LETTER

A log-based testing approach for detecting faults caused by incorrect assumptions about the environment

Sooyong Jeong[†], ***Student Member,*** **Ajay Kumar Jha**[†], ***Nonmember***, **Youngsul Shin**[†], ***Nonmember and*** **Woo Jin Lee**[†*], ***Nonmember***

SUMMARY Embedded software developers assume the behavior of the environment when specifications are not available. However, developers may assume the behavior incorrectly, which may result in critical faults in the system. Therefore, it is important to detect the faults caused by incorrect assumptions. In this letter, we propose a log-based testing approach to detect the faults. First, we create a UML behavioral model to represent the assumed behavior of the environment, which is then transformed into a state model. Next, we extract the actual behavior of the environment from a log, which is then incorporated in the state model, resulting in a state model that represents both assumed and actual behaviors. Existing testing techniques based on the state model can be used to generate test cases from our state model to detect faults.

Keywords: *log-based testing, model-based development, incorrect assumptions, fault detection*

1. Introduction

Embedded software developers assume behaviors of the environment when specifications are not available. However, it is challenging for them to correctly assume every possible behavior. The trivial behaviors that are not assumed or assumed incorrectly may not severely impact the system. However, if all the significant behaviors are not assumed correctly, it can result in critical faults in the system, eventually leading to system failure. There are several notable cases of real-world critical software failures caused by incorrect assumptions. For example, NASA's Mars Polar Lander was destroyed because the system incorrectly identified vibrations, caused by the deployment of the stowed legs, as a surface touchdown.

There are various reasons for making incorrect assumptions. For example, the environment is so complex that unexpected behaviors may arise [1]. Furthermore, embedded system developers may overlook unexpected behaviors [2]. The aforementioned reasons indicate that making incorrect assumptions or missing to make some assumptions is inevitable. Therefore, testing embedded software system depending only on developers' assumption is insufficient to detect faults.

To solve the problem of assuming behavior incorrectly or forgetting to assume some behavior, we propose a log-based testing approach, which not only considers developers' assumptions while testing the system but also takes account of the actual behavior of the environment. Our main goal is to develop a testing technique that can take account of the actual behavior of the environment in addition to developers' assumptions. To realize the goal, we use a combination of model-based and log-based approach that can detect faults caused by incorrect assumptions.

The remainder of this letter is organized as follows. Section 2 describes our testing approach to detect faults caused by incorrect assumptions. Section 3 presents preliminary experimental results and Section 4 concludes the letter.

2. A log-based testing approach

Developers' assumptions about the environment can be represented in various formats. Currently, the environment model [3, 4] is the most popular model. It is used to represent not only the environment but also the entire embedded software system. However, the environment model is created manually [5, 6], which is a cumbersome task. Furthermore, the model is not suitable for our approach, which requires adjustment of the model according to the actual behavior of the environment represented in the log. In our approach, we use the UML behavioral model [7] to represent the environment. The behavioral model is transformed into another state model named *base model*, which is flexible enough to represent the actual environment from the log.

2.1 Base model extraction from the behavioral model

The base model is extracted from the existing UML behavioral model created by developers. The extraction is possible because UML is the de-facto standard for model-based development, which provides a range of modeling artifacts required to develop an embedded system [8], and it has the potential to be applied to complex systems such as real-time systems [9]. In the model-based development, developers insert their assumptions of the environment implicitly when they do the modeling of the embedded software.

Figure 1 shows an overview of the base model extraction approach. First, we mask all the environment components except one in the behavioral model to generate a masked model for the environment component. Then, we use the masked model to generate a state model for the

†The author is with Kyungpook National University, Buk-gu, Daegu, 41566 South Korea.
*Corresponding Author

Copyright © 20XX The Institute of Electronics, Information and Communication Engineers

environment component. We repeat the procedure until state models for all the environment components are generated. The process results in a set of state models, which is a base model of the environment.

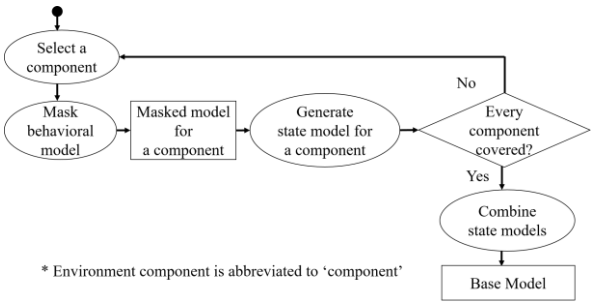


Figure 1 The overview of base model extraction

There are two major activities in transforming the UML behavioral model into a base model. One of the key activities is the masking process. The main purpose of the masking process is to represent developers' assumptions for the behavior of an individual environment component rather than representing the behavior of all the environment components. It helps to simplify the behavioral model drastically.

Masking is achieved by removing guard conditions that are irrelevant to an individual environment component. By traversing each transition in the behavioral model, we detect the guard conditions that are irrelevant to the environment component. If a transition has a guard condition that includes only irrelevant environment components, we remove the guard condition and identify the transition as a λ-transition. However, a guard condition may have a combination of the environment component for which we are masking and other irrelevant environment components. In such cases, we only remove irrelevant environment components from the guard condition. Figure 2 shows an example of the masking process. Given the behavioral model for the environment components 'a' and 'b' as shown in Figure 2(a), the masked model for the environment component 'a' is shown in Figure 2(b). The guard condition 'a>30 & b>0' is masked to 'a>30' and the guard condition 'b>0' is masked to 'λ'.

Another key activity in transforming the behavioral model into a base model is to generate state models from the masked models. We use two different features of the masked model to generate state models for each environment component. The first feature is the values of an environment component present in the guard conditions. We use the values to represent the states of the environment component. For example, as shown in Figure 2(b), the environment component 'a' has four different values in the masked model such as 'a<=30', 'a>30', '30<a<=50', and 'a>50'. Therefore, the environment component 'a' can be represented by four different states. However, some of the values are overlapping. For example, the value 'a>30' can be subsumed by the values '30<a<=50 and a>50'. Therefore, we merge the overlapping values before representing the states for the environment component 'a' as shown in Figure 3.

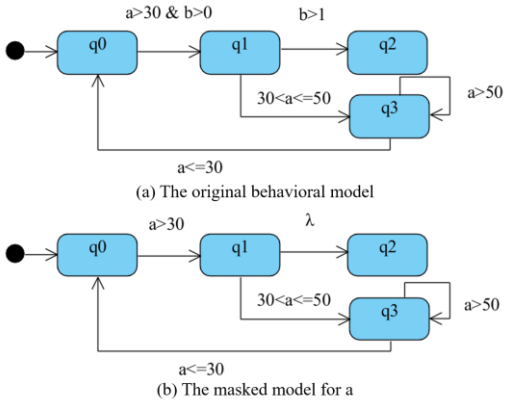


Figure 2 Masking the behavioral model for an environment component

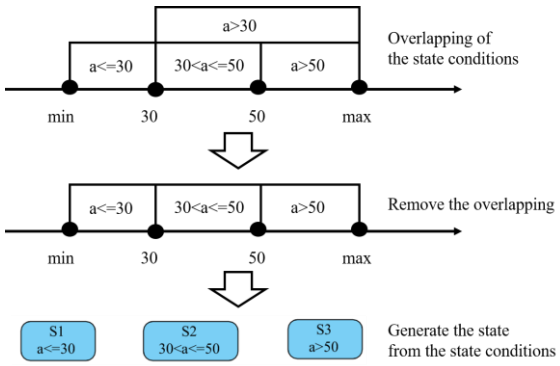


Figure 3 Generating states of environment component from the masked model for state component.

After generating states from the values of the guard conditions, we use the transitions of the masked model to generate transitions between the states to complete a state model for an environment component. For example, the masked model shown in Figure 2(b) has a path '*q1-q3-q0*'. The state *q3* can be reached from the state *q1* via '30<*a*<=50', and the state *q0* can be reached from the state *q3* via '*a*<=30'. Therefore, we can connect the state *S2* representing '30<*a*<=50' and the state *S1* representing '*a*<=30' with a transition '*S2*→*S1*' in our state model. In this way, we repeat the transition generation for each path in the masked model to get a state model for the environment component as shown in Figure 4.

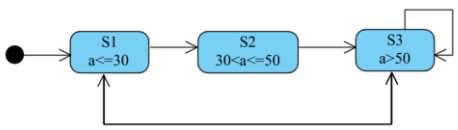


Figure 4 An example of the base model for an environment component

We repeat the process of masking the behavioral model and converting a masked model to a state model for each environment component. Finally, we combine the state models of each environment component to get the base model, which includes the assumptions for each environment component.

2.2 Adjusting base model to reflect behavior from the log

We extracted a base model from the behavioral model, which represents developers' assumptions about the behavior of the environment. If we do the black-box testing using the base model at this stage, it may not detect the faults caused by unexpected behaviors that are not assumed by developers. However, we intend to detect faults caused by unexpected behaviors too. Therefore, we adjust the base model to include the actual behavior from the log, resulting in a base model that includes not only the assumed behaviors but also the actual behaviors. In this section, we further explain the process of incorporating actual behaviors from the log in the base model.

The first step is to collect a log that contains the actual behavior of the environment. It is important that we collect the log before the implementation of the system under test (SUT). Therefore, we use the existing systems that work under the same environment as SUT to collect the log. For example, if we need to test a control software for a new model of an air conditioner, we need data of the outdoor temperature. In such a case, we can collect the data using the old model of an air conditioner or we can simply use the temperature sensor module to collect the data.

The second step is to convert the collected data to a discrete form. The environment exists continuously. Therefore, the data representing the actual behavior of the environment in the log will be in continuous form. However, the base model does not take data in a continuous form. Therefore, we have to convert the data to a discrete form. In our approach, we use the sampling technique [10] to convert the continuous data to a discrete form. We determine the sampling rate based on the characteristics of the SUT. If the interface of the SUT receives the input for a fixed period, the sampling rate will be the same as the specified period. On the contrary, if the interface of the SUT is an interrupt-driven, the sampling rate will be the maximum of the polling rate of the interface. In addition to these techniques, we can utilize the signal processing of the discrete log to get extra information about the environment components [10]. For example, autoregressive model (AR) and moving average model (MA) may provide extra characteristics of the environment components that are not expressed in raw log data.

The third step is to incorporate the sampled log data in the base model to detect the unexpected behavior of the environment component that is either not assumed by developers' or assumed incorrectly. To achieve the goal, we analyze the sampled log data. If a value in the log representing the environment component matches a state of the base model for the environment component, a pair of two sequential values representing the same environment component is retrieved from the log and matched with a transition of the base model. If the retrieved transition does not exist in the base model, we regard the transition as unexpected behavior of the environment component and we insert the transition in the base model. To distinguish the modified model from the base model, we call it *log-adjusted model*. We repeat the process for each environment component and get the environment model representing both assumed and actual behavior as shown in Figure 5.

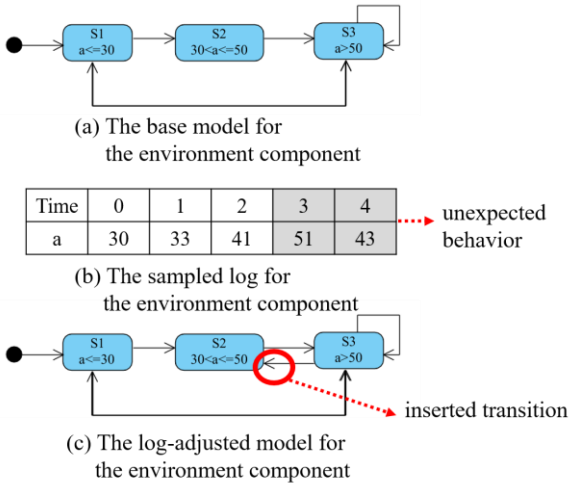


Figure 5 The log adjusting for the base model

3. Evaluation

To evaluate the effectiveness of our approach in detecting faults caused by incorrect assumptions, we conducted an experiment with an Android application named Gyroscope Explorer. In our experiment, we used the 1.5.1 version of the application [11]. As shown in Figure 6, it visualizes the rotation of an Android device using a gyroscope sensor. However, this version of the application has a fault, which crashes the application if the device is rotated to a specific position. The fault was caused by an incorrect assumption that the device will not be rotated to the specified position.

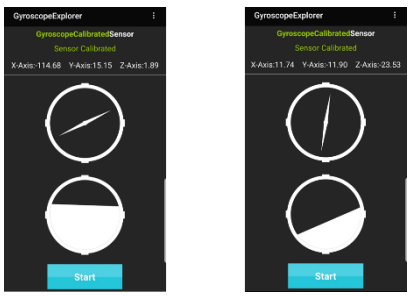
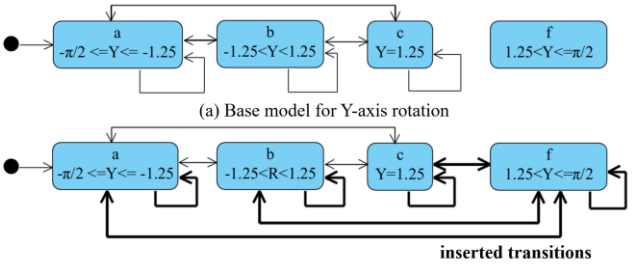


Figure 6 the screenshot of Gyroscope Explorer application

As shown in Figure 6, the white color in the circle represents the Y-axis rotation. We extracted the implicit assumptions by analyzing the source code of Gyroscope Explorer. We presumed the state model for the system from the source code, which is then converted to a base model for Y-axis rotation as shown in Figure 7(a). The base model shows that the developer did not assume the value of Y-axis rotation over 1.25 radian. To adjust the base model, we collected logs of a gyroscope, under the circumstances that

can be occurred in real life, by using Galaxy S9+ smartphone and AndroSensor [13]. Since Gyroscope Explorer updates the visualization on an interval of 100 ms, the collected logs were sampled to 19,034 items with a sampling rate of 100 ms. The sampled logs were used for adjusting the base model as shown in Figure 7(b), resulting in a log-adjusted model with additional transitions about the Y-axis rotation.



(a) Base model for Y-axis rotation

(b) Log-adjusted model for Y-axis rotation

Figure 7 The base model and log-adjusted model for Y-axis rotation.

In this experiment, we generated test cases from both the base model and the log-adjusted model using ATSA (All transition state algorithm) [14]. The base model generated total 9 test cases {(a, b), (a, c), (c, b), (b, c), (c, a), and so on} and the log-adjusted model generated 10 additional test cases {(a, f), (b, f), (c, f), (f, f), and so on}. To execute the test cases, we converted the state conditions in each state with real values that satisfy the conditions. Then, we executed Gyroscope Explorer by feeding the real values of the Y-axis rotation through sensor listener using Galaxy S9+ smartphone. We executed test cases of both the base model and the log-adjusted model and checked whether the fault was detected. The test cases involving (a, f), (b, f), and (c, f) transitions in the log-adjusted model were able to detect the fault. The exception traces of the crash caused by the fault is shown in Figure 8.

```
ronics.com.gyroscopeexplorer D/ViewRootImpl@6b296ea[GyroscopeAct
ronics.com.gyroscopeexplorer D/AndroidRuntime: Shutting down VM
ronics.com.gyroscopeexplorer E/AndroidRuntime: FATAL EXCEPTION:
lorer, PID: 10575
 be >= 0
(Bitmap.java:2177)
java:2254)
.activity.gauge.GaugeRotation.drawFace(GaugeRotation.java:329)
.activity.gauge.GaugeRotation.onDraw(GaugeRotation.java:391)
```

crash and terminate

Figure 8 the exception traces of the crash for the fault

4. Conclusion

This letter proposed a model-based and log-based testing approach for detecting faults caused by incorrect assumptions. To evaluate the effectiveness of our approach in detecting faults caused by incorrect assumptions, we performed a preliminary experiment using an open-source Android application. The result of the experiment shows that the proposed approach is effective in detecting faults caused by incorrect assumption.

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] C. Ebert, C. Jones, “Embedded software: Facts, figures, and future,” Computer, vol.42, no.4, pp.42-52, June 2009.

[2] Y. Shinyashiki, T. Mise, M. Hashimoto, K. Katamine, N. Ubayashi, T. Nakatani, “Enhancing the ESIM (Embedded Systems Improving Method) by Combining Information Flow Diagram with Analysis Matrix for Efficient Analysis of Unexpected Obstacles in Embedded Software,” Proc. 14th APSEC, Nagoya, Japan, pp.327-333, Dec. 2007.

[3] K. D. Müller-Glaser, G. Friek, E. Sax, M. K Kühl, “Multiparadigm Modeling in Embedded Systems Design,” IEEE Trans. Contr. Syst. T., vol.12, no.2, March 2004.

[4] G. Karsai, S. Neema, D. Sharp, “Model-driven architecture for embedded software:A synopsis and an example”, Sci. Comput. Program., vol.73, no.1, pp.26-38, June 2008.

[5] F. Siavashi, D. Truscan, “Environment Modeling in Model-Based Testing: Concepts, Prospects and Research Challenges,” In Proc. of the 19th International Conference on Evaluation and Assessment in Software Engineering, pp.30-35, Nanjing, China, April 2015.

[6] M. Z. Iqbal, A. Arcuri, L. Briand, “Environment modeling and simulation for automated testing of soft real-time embedded software,” Softw. Syst. Model, vol.14, no.1 pp.483-524, Feb. 2015.

[7] OMG Unified Modeling Language Specification, “https://www.omg.org/spec/UML/2.5.1/PDF”, accessed Jul. 3. 2019.

[8] T. Schattkowsky, W. Müller, “Model-Based Design of Embedded Systems,” In Seventh IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, Vienna, Austria, pp. 113-128, May 2004.

[9] M. U. Khan, K. Geihs, F. Gutbrodt, P. Gohner, and R. Trauter. "Model-driven development of real-time systems with UML 2.0 and C." In Fourth Workshop on Model-Based Development of Computer-Based Systems and Third International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MBD-MOMPES'06). IEEE, 2006.

[10] A. V. Oppenheim, R. W. Schaffer, “Discrete-Time Signal Processing,” Pearson, London, 2009.

[11] Gyroscope Explorer / Commit [4359bd] https://github.com/KalebKE/GyroscopeExplorer/commit/4359bd54d0022cc7f5049ed9f06c1d0a636bbf06, accessed Jul. 29. 2019.

[12] SensorManager | Android Developers, https://developer.android.com/reference/android/hardware/SensorManager.html, accessed Jul. 25. 2019.

[13] AndroSensor, http://www.fivasim.com/androsensor.html, accessed Aug. 2. 2019.

[14] 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:10.1016/j.jksuci.2019.05.005