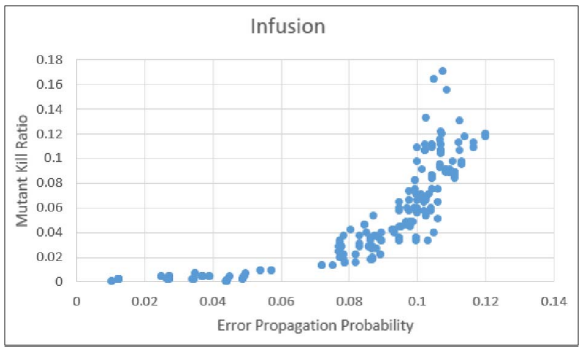

Fig. 5. Infusion

VI. Conclusion

In this paper, we proposed a new metric for test case prioritization based on the error propagation probability of the test cases. The benefit of our technique is that it can provide a statistically consistent and constant prediction for the fault-finding capability of test cases. The experiment results showed that there is a high correlation between the test case priority and its fault-finding capability. For future work, we will expand our metric to evaluate fault-finding capability of a set of test cases, not but a single test case. Individually good test cases may not show good performance collectively in fault-finding capability. This is because they can catch same errors redundantly. Since the redundant catch of the same errors does not improve the efficiency of test cases, a new metric for a set of test cases that minimizes redundant catch while maximizing test efficiency is required.

References

[1] J. Chen, Y. Lou, L. Zhang, J. Zhou, X. Wang, D. Hao, and L. Zhang, "Optimizing test prioritization via test distribution analysis", Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Pages 656–667, October 2018.

[2] Y. Lu, Y. Lou, S. Cheng, L. Zhang, D. Hao, Y. Zhou, and L. Zhang, "How does regression test prioritization perform in real-world software evolution?", Proceedings of the 38th International Conference on Software Engineering, Pages 535–546, May 2016.

[3] R. Saha, L. Zhang, S. Khurshid, and D. Perry, " An Information Retrieval Approach for Regression Test Prioritization Based on Program Changes", Proceedings of the 37th International Conference on Software Engineering, Pages 268-279, May 2015.

[4] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Test case prioritization: An empirical study", Proceedings of the IEEE International Conference on Software Maintenance, pages 179-188, Oxford, England, UK, August 1999.

[5] A. Srivastava and J. Thiagarajan, "Effectively prioritizing tests in development environment", Proceedings of the International Symposium on Software Testing and Analysis, pages 97-106, July 2002.

[6] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud,"The synchronous data flow programming language LUSTRE", Proceedings of the IEEE, vol 79, no 9, pages 1305–1320, 1991.

[7] M. Park, "An approach for oracle data selection criterion", Ph.D. Dissertation, The University of Minnesota, 2010.