Function Interaction Testing Technique for Embedded Software by Reusing Unit Test Cases

Youngsul Shin, Yunja Choi, and Woo Jin Lee

School of Electrical Engineering and Computer Science,
Kyungpook National University, Korea
youngsulshin@gmail.com, yuchoi76@knu.ac.kr, woojin@knu.ac.kr

Abstract. Software faults tend to occur when functions interact with other functions. As a tester cannot know internal information of the components, it is difficult that the tester defines test cases to find the hidden faults occurred by function interaction. Reusing test cases defined at the unit testing phase provides the tester with behavioral characteristics of each function. A test sequence to cover the reused test cases combines the behavioral characteristics of each function and may find the faults arisen by function interaction. This paper generates a test sequence to cover reused test cases in order to examine function interaction.

Keywords: Function interaction, Test case reuse, Test sequence generation.

1 Introduction

As the radiation overexposure incident occurred at the National Oncology Institute in Panama City, twenty-three of the 28 overexposed patients had died by September 2005 [1]. Functions of a software module calculate a wrong radiation dose since they provide each other with effects that developers do not intend when the software system stays on a specific state. Most faults of complicated software tend to be occurred by impacts of function interaction [2]. Therefore, reliability of the function interaction decides that of the whole system.

The existing approaches to test function interaction are not suited to find the hidden faults since they select one representative test case at random [3,4]. A function has several execution paths that produce different outputs. With one test case, a tester cannot find the faults occurred by function interaction. In order to cover diverse execution paths of each function, the tester should enter many test cases into the functions. An execution path of a function is determined by values of function parameters and global variables. Therefore, the tester should consider the function parameters and global variables in order to define test cases to execute diverse execution paths of the functions. However, the tester cannot completely analyze internal behavior of the component since the tester treats the components as black-box.

This paper proposes an approach to test function interaction by reusing test cases defined at the unit testing phase. The unit test cases provide the tester with internal behavior information of the functions. A test sequence to cover the reused test cases examines diverse execution paths of each function and may find the hidden faults occurred by impact of function interaction.

2 Related Work

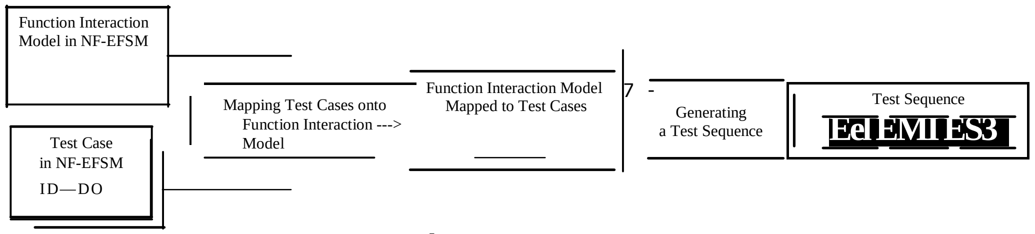
The approaches for testing function interaction are mainly based on the Finite State Machine (FSM) [5]. However, the FSM cannot represent the dynamic behavior of the component since it does not model the constraints of parameters and global variables [6]. The tester cannot use test cases to examine diverse execution paths of functions. Other approaches based on the Extended Finite State Machine (EFSM) [7] and state machine of the Unified Modeling Language [8] have the data constraints. However, the approaches based on EFSM and state machine decide the parameter values at random. The test cases generated by the approaches are not effective enough to examine diverse execution paths of the functions. Furthermore, the approaches newly define test cases to verify function interaction [9,10]. Our approach reuses test cases used for the unit testing in order to examine diverse execution paths of the functions. Comparison of the existing approaches is shown in Table 1.

Table 1. Comparison of the existing approaches.

Approach for Function Interaction	Parameter Constraints	Test Case for Diverse Execution Paths	Reusing Unit Test Cases
FSM-Based Approaches	X	X	X
EFSM-Based Approaches	0	X	X
State Machine-Based Approaches	0	X	X
The Proposed Approach	0	0	0

3 Test Sequence Generation by Reusing Unit Test Cases

This chapter describes an approach to generate a test sequence reusing test cases. As shown in Figure 1, the first step of our approach maps the test cases used for the unit testing onto the function interaction model. The algorithm based on the greedy approach produces a test sequence to time-efficiently cover all the mapped test cases.

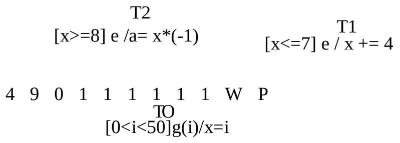


Function Interaction Model in NF-EFSM
Test Case in NF-EFSM
ID—DO
Mapping Test Cases onto Function Interaction ---> Model
Function Interaction Model Mapped to Test Cases
Generating a Test Sequence
Test Sequence

Figure 1. Procedure to generate a test sequence.

3.1 Mapping Unit Test Cases onto Function Interaction Model

The NF-EFSM model is the 8-tuple) where S is the finite set of states, s_o is the initial state, C is the finite set of global variables, $_{60}$ denotes the assignment of initial values to the global variables in C, P is the set of parameters, I is the set of input declarations, 0 is the set of output declarations, and T is the finite set of transitions. The transition t ET is defined by the 5-tuple $(_{sog}r_{i,gD,a,\$})$ in which s_s, is the source state of t, g_i is the input guard, $g_{r)}$ is the domain guard, a is the sequence operation, and s_t is the target state of t. The g_1 is expressed as the 3-tuple *(i,Pi,gp)* in which i E I U *{NIL},* E *P,* and g_p^1 is the parameter guard. A NF-EFSM model with the domain guard cannot generate feasible test sequences. To remove the domain guard, our approach uses the state split method proposed by Hierons, R.M. et al [11]. The domain guards in Figure 2 are deleted and transformed to the model without the domain guard by the state split method as shown in Figure 3.



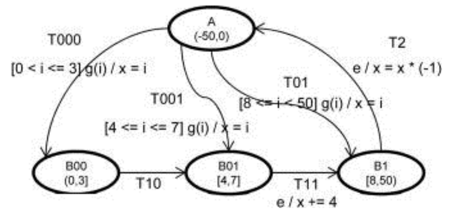
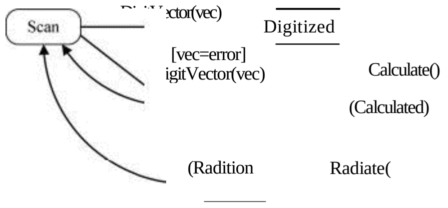
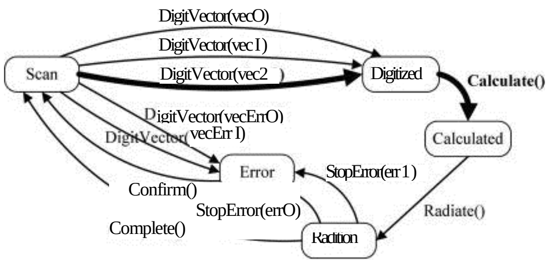

A
(-50,0)
T000
[0 < i <= 3] g(i) / x = i
T2
e / x = x * (-1)
T01
T001
[8 <= i < 50] g(i) / x = i
[4 <= i <= 7] g(i) / x = i
B00
(0,3]
T10
B01
[4,7]
T11
e / x += 4
B1
[8,50)

Figure 2. Model with the domain guard. **Figure 3.** Model without the domain guard.

A function *dom()* takes a state in NF-EFSM and returns the domain of the global variables that the state encodes. The rule to map test cases on the function interaction model uses the principle of state recognition in which a state is distinguishable from the other states using the domain of the global variables which is encoded by each state since *dom(s) n dom(s')= QS ,* where *s, s' c S, s # s'* in NF-EFSM. Precondition, postcondition, parameter values of a unit test case in source code is transformed to source state, target state, and parameter values of a trigger in NF-EFSM. Given a test case mi= *,C',o-', ,P' , I' ,0' ,T')* where t'E *T'= (sc , g', , g' , a' ,s'$_t$* *l*and a function interaction model M= *(s, s$_o$,* c,0-0, *p,/, 49,T)* where *t E T* the test case *M'* mapped on *M* should satisfy following conditions:

- *dom(s'$_s$) dom(s$_s$)*
- a trigger *i'* of *g'$_1$* is identical with a trigger *i* of *g$_r$,*
- *dom(gy)gdom(g^l),* and
- *dom(s'$_i$)c dom(s)*





(a) Function interaction model (b) Test case mapping

Figure 4. Function interaction model of the radiation therapy software and test case mapping.

The function interaction model of the radiation therapy software is shown in Figure 4 (a) and eleven test cases are mapped on the model as shown in Figure 4 (b). Once the two test cases that are represented by bole lines are executed sequentially bold lines, the interaction fault to recommend twice the necessary radiation dose is arisen.

3.2 Generating a Time-Efficient Single Test Sequence

After the test cases are mapped on the model, a test sequence to cover all the reused test cases is generated by the greedy test sequence (GTS) algorithm using Floyd's algorithm. The GTS algorithm understands the NF-EFSM model as a directed graph *G=(V,E).* Test sequence generation is a problem to find a path that covers *Ec g E,* where

is a set of edges mapped to test cases. The problem has a property that *G* is a directed and weighted pseudograph where the weight of an edge is time to spend for its execution. The GTS algorithm is given in Figure 5. First, reset edges are inserted from every vertex to the initial vertex on the graph *G* in order to cover the test cases with a single test sequence. The GTS algorithm generates the minimum-weighted adjacency matrix *D* of the graph *G[E,J=(V,Ed.* It finds the closest edge $_{ec}$i E E$_{,}$ from the current vertex using the matrix *D.* Once all the edges of *E,* is covered, the algorithm terminates. Otherwise, it repeatedly finds the closest edge. The existing methods focus on producing optimal test sequences using the Rural Chinese Postman (RCP) problem that has been shown to be NP-complete for the most general case. The GTS algorithm solves the problem approximately. Approximation algorithms are not guaranteed to yield optimal solutions, but rather yield solutions that are reasonably close to optimal in short time [12].

```
Generation of a Single Test Sequence
Problem: Determine a path to cover specified edges.
Inputs: a graph G = (V , E ), a set of specified edges
E Outputs: TS which is the path to cover E, begin

Add reset edges to G;
Add the weight of context variable verfication to weight(e E E);
minAdj = MINADJ(G,E,);
D = FLOYD(minAdj);
curr = initial vertex;
while(E, z 0){
    e'; CLOSEST(curr,E„D);
    TS = TS xINTERM(curr,source(e0)x{e0;
    curr = target(e'c);
    Ec = Ee — fe0;

return TS;
end
```

Figure 5. The GTS algorithm.

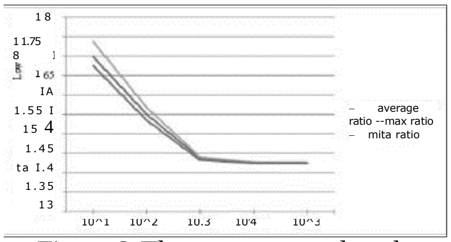


Figure 6. The test sequence length.

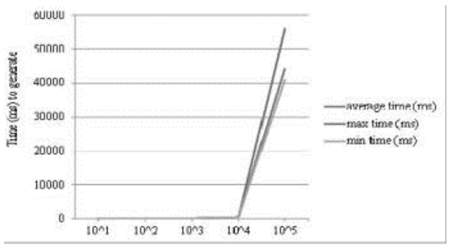


Figure 7. The test sequence generation time.

3.3 Evaluation of the GTS algorithm

We generate test sequences for the *Initiator* process of *Inres* [13]. The length of the generated test sequences is approaching 1.4 times of the minimum bound length as shown in Figure 6 of which x-axis is the number of test cases. Actually, as the optimal length cannot be obtained, we use the minimum bound length that is the length of an ideal test sequence. The generation time of the test sequences rapidly increases from 10,000 test cases as shown in Figure 7. However, the generation time for 100,000 test cases is below 60,000 milliseconds. An interaction fault of the model in Figure 4 (a) is found by reusing the test cases as shown in Figure 4 (b). When the test cases represented by bold lines of Figure 4 (b) are sequentially executed, the hidden fault is arisen. Without test case reuse, it is difficult that the tester defines significant test cases to find the hidden fault.

4 Conclusion

This paper presents the function interaction testing approach to generate a time-efficient single test sequence for high-confidence software by means of reusing test cases defined at the unit testing phase. The mapping rule connects test cases with the function interaction model using the principle of state recognition. The case study of the *Initiator* process shows that the GTS algorithm efficiently generates a test sequence to cover all the reused test cases. The generated test sequence can finds hidden faults occurred by function interaction.

Acknowledgement

This work was supported by the IT R&D program of MKE/KEIT. [10041145, Self-Organized Software-platform (SOS) for welfare devices].

References

1. Borras, C.: Overexposure of Radiation Therapy: Problem Recognition and Follow-up Measures. Pan American Health Organization, pp.173-187 (2006)
2. Edwards. S.H.: Black-Box Testing Using Flowgraphs: An Experimental Assessment of Effectiveness and Automation Potential. Software Testing, Verification and Reliability, vol. 10, pp. 249--262 (2006)
3. Fummi, F., Sciuto, D.: A Hierarchical Test Generation Approach for Large Controllers. IEEE Transactions on Computers, vol. 39, pp. 33-38 (2000)
4. Cho, J., Choi, S., Chae, S.I.: Constrained-Random Bitstream Generation for H.264/AVC Decoder Conformance Test. IEEE Transactions on Consumer Electronics, vol. 56, pp. 848-855 (2010)
5. Masod, A., Ghafoor, A., Mathur, A.: Conformance Testing of Temporal Role-Based Access Control Systems. IEEE Transactions on Dependable and Secure Computing, vol. 7, pp. 144--158 (2010)
6. Lai, R.: A Survey of Communication Protocol Testing. The Journal of Systems and Software, vol. 62, pp. 21--46 (2002)
7. Chen, Y. Probert, R.L., Ural, H.: Model-based Regression Test Suite Generation Using Dependence Analysis, Proceedings of AMOST, pp. 54--62 (2007)
8. Object Management Group: OMG UML Superstructure, (2011)
9. Halle, S., Bultan, T., Hughes, G., Alkhalaf, M., Villemaire, R.: Runtime Verification of Web Service Interface Contracts, vol. 43, pp. 59--66 (2010)
10. Kansomkeat S., Rivepiboon W.: Automated-Generating Test Case Using UML Statechart Diagrams, Proceedings of SAICSIT, pp. 29--300 (2003)
11. Hierons, R.M., Kim, T.H., Ural, H.: On the Testability of SDL Specifications, Computer Networks, vol. 44, pp. 681--700 (2004)
12. Neapolitan, R., Naimipour, K.: Foundations of Algorithms Using C++ Pseudocode, second edition. Jones and Bartlett Publishers, London (1998)
13. Hogrefe, D.: OSI Formal Specification Case Study: the Inres Protocol and Service, Technical Report IAM-91-012, pp. 5 (1991)