LETTER

Usage Log-Based Testing of Embedded Software and Identification of Dependencies among Environmental Components

Sooyong JEONG[†a)], ***Member***, **Sungdeok CHA**[††], ***and*** **Woo Jin LEE**[†b)], ***Nonmembers***

SUMMARY Embedded software often interacts with multiple inputs from various sensors whose dependency is often complex or partially known to developers. With incomplete information on dependency, testing is likely to be insufficient in detecting errors. We propose a method to enhance testing coverage of embedded software by identifying subtle and often neglected dependencies using information contained in usage log. Usage log, traditionally used primarily for investigative purpose following accidents, can also make useful contribution during testing of embedded software. Our approach relies on first individually developing behavioral model for each environmental input, performing compositional analysis while identifying feasible but untested dependencies from usage log, and generating additional test cases that correspond to untested or insufficiently tested dependencies. Experimental evaluation was performed on an Android application named Gravity Screen as well as an Arduino-based wearable glove app. Whereas conventional CTM-based testing technique achieved average branch coverage of 26% and 68% on these applications, respectively, proposed technique achieved 100% coverage in both.

key words: *embedded software testing, environmental modeling, test coverage*

1. Introduction

Testing of embedded software is well-known to be time-consuming and technically challenging in that multiple sensors often continuously feed inputs to software and that feedback control from actuators has an impact on subsequent inputs. Developers generate test cases based on known dependency relations, and Siegl [12] proposed how to perform dependency analysis on input/output elements. Embedded software testing techniques sometimes rely on unrealistic assumptions that inputs are independent of each other, and Classification Tree Method (CTM) [1] is such an example. Unfortunately, test cases generated based on such assumptions are often insufficient to achieve adequate quality assurance [11], and the difficulty of testing grows significantly if there are more dependencies to consider.

Two recent crashes of Boeing 737 Max 8 aircrafts [3] offer important lessons to learn. It appears that, when developing software for Maneuvering Characteristics Augmentation System (MCAS) system, Boeing did not fully understand or analyze complex dependencies among AoA (Angle of Attack) sensor malfunctions, stall warnings, movement of the fuselage, and pilot's flight decisions. In addition to several hundred lives lost in the crashes, MCAS design flaw is estimated to have caused Boeing more than $20 billion in cost.

While exhaustive testing is ideal, especially when applied to safety-critical software, it is well-known to be impractical. As a practical alternative, we propose to use usage log to identify dependency relations that are either untested or insufficiently tested in embedded software. Usage log is analogous to data collected by black boxes (or data recorders) often installed in aircrafts and vehicles. It has been traditionally used primarily for investigative purpose when analyzing what caused accidents.

One might argue that it is perhaps unrealistic to expect that usage log is available while software is being developed. One could also argue that usage log may not contain relevant and accurate enough information to enable detection of potential errors in the System Under Test (SUT).

Perhaps one might confuse the roles of usage and test logs. The former is not to determine if SUT successfully delivers the expected functionality. Rather, it is a collection of data that represent environmental inputs SUT is expected to work under. Therefore, usage log does not require the presence of expected output from SUT. It is important to understand that usage log need not be generated from SUT itself. If SUT is an enhancement on existing systems, usage log collected from an earlier system is often adequate. Logs collected from competing products can also be used if sensor inputs are compatible.

Figure 1 illustrates how the proposed approach works. First, we develop a preliminary model on behavior of each environmental component (e.g., sensor or input) based on existing information on SUT or developer's domain-knowledge (i.e., step A). If sensor input has values in multiple dimensions (e.g., accelerometer sensor inputs in x, y, and z axes), behavioral models could be developed separately.

Preliminary model might be imperfect in that some states or transitions are missing or incorrect. Using the technique published in [4], each model is first analyzed individually to ensure its correctness and completeness. Usage log contains information on time-triggered sensor inputs to embedded software, and it may reveal states or transitions that are incorrect or currently missing but feasible. (See [4] for further details.) This step results in obtaining individually complete and correct state models of environmental components, but dependency relations among environmental com-

Manuscript received April 23, 2021.
Manuscript revised July 1, 2021.
Manuscript publicized July 28, 2021.

[†]The authors are with Kyungpook National University, Daegu, 41566 South Korea.
[††]The author is with Korea University, Seoul, 02841 South Korea.
a) E-mail: kyo1363@naver.com
b) E-mail: woojin@knu.ac.kr (Corresponding author)

DOI: 10.1587/transinf.2021EDL8042

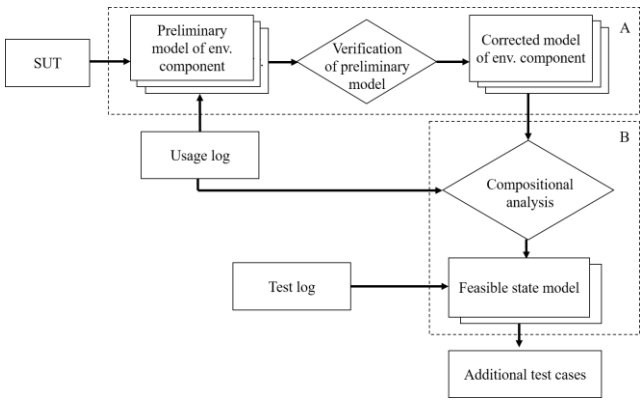


Fig. 1 The workflow of the proposed method

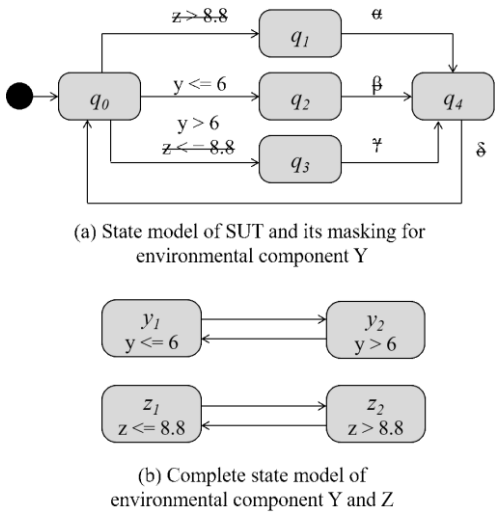


Fig. 2 State model of individual environmental component and its masking of irrelevant guard conditions and transitions

ponents have not been analyzed.

Brute-force compositional analysis is likely to result in state and transition explosion for embedded software interacting with multiple sensors and actuators. Therefore, we use information included in usage log to identify dependencies that are highly unlikely to occur in deployment and therefore safe to ignore in testing.

An important contribution of this paper is that usage log is highly useful in performing systematic dependency analysis and that usage log is an effective source in improving coverage and adequacy of existing test cases. Missing or insufficiently tested dependencies among environmental components can be identified and additional tests generated.

This paper is organized as follows. Section 2 describes each step of the proposed approach in greater detail using an illustrative example. Section 3 discusses the performance of our technique when applied to a wearable glove based on Arduino, and an Android application Gravity Screen [5]. When evaluated on hypothetical software errors seeded into software using mutation technique, our technique demonstrated superior performance when compared against the widely used CTM testing technique. Section 4 concludes the paper.

2. Usage Log-Based Testing and Identification of Dependencies among Environmental Components

2.1 Preliminary Behavioral Model of Individual Environmental Component

In order to perform systematic testing of embedded software, it is necessary to develop correct and complete environmental model that capture both behavior of each component (e.g., sensor) and dependencies among them. Development of a global state model may appear attractive, such effort is likely to be too complex and error-prone. As an alternative, we propose to develop a state model for each environmental component in isolation. During this process, some states and transitions that are known to be irrelevant (or indistinguishable) in terms of interaction with other components may become simplified (e.g., masked). For example, in the case of accelerometer used by Gravity Screen app [5], detailed and accurate behavior model of y axis input is shown in Fig. 2 (a). Behavioral model could be simplified if Gravity Screen app requires different response to inputs only around the threshold value of 6. Masking process may also remove irrelevant guard conditions in the model. For example, guard condition unrelated to y (e.g., z > 8.8) is removed.

If usage log mapping onto preliminary model detect missing transitions, they are added so that complete and correct behavioral model for each environmental component is obtained.

2.2 Compositional Analysis and Dependency Identification

Once the behavior of the individual model is validated and simplified, compositional models are automatically generated by applying a Cartesian product of the state models. While compositional models include all of theoretically possible and exhaustive combinations of cases, some states and transitions are infeasible when deployed.

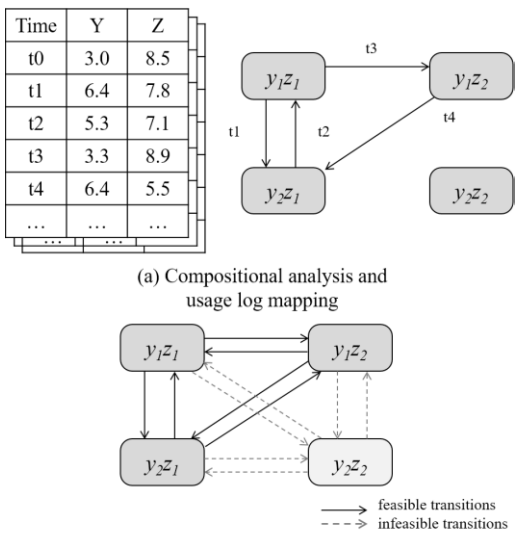
Usage log reflects the characteristics of sensor inputs the SUT is expected to encounter in deployment, and it provides useful insight on how size and complexity of compositional model can be reduced. See Fig. 3 (a) on how time-triggered usage log collected at t0 through t4 are mapped to the compositional model derived from individual models shown in Fig. 2 (b). Composite state y_2z_2 was automatically generated, but it is shown to never occur in reality. Such assertion would be safe to make only if usage log contains large and representative enough information on the environment with which SUT interacts. As shown in Fig. 3 (a), each data sequence in usage log may contain partial dependency relation among environmental components. Mapping of all data sequences in usage log would reveal all of the feasible transitions in the compositional model. This step is repeat-

edly applied to all possible combinations of environmental inputs. See Fig. 3 (b) for illustration.

2.3 Test Coverage Enhancement Based on Dependency Relation

In the final phase of the proposed approach, existing test log is mapped onto the compositional model to measure its coverage and adequacy. Each test case contains time-sequenced inputs and expected actuator outputs. If some complex or subtle dependency relations are partially implemented, existing test cases would cover only a subset of feasible states and transitions in the compositional model. If implementation is incorrect, it is also possible that transitions that were never shown to occur in usage log may suddenly appear possible. All such anomalies are to be investigated as they are indicators of potential errors in embedded software.

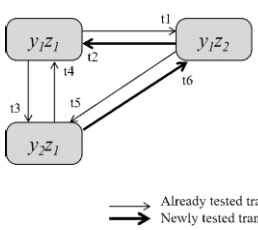
Figure 4 illustrates that existing test sequences, TS#1 and TS#2, cover all three feasible states (e.g., y_1z_1, y_1z_2, and y_2z_1) but that some feasible transitions (e.g., t2 and t6) remain uncovered. It means that additional test sequences (e.g., TS#3 and TS#4) are necessary by applying



(a) Compositional analysis and usage log mapping

(b) Compositional model Y || Z after mapping all usage logs

Fig. 3 The identification of environmental component dependency



ID	Test sequence
TS#1	t1→t5→t4
TS#2	t3→t4→t1
TS#3	t1→t5→**t6**
TS#4	t1→**t2**→t3

(a) Feasible state model of Y || Z

(b) Test sequences for Y || Z and the addition

Fig. 4 Test enhancement with additional test cases

state model-based testing techniques such as [6]. Mapping of test log onto the compositional model provides invaluable insights on which part of software further testing resource must be allocated.

One must understand that coverage value may vary depending on the chosen criteria. For example, 79% coverage of statement/branch coverage may yield only 67% coverage when expression coverage is chosen instead [9]. It is equally important to note that even 100% of chosen coverage criteria may fail to detect faults unless test cases are powerful enough to trigger generation of incorrect output [10].

3. Case Study

To investigate the effectiveness of the proposed approach, we performed testing experiments on two embedded applications and compared test coverage against the CTM-based technique. CTM-based test scenario was generated using a randomizer that continuously add test cases until it satisfies CTC_2 (pairwise combination of classification-tree coverage) criterion [2]. We generated the CTM test scenario 10,000 times and measured its performance in terms of the average, minimum, and maximum coverages. See Table 1 for details.

The first experiment used embedded software running on an Arduino-based wearable glove. The device translates American sign language into text by reading the bend of fingers with five flex sensors [13]. We collected the usage log of 6,297 entries, and the compositional analysis revealed 10 (or $_5C_2$) pairs of dependencies on finger movements between the muscles and bones. We created state models of flex sensors, generated test scenarios, and compared to those generated by the CTM-based technique. We measured how many branches of the source code out of 26 branches correspond to the alphabet 'A' to 'Z' in American sign language. Test scenarios from the proposed method succeeded in covering 100% of the branch coverage, whereas the CTM test scenario achieved only 67.5% (or 16.2 out of 26) coverage on average. The maximum and minimum state coverage measures were 92.3% and 34.6%, respectively.

The other experiment was on Gravity Screen [5], an Android application that accepts input from 8 different environmental components including accelerometer and proximity sensors. It determines when the smartphone screen is best turned on or off. To apply the proposed method, we collected the usage log of 29,218 records by Samsung Galaxy S9+ smartphone. The compositional analysis revealed that there are 14 pairs of dependency relationships among environmental components, and we were able to significantly

Table 1 Test result of experiment on Gravity Screen Android application

		Total num.	CTM Avg	CTM Max	CTM Min	Proposed method
Wearable glove	Branch Coverage	26	16.2 (67.5%)	24 (92.3%)	9 (34.6%)	26 (100%)
Gravity Screen	Transition Coverage	4,459	1,151 (25.8%)	2,399 (53.8%)	608 (13.6%)	4,459 (100%)
	Mutant kills	41	18.5 (41.2%)	28 (68.3%)	5 (12.2%)	31 (75.6%)

reduce the size of the test scenario. If we choose to perform brute-force combinatorial testing, 28 (or ${}_8C_2$) different pairs of the state models would have required testing effort. In order to perform a realistic evaluation of potential software errors, we chose to inject some errors into software and compare the effectiveness of our approach against CTM testing technique. Because source code is not publicly available, we decompiled APK file by FernFlower Decompiler [7], generated 41 mutants of the APK file with MutAPK, the automatic mutation tool [8], and measured the effectiveness of testing using the following criteria: 1) percentage of covered transitions, and 2) number of mutants killed. Each mutant represents a sample of potential errors, and a test sequence is considered to have successfully detected the error if a mutant is killed. The CTM showed that only about a quarter (or 25.8%) of transitions are covered, and only 18.5 out of 41 mutants are killed on the average. Our technique, however, successfully covered all of 4,459 feasible transitions found in the compositional model. In terms of mutants killed, our technique outperformed CTM testing technique in that 31 out of 41 mutants have been killed.

4. Conclusion

Embedded software testing poses a significant challenge to software engineers if complex and subtle dependencies exist among inputs. This is especially true if embedded software is deployed in safety-critical applications. Developers of embedded software may misunderstand the true nature of subtle dependencies. Or, developers may have underestimated the importance and significance of some of the known dependencies. Such shortcomings might result in serious quality issues with embedded software, and existing testing technique such as CTM may become inadequate.

In this paper, we proposed how usage log can offer invaluable help to developers and test engineers of embedded software. While usage log had primarily been used primarily for investigative purposes, it is highly useful during development in identifying some dependencies untested or insufficiently tested. We use usage log to obtain a complete and consistent behavior model of each environmental component, and it is also used to filter only feasible states and transitions when compositional dependency analysis is performed. Mapping of test logs on compositional models may reveal untested or insufficiently tested aspects of embedded software, and additional test cases are generated using model-based testing technique such as [6]. Experimental evaluation on different applications convincingly demonstrated that our technique is powerful and effective.

Acknowledgments

This research was supported by the Basic Science Research Program through the National Research Foundation of Korea funded by the Ministry of Education (NRF-2017R1D1A3B04035880 and NRF-2018R1A6A1A03025109).

References

[1] M. Grochtmann et al., "Test case design using classification trees and the classification-tree editor CTE," Proc. 8th International Quality Week, vol.95, 1995.
[2] K. Lamberg, M. Beine, M. Eschmann, R. Otterbach, M. Conrad, and I. Fey, "Model-based testing of embedded automotive software using Mtest," SAE transactions, no.2004-01-1593, pp.132–140, 2014.
[3] P. Johnston et al., "The Boeing 737 MAX saga: lessons for software organizations," Software Quality Professional, vol.21, no.3, pp.4–12, 2019.
[4] S. Jeong, A.K. Jha, Y. Shin, and W.J. Lee, "A Log-Based Testing Approach for Detecting Faults Caused by Incorrect Assumptions About the Environment," IEICE Trans. Information and Systems, vol.E103-D, no.1, pp.170–173, 2020.
[5] Gravity Screen – On/Off, https://play.google.com/store/apps/details?id=com.plexnor.gravityscreenofffree, accessed 05. Nov. 2020.
[6] S. Pradhan, M. Ray, and S.K. Swain, "Transition Coverage based Test Case Generation from State Chart Diagram," Journal of King Saud University – Computer and Information Sciences, 2019. doi: https://doi.org/10.1016/j.jksuci.2019.05.005
[7] FernFlower Decompiler, https://github.com/fesh0r/fernflower, accessed 25. July 2020.
[8] C. Escobar-Velásquez, M. Osorio-Riaño, and M. Linares-Vásquez, "MutAPK: Source-Codeless Mutant Generation for Android Apps," The 34th IEEE/ACM International Conference on Automated Software Engineering (ASE'19), pp.1090–1093, 2019.
[9] Y. Cheng, M. Wang, Y. Xiong, D. Hao, and L. Zhang, "Empirical Evaluation of Test Coverage for Functional Programs," Proc. IEEE International Conference on Software Testing, Verification and Validation, pp.255–265, 2016.
[10] P. Piwowarski, M. Ohba, and J. Caruso, "Coverage measurement experience during function test," Proc. 15th International Conference on Software Engineering, pp.287–301, 1993.
[11] E. Lehmann et al., "Test case design by means of the CTE XL," Proc. 8th European International Conference on Software Testing, Analysis & Review (EuroSTAR 2000), 2000.
[12] S. Siegl, M. Russer, and K.-S. Hielscher, "Partitioning the requirements of embedded systems by input/output dependency analysis for compositional creation of parallel test models," 2015 Annual IEEE Systems Conference (SysCon) Proceedings, pp.96–102, 2015.
[13] A glove that translate sign language into text and speech, https://www.hackster.io/173799/a-glove-that-translate-sign-language-into-text-and-speech-c91b13, accessed 15 Nov. 2020.