# Stopping Criteria for Regression Testing in GUI Application using Failure Intensity and Failure Reliability

Chalita Somsorn and Peraphon Sophatsathit
*Advanced Virtual and Intelligent Computing (AVIC) Center*
*Department of Mathematics and Computer Science*
*Faculty of Science, Chulalongkorn University, Bangkok, Thailand*
chai_ing22425@hotmail.com, peraphon.s@chula.ac.th

*Abstract*—***We propose some criteria for GUI regression testing to determine the appropriate time to stop without wasting too much testing cost. This is essential for all software upgrades that can be released in a reasonably short time, yet still guarantees the product quality. One difficulty to achieve such a target depends on the sequence of test cases being input. The order affects the number of found failures. As such, the proposed methodology randomizes the order of test cases into different sequences for the regression test input. When a failure is found, it is edited immediately before the test resumes. Performance of the proposed criteria encompasses three measures, namely, failure intensity, cost of testing and editing, and reliability. The reliability function incorporates Weibull distribution to better reflect the test data. The proposed methodology is tested using real GUI applications as test data and shows satisfactory results on stopping criteria.***

***Keywords-GUI;regression testing; failure; reliability; test cases.***

## I. INTRODUCTION

Graphical user interface (GUI) is an important part of a software system. It makes software applications easy to use by providing a front end that receives events from users and interacting with the underlying applications through messages or method calls. Compare to traditional software systems, GUI applications have wider range of user bases which increase the chance of encountering failures and repeated requirement changes. This results in frequent code modifications that may introduce new faults. These in turn lead to new failures in already tested applications. To cope with such predicament, testing for their correctness is essential to ensure safety, robustness, and usability of the software. The process of testing a software system after changes has two main parts: regression testing for ensuring that the modifications do not affect existing software functionalities and non-regression testing for ensuring that new functionalities are implemented correctly.

The nature of GUI applications poses unique challenges for regression testing. Firstly, because GUI inputs and outputs depend on the graphical layout of components, the expected results of existing test cases may become obsolete when there are changes in input-output mapping. Secondly, in addition to technical understanding, GUI application testers are required to understand the modes of operation in order to produce failures that are not expected by the developing team. Lastly, detecting frequent code modifications and adapting the old test cases to them demand efficient testing mechanisms.

From the business perspective, releasing software early has the benefits of an earlier market introduction. On the contrary, hurriedness of releasing may lead to insufficient testing time and unsatisfactory software quality. In general, software quality depends on many factors such as the intricacy of user's requirements, algorithm complexity, level of reliability that needs to be reached, etc. Exhaustive testing, while providing the best software quality, requires too much time, cost, effort, and impractical to carry out. Thus, determining the appropriate time to stop testing is crucial for maximizing the profits from early software release and reducing the risk of inadequate software quality.

This research proposes a new method to determine when regression testing should be stopped. As each test sequence contains many test iterations where the number of iterations depends on the number of failures, the estimated failure intensity of the test cases can be measured. A number of statistics are collected, namely, failure intensity and cumulative average failure to determine the reliability of test results. The procedural details will be described in the sections that follow.

The rest of this paper is organized as follows. Section II reviews some related work. The proposed methodology is described in Section III. Section IV shows the experiment and the results so obtained. Some concluding remarks and future work are given in Section V.

## II. LITERATURE REVIEW

There are three issues pertaining to this work, namely, regression testing, GUI testing, and criteria for when to stop testing. We will look into a brief overview of each issue.

### A. Regression Testing

Regression testing focuses mainly on testing to ensure that modifications of the previous version of the application do not alter existing software functionalities. Normally, regression testing is done by rerunning old test cases. As the software system grows, the number of test cases increases tremendously. Unfortunately, only a fraction is relevant to modifications. To save time and resources, test case selection must be employed to select only the test cases that are pertinent to the modifications.

**978-1-4799-8353-7 /15/$31.00 ©2015 IEEE**

Many techniques have been proposed in the literature based on methods such as textual differencing, dataflow analysis, etc. A detailed list of regression test selection techniques can be found in [1,8].

### *B. GUI Testing*

A GUI testing method based on function diagram was proposed by Hui, et al. [9] to improve the efficiency of object–oriented software. The method compared the function diagram of the previous version of the software with the modified version to determine which test cases should be used. Memon, et al. [10] used GUI control flow graph (G-CFG) and GUI call-graph to represent the event behavior and invoking behavior of the components. The original and modified GUIs' representations were compared to detect obsolete test cases. These test cases were subsequently modified for reuse. However, constructing G-CFG of the application under test could be time-consuming for large application and therefore was not very practical in some cases.

### *C. Criteria for When to Stop Testing*

The question of when to stop testing involves many factors. Some of them are related to economic reasons, such as the cost of continued testing and the expected losses due to faults that remain. Others depend on the expected quality of the software system, such as fault detection rate, mean time between failures, the complexity and difficulty of the system, and severity of the failures that may occur.

One way to determine the appropriate stop is by quantifying the reliability of a software system. This leads to the development of models collectively known as Software Reliability Models (SRMs). These models try to estimate system reliability by fitting a theoretical distribution to failure data and use it to design stopping criteria of testing.

The followings are the assumptions used in software reliability modeling [2,3]:

(1) The software system is subject to failures at random times caused by the manifestation of remaining faults in the system.

(2) The total number of faults at the beginning of testing is finite and the failures caused by it are also finite.

(3) The mean number of expected failures during the time interval $(t, t+\Delta t]$ is proportional to the mean number of remaining faults in the system. It is equal likely that a fault will generate more than one failure and a failure may be caused by a series of dependent faults.

(4) Each time a failure occurs, the fault that caused it is completely removed and no new faults will be introduced.

From assumption (3) above, the following relationships can be derived:

$$\frac{dm(t)}{dt} = r \times (a - \alpha \times m(t)) \tag{1}$$

which, by solving boundary condition $m(0) = 0$, leads to

$$m(t) = \frac{a}{\alpha} \times \left(1 - e^{-r\alpha t}\right) \tag{2}$$

$$\lambda(t) = ar \times e^{-r\alpha t} \tag{3}$$

where $m(t)$ is the expected number of software failures at time $t$, $r$ is the failure detection rate per remaining fault, $a$ is the expected number of initial faults, $\alpha$ is the quantified ratio of faults to failures, and $\lambda(t)$ is the failure intensity function. Thus, software reliability function is defined as follows [4]:

$$R(\Delta t \mid t) = e^{-\left(m(t+\Delta t) - m(t)\right)} \tag{4}$$

where $t \geq 0$, $\Delta t > 0$. The function $R(\Delta t/t)$ represents the probability that a software failure doesn't occur during the time interval $(t, t+\Delta t]$.

It is also assumed, in an ideal situation, that fault correction during software testing process does not introduce any new faults and the reliability of the software increases as faults are uncovered and fixed. Unfortunately, in practice, it is difficult to meet the assumptions of the above ideal case.

## III. PROPOSED METHODOLOGY

In this research, a model to determine a set of stopping test criteria in order to achieve software application reliability is proposed. Several factors affecting software reliability are considered, namely, number of faults, number of failures, testing time, editing time, fault detection rate (FDR), failure intensity, testing cost, editing cost, and reliability.

A fault is defined as a mistake in the software application, and a failure occurs when the application does not comply with the specifications due to a fault or combination of faults. Testing time is the time the test team needs to execute the previously planned test cases. Editing time is the time the developing team needs to edit the software application. Failure intensity is the number of failures divided by testing time. Fault detection rate is the number of faults divided by the sum of testing time and editing time. Testing and editing costs are estimated from testing and editing time using average salary given in [5]. The outcome of this estimation is the expected cost of continuing testing which is proposed as follows:

$$Expected\ Cost = \left(C_{testing} \times T_t\right) + \left(C_{editing} \times T_e\right) \tag{5}$$

where $T_t$ is the expected testing time estimated from failure intensity function $\lambda(t)$ of Equation (3) and the failure intensity objective $F_0$, which is set to 0.01 in this study. Finding $T_t$ such that $\lambda(T_t+T_{tp}) \leq F_0$ yields

$$T_t = \left( -\frac{\ln\left(\frac{F_0}{ar}\right)}{r\alpha} \right) - T_{testing\ previous} \tag{6}$$

where $T_{tp}$ is the sum of actual testing time of the previous iterations, and $T_e$ is the expected editing time estimated from the expected number of remaining faults divided by the editing speed of the previous iteration, that is,

$$T_e = \frac{\#remaining\ faults}{v_{previous}} \tag{7}$$

The software reliability [Eq(4)] is modified to incorporate stretched exponential function known as the complementary cumulative Weibull distribution [2]. The distribution characteristics depend on the value of Weibull 2-parameter, i.e.,

the shape parameter $\beta$ and the scale parameter $\eta$. Thus, the modified reliability function becomes

$$R(\Delta t \mid t) = e^{-\eta\left(m(t+\Delta t)^{\beta} - m(t)^{\beta}\right)} \quad (8)$$

where $\beta>0$ and $\eta>0$. In this study, the proper values obtained from preliminary experiment are $\beta=0.75$ and $\eta=0.1$.

The stopping criteria are decided by one of the two conditions as follows:

1) If failure intensity $\lambda(t)$ is less than or equal to cumulative average failure intensity in current iteration, compute the expected cost using Eq(5) to determine if
   a) the cumulative cost in current iteration plus the expected cost of the next iteration are less than or equal to the threshold cost, use Eq(8) as the stopping criterion provided that
      i) $R(\Delta t/t)$ is greater than or equal to 85%, stop;
      ii) $R(\Delta t/t)$ is less than 85%, continue testing; or
   b) the cumulative cost in current iteration plus the expected cost of the next iteration are greater than the threshold cost, use Eq(8) as the stopping criterion provided that
      i) $R(\Delta t/t)$ is greater than or equal to 75%, stop;
      ii) $R(\Delta t/t)$ is less than 75%, continue testing.
2) If failure intensity $\lambda(t)$ is greater than cumulative average failure intensity in current iteration, then
   a) If the cumulative cost is less than or equal to the threshold cost, continue testing; or
   b) If the cumulative cost is greater than the threshold cost, use Eq(8) as the stopping criterion provided that
      i) $R(\Delta t/t)$ is greater than or equal to 75%, stop;
      ii) $R(\Delta t/t)$ is less than 75%, continue testing.

The proposed methodology starts with production software that involves a number of GUI screens. It is used in a preliminary test to decide the threshold cost of initial total cost and software reliability. Additional test code is added to set the stage of regression test, i.e., seeded faults are injected to be tested by selected data sets and test cases. The selection process considers how each GUI function of the software works. A set of test cases is then created based on the guidelines in [6] to comply with the software function. Since execution sequence of the test cases affects the occurrence of faults and failures, all test cases will be organized into many sequences of tests in random order. The regression test proceeds one iteration at a time for each test sequence. The first test case of the first sequence is executed. If a fault occurs, the corresponding faulty code is edited to fix the erroneous code. The second test case is then executed. This process repeats until all test cases in the first sequence are exhausted. The first regression test iteration is said to finish. Meanwhile, test statistics are collected to analyze if the test stopping criteria are met and the entire process terminates. Otherwise, the test continues on next iteration of the second sequence.

## IV. Experiments and Results

The proposed method was tested with an open-source GUI application named jsyntaxpane [7], which consisted of 99 classes and approximately 3,550 lines of code. This application is a sub-class of Java jEditorPane with added support for syntax highlighting of 22 file types. Each file type has its own lexical analyzer to serve different functionalities. Additional functionalities could also be added. Fault seeding was performed to initialize the test process and the regression test began as described earlier.

The test toolset and their environment were NetBeans IDE 8.0.2 [7] running on Windows 7 64-bit operating system with Intel(R) Core(TM) i7-3520M CPU and 8.00 GB RAM.

The version of jsyntaxpane used in the experiment contained two types of faults, namely, initial faults and seeded faults. An initial fault is an unintended fault that exists in the application before enhancement. Bug reports provided in the application project page and selected test cases were employed to uncover the initial faults. Seeded faults were added during test execution according to the average fault distribution of the software systems provided in [6]. There were 21 and 19 lines of code that contained initial faults and seeded faults, respectively. A total of 40 faults were tallied which caused 37 failures in the application. Table I summarizes the types of faults in the experiment.

TABLE I. Faults distribution in preliminary experiment.

| Type of faults | #lines | |
|---|---|---|
| | Initial faults | Seeded faults |
| **FUNCTIONALITY AS IMPLEMENTED** | | |
| Feature misunderstood, wrong | 9 | |
| Feature interaction | 4 | |
| Missing feature | 8 | |
| **STRUCTURAL BUGS** | | |
| Control logic and predicates | | 2 |
| Loops and iterations | | 1 |
| Arithmetic expressions | | 2 |
| Logic or Boolean, not control | | 1 |
| Initialization | | 1 |
| Other processing | | 6 |
| **DATA** | | |
| Other data definition, structure, declaration bugs | | 1 |
| Value | | 2 |
| Wrong object accessed | | 1 |
| Other access and handling | | 2 |
| **Total** | 21 | 19 |

Table II shows the *expected testing time*, *expected editing time,* and *expected cost* of each iteration computed from previous iteration using Equation (5). The cost is estimated in dollars ($) using average salary given in [5]. *#rem-faults, #faults,* and *#fails* are the number of remaining faults at the beginning of each iteration, the number of faults that have been corrected, and the number of failures that have occurred in each iteration, respectively. *FDR* represents the number of faults per minute. *Failure intensity* is the number of failures per minute of testing time. $\alpha$, $r$, and $\lambda(t)$ are defined earlier. $\lambda(t)$ *avg* is the average of $\lambda(t)$ from the start of each sequence. $m(t)$ and $m(t+\Delta t)$ are the expected number of failures used to calculate the *reliability* $R(\Delta t/t)$ by means of Equation (8), where $\Delta t$ is set to one year.

It can be seen that the expected testing time and expected editing time tend to overestimate the actual testing time and actual editing time. At any rate, both the expected and actual time tend to go in the same direction. The $\alpha$ calculated in each iteration is used to estimate the actual $\alpha$, which turned out to be 1.081. Meanwhile, $\lambda(t)$ gives a projection of how future failure intensity will behave. As the number of faults decreases in each iteration, the reliability increases. Note that the final value of reliability in each sequence is not equal to one another. This is because the sequence of test cases affects the number of test

TABLE II. EXAMPLES OF FAULT STATISTICS BEING COLLECTED.

| Test seq. | Rev. | Expected testing time (min) | Expected editing time (min) | Expected cost ($) | Actual testing time (min) | Actual editing time (min) | Actual cost ($) | #rem. faults | #faults | #fails | α | r | FDR | Failure intensity | λ(t) | λ(t) avg | m(t) | m(t+Δt) | Reliability |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ***1*** | 1 | N/A | N/A | N/A | 23.94 | 465.82 | 387.29 | 40 | 20 | 9 | 2.22 | 0.0094 | 0.04 | 0.38 | 0.23 | 0.23 | 7.08 | 18.00 | 0.64 |
| | 2 | 140.45 | 465.82 | 461.73 | 44.14 | 174.23 | 167.34 | 20 | 12 | 4 | 2.46 | 0.0045 | 0.06 | 0.09 | 0.06 | 0.09 | 3.16 | 8.13 | 0.78 |
| | 3 | 115.47 | 116.15 | 166.53 | 23.91 | 79.84 | 79.03 | 8 | 6 | 1 | 2.71 | 0.0052 | 0.06 | 0.04 | 0.03 | 0.06 | 0.85 | 2.95 | 0.87 |
| | 4 | 3.15 | 26.61 | 23.27 | 21.23 | 0.00 | 13.56 | 2 | 0 | 0 | 2.71 | 0.0000 | 0.00 | 0.00 | 0.00 | 0.05 | 0.00 | 0.00 | 1.00 |
| | 5 | N/A | N/A | N/A | | | | | | | | | | | | | | | |
| ***2*** | 1 | N/A | N/A | N/A | 39.76 | 370.02 | 320.90 | 40 | 26 | 11 | 2.36 | 0.0069 | 0.06 | 0.28 | 0.14 | 0.14 | 8.09 | 16.92 | 0.70 |
| | 2 | 138.88 | 199.24 | 247.84 | 27.44 | 79.84 | 81.29 | 14 | 6 | 2 | 2.46 | 0.0052 | 0.05 | 0.07 | 0.05 | 0.09 | 1.69 | 5.69 | 0.80 |
| | 3 | 111.33 | 106.45 | 156.14 | 25.95 | 45.34 | 52.78 | 8 | 0 | 0 | 2.46 | 0.0000 | 0.00 | 0.00 | 0.00 | 0.06 | 0.00 | 0.00 | 1.00 |
| | 4 | N/A | N/A | N/A | | | | | | | | | | | | | | | |
| ***3*** | 1 | N/A | N/A | N/A | 28.34 | 226.48 | 198.97 | 40 | 27 | 9 | 3.00 | 0.0079 | 0.11 | 0.32 | 0.16 | 0.16 | 6.54 | 13.33 | 0.75 |
| | 2 | 98.00 | 109.05 | 149.69 | 20.91 | 90.10 | 85.31 | 13 | 7 | 5 | 2.43 | 0.0184 | 0.06 | 0.24 | 0.09 | 0.12 | 3.25 | 5.35 | 0.90 |
| | 3 | 53.76 | 77.23 | 96.02 | 21.43 | 0.00 | 13.69 | 6 | 0 | 0 | 2.43 | 0.0000 | 0.00 | 0.00 | 0.00 | 0.09 | 0.00 | 0.00 | 1.00 |
| | 4 | N/A | N/A | N/A | | | | | | | | | | | | | | | |

iterations, the number of uncovered faults, and failures in each iteration, all of which affect the value of reliability.

The experimental results supported our hypothesis that the sequence of test case affected the reliability and failure intensity. Because test cases were generated according to software functionality, the order of the test cases was grouped by type of functions. For example, in sequence 1, the test cases were sorted according to the similarity of function and the arrangement of GUI options. The ordering was randomized and tested in a similar manner for each sequence, yielding the test statistics as shown in Table II.

From Table II, sequence 1 stopped in iteration 2 because, in iteration 1, the cumulative average failure intensity equaled the failure intensity and the sum of cumulative cost. The expected cost of iteration 2 was less than the threshold cost which was set to $600. However, reliability function in iteration 1 was less than 75% and testing continued. In iteration 2, the reliability function was greater than 75%. Applying the stopping criteria to test sequence 2 and 3, the reliability values became 1.00 and 0.90, respectively.

Since the stopping criteria depended on failure intensity, reliability and cost, the total cost was controlled not to exceed the threshold cost within budget while keeping the reliability of software at the proper level before releasing the software. This set up determines the appropriate time to stop testing.

## V. CONCLUSION

Determining when to stop regression testing is one of the most critical problems, thus finding the stopping criteria of regression testing is important. The proposed methodology not only considers the theoretical testing aspects such as failure intensity, reliability, and time, but also administrative aspects, in particular, testing cost to confine it within the allotted budget. Thus, stopping criteria become a multi-aspect issue that calls for careful considerations. Optimizing regression test techniques to arrive at minimal cost will be a challenging future work. More efficient test sequence algorithms are needed to improve the probability of fault occurrences, thereby reducing time complexity of the proposed methodology.

## REFERENCES


[1] S. Biswas, R. Mall, M. Satpathy, and S. Sukumaran, "Regression test selection techniques: A survey," *Informatica: An International Journal of Computing and Informatics*, vol. 35, no. 3, pp. 289–321, 2011.

[2] M. R. Lyu and others, *Handbook of software reliability engineering*, vol. 222. IEEE computer society press CA, 1996.

[3] C.-T. Lin and C.-Y. Huang, "Software Release Time Management: How to Use Reliability Growth Models to Make Better Decisions", in *2006 IEEE International Conference on Management of Innovation and Technology*, 2006, vol. 2, pp. 658–662.

[4] M. Xie, *Software Reliability Modelling*. World Scientific, 1991.

[5] "Software Engineer I Salary", *Salary.com*. [Online] http://www1.salary.com/Software-Engineer-I-salary.html. [Accessed: 28-Mar-2015].

[6] B. Beizer, "bug taxonomy - Otto Vinter", [Online] https://ottovinter.dk/bugtaxst.doc.

[7] "NetBeans IDE The Smarter and Faster Way to Code", *Netbeans IDE 8.0.2*. [Online] https://netbeans.org/. [Accessed: 15-Mar-2015].

[8] A.K. Onoma, W.T. Tsai, M.H. Ponawala, and H. Suganuma, "Regression Testing in an Industrial Environment", *Communicaiton of the ACM*, May1988, vol. 41, no. 5, pp. 81-86.

[9] Z. Hui, R. Chen, S. Huang, B. Hu, "GUI regression testing based on function-diagram", *2010 IEEE International Conference on Intelligent Computing and Intelligent Systems (ICIS)*, 2010: pp. 559–563. doi:10.1109/ICICISYS.2010.5658394.

[10] A.M. Memon, M.L. Soffa, "Regression Testing of GUIs", *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ACM, New York, NY, USA, 2003: pp. 118–127.