KIPS Tr. Software and Data Eng.
Vol.6, No.9 pp.419~428 pISSN: 2287-5905

https://doi.org/10.3745/KTSDE.2017.6.9.419

Fault Injection Based Indirect Interaction Testing Approach for Embedded System

Muhammad Iqbal Hossain† · Woo Jin Lee††

ABSTRACT

In an embedded system, modules exchange data by interacting among themselves. Exchanging erroneous resource data among modules may lead to execution errors. The interacting resources produce dependencies between the two modules where any change of the resources by one module affects the functionality of another module. Several investigations of the embedded systems show that interaction faults between the modules are one of the major cause of critical software failure. Therefore, interaction testing is an essential phase for reducing the interaction faults and minimizing the risk. The direct and indirect interactions between the modules generate interaction faults. The direct interaction is the explicit call relation between the modules, and the indirect interaction is the remaining relation that is made underneath the interface that possesses data dependence relationship with resources. In this paper, we investigate the errors that are based on the indirect interaction between modules and introduce a new test criterion for identifying the errors that are undetectable by existing approaches at the integration level. We propose a novel approach for generating the interaction model using the indirect interaction pattern and design test criteria that are based on different interaction errors to generate test cases. Finally, we use the fault injection technique to evaluate the feasibility and effectiveness of our approach.

Keywords : Indirect Interaction, Interaction Testing, Test Case Generation, Fault Injection, Embedded System

임베디드 시스템의 결함 주입 기반 간접 상호작용 테스팅 기법

Muhammad Iqbal Hossain† · 이 우 진††

요 약

임베디드 시스템에서는 모듈간의 상호작용으로 데이터를 주고 받는다. 이때 오류가 포함된 리소스 데이터를 전달하면 시스템의 실행 오류를 유발할 수 있다. 상호작용에 활용되는 리소스들은 모듈간의 의존관계를 만들며 의존관계에 있는 모듈의 변화가 다른 모듈의 기능에 영향을 미치게 된다. 몇몇 임베디드 시스템 조사 자료에 따르면 모듈간의 상호작용 오류가 심각한 소프트웨어 실패의 원인이 되기도 한다. 상호작용 테스팅 단계에서는 이러한 상호작용 오류를 검출하여 시스템 실패의 위험을 낮추고자 한다. 모듈간의 상호작용은 직접 또는 간접적으로 일어난다. 직접적인 상호작용은 모듈의 직접 호출을 통해 이루어지며, 간접 상호작용은 직접호출 이외에 리소스 데이터의 의존성을 통해 나타난다. 이 연구에서는 직접 상호작용에 의한 오류 검출 방식에서는 발견되지 않는 간접 상호작용과 연관된 오류를 검출하고자 한다. 먼저 상호작용 패턴을 분석하여 상호작용 모델을 생성하고 이를 기반으로 상호작용 오류를 검출하는 테스트 케이스 디자인 방법을 제안한다. 마지막으로 결함 주입 기법을 이용하여 제안된 방법의 효용성 및 실효성을 분석한다.

키워드 : 간접 상호작용, 상호작용 테스팅, 테스트 케이스 생성, 결함주입 기법, 임베디드 시스템

1. Introduction

From complex safety critical systems like automobile, medical system to home appliances, cellular phones even toothbrush is controlled by embedded software. So testing embedded system became a serious concern in product development life cycle. A study commissioned by the National Institute of Standards and Technology found that software errors cost the US economy $59.5 billion in every year. The study estimated over one-third of that amount, $22.2 billion, could be eliminated by improving test techniques [1]. An embedded system is a combination of processors, sensors, and actuators which have intensive interaction with resources. Also, development procedure is

※ 본 논문은 2017년도 정부(교육부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임(No. NRF-2017R1D1A3B04035880).
† 준 회 원 : 경북대학교 컴퓨터학부 박사과정
†† 정 회 원 : 경북대학교 컴퓨터학부 교수
Manuscript Received : March 17, 2017
First Revision : June 12, 2017
Accepted : June 17, 2017
* Corresponding Author : Woo Jin Lee(woojin@knu.ac.kr)

complex and changes to software interfaces and hardware are common, which makes the testing challenging. Embedded system comprises several modules and exchange resource data by interacting among themselves. As a result, any changes in resources by one module affects the functionality of another module. Therefore, interaction testing is an essential phase to reduce the interaction faults and to minimize the risk. Interaction faults are generated by the direct and indirect interaction between modules where the direct interaction is made through interfaces and the indirect interaction is made underneath of interface in which data dependence relationship with resources may cause a different outcome. Therefore, it is necessary to verify the correctness of each indirect interaction.

Several investigations of aerospace problems show critical software failures in aerospace missions are caused by functional interactions among components and incomplete specifications. Lutz examined 387 software errors uncovered during integration and system testing of the Voyager and Galileo spacecraft [2]. In the North Eastern United States, one of the biggest power blackouts in history happened on August 14, 2003. While the causes of this blackout were nothing to do with a software bug, it could have been averted. Two parts of a system were competing over the same resource and could not resolve the conflict, which indirectly caused the alarm system to freeze and stop processing alerts [3]. In 2014, automobile company Honda recalled 175,000 hybrid vehicles in Japan for a software problem. A software glitch in engine control module causes sudden acceleration in cruise control module [4]. All possible direct interaction scenarios between engine control and cruise control were tested but changes of resources by engine control module indirectly affected the cruise control unit resulting into an error. In all these cases, problems can be avoided if we put more effort on testing the indirect interaction between modules. In high-risk sectors, embedded systems need to go through a rigorous testing process. At first, each software module is tested separately as a unit and then combined to proceed for integration testing. The integration testing has the goal of proving whether developed features work together well enough for the software to submit for system testing. When combining all modules together, errors can emerge from their interactions. We still rely on traditional black box testing or genetic algorithm based approaches capable of finding particular faults caused by direct interaction but faults emerged by indirect interaction are difficult to find due to the lack of standard pattern and model.

In this paper, we propose a noble approach to generate an interaction model and then investigate several kinds of indirect interaction that causes errors through shared resources, file, device etc. denoted as interacting variable throughout this paper. The main contributions of this paper are:

- Generate an interaction model and categorize fault type.
- Specify abnormal indirect interaction pattern.
- Evaluate our approach through fault injection techniques widely used to validate the testing model.

By fault injection technique, we like to show how existing approaches incompetent to find the faults which are generated by indirect interactions.

2. Related Works

There was a few work on integration testing of the embedded system, which considers the internal behavior of the system but lacks a standard model. Most of the existing integration testing methods such as Genetic algorithm method, coupling based method, decision table method, variable strength array, verification pattern etc. define test cases from software specifications and did not consider internal execution paths of integrated modules for detecting function interaction faults. Fault injection/Mutation-based technique was used to evaluate a test approach. Many researchers discussed several faults that can be generated during integration testing but none of them are related to indirect interaction faults. An integration error occurs when an incorrect value is passed through a unit connection in [5]. They illustrated how incorrect values entering and exiting a unit call and causes erroneous output. Here, only actual parameter, global variable, and return value are considered. One of its weakness is that it was a mutation operator based technique and imposes a higher cost as every location in the program where the global variable used/defined is a potential location for mutation. This paper introduced an improved, simple and easy technique of interface faults insertion using AspectJ for Java component-based applications [6]. The technique can ignore the entire execution of an interface service, corrupting its input values and returning a bogus return value. The faults are focused on the interface that can be invoked in different ways and would lead to different event executions. Also, there is no control over when the fault should be triggered because faults are triggered by the program itself, whenever the program calls the interface services. This work proposed a fault injection strategy to test the interaction among components [7]. For that reason, interface faults were introduced by corrupting input data as well as interface output data. Even though almost every researcher focused on interface information and generate faults according to the input and output of the module, erroneous or incomplete interface specifications may lead to futile faults. We need special faults that occur during interaction among modules, which could not be found by analyzing the interface information.

A Coupling-based testing technique is proposed here [8] and used Mistix program, a UNIX file system, as a case study which does not have any call, stamp data/control, or external coupling also it is unknown how the technique will behave in the more complex system. 21 faults are inserted into Mistix, which does not reflect the integration/interaction relationship of modules. This survey paper in [9] identifies one of the major challenges in integration testing in component-based software engineering was identifying the dependencies. The author investigates how to observe system' s dynamic behavior in component integration testing. Here components are treated as a black box and observe their interrelationship by statements, execution sequence, glued parameter etc. Here, only basic interaction is observed and their method cannot find the indirect interaction among components.

3. Indirect Interaction

Embedded system encompasses a broad range of hardware and software system where the software system is divided into several modules, which are developed by several vendors or different developer team. An interaction take place when two or more modules have a calling relationship among them or while accessing same resources by several modules. Although some researcher uses the same term to classify feature interaction, human-computer interaction, interaction testing etc. which are quite distinct from our work. For example, the interaction testing focused on how components interact each other by changing the combination of components. Suppose there are four components, each with three different values, resulting in 81 possible system configurations. Each of the system tests must be run in each of these 81 configurations in order to detect any unexpected interaction faults that will occur between components [10]. A feature interaction is a situation in which two or more features exhibit unexpected behavior that does not occur when the features are used in isolation. Several approaches can be used to implement features cohesively in order to be able to compose them in different combinations [11].

According to the interaction relation, we divide them into direct and indirect interaction. Direct interaction is the explicit call relation between modules where callee module provides all input, output, and other reference information to the caller module. On the other hand, in indirect interaction, reference or resource sharing information is not present in module interface but accessed inside the body of the module where possible errors can occur. For example, in the embedded system shared variable, file, external device etc. are used extensively inside a module where caller module has no information about those. As a result, there creates an indirect interaction between two modules which access that particular resource or reference separately. Any change or error in that resource affects all the accessing modules and may open a path for unauthorized access to the resources. The main difference between integration testing and interaction testing is that in integration testing, data transaction is visible such as parameter (variable, file, memory) return value etc. but in interaction testing, data transaction in not visible from the abstract viewpoint of the system.

3.1 Formal Model for Indirect Interaction

The interaction between modules is done by clearly defined and documented interface through a parameter or return value and most of the existing works focused on faulty message/data passing through modules. The functional interface contains the necessary information to interact with another module. Most of the time interfaces are not well documented and only contain direct interaction information, not an indirect one. Finding indirect interaction is a complicated task because of lack of standard pattern and model. An indirect interaction visualized as the exchange of resources among modules, and resources usually shared between modules indirectly through files, shared variables, I/O devices, where any changes to a resource by one module may affect another module. We represent both direct and indirect interaction using an interaction model generated by extending call graph in Fig. 1.

An indirect interaction can be described as a hidden dependency between two modules through several kinds of resources where any change in one resource by a module affects the behavior of another module. At first, a

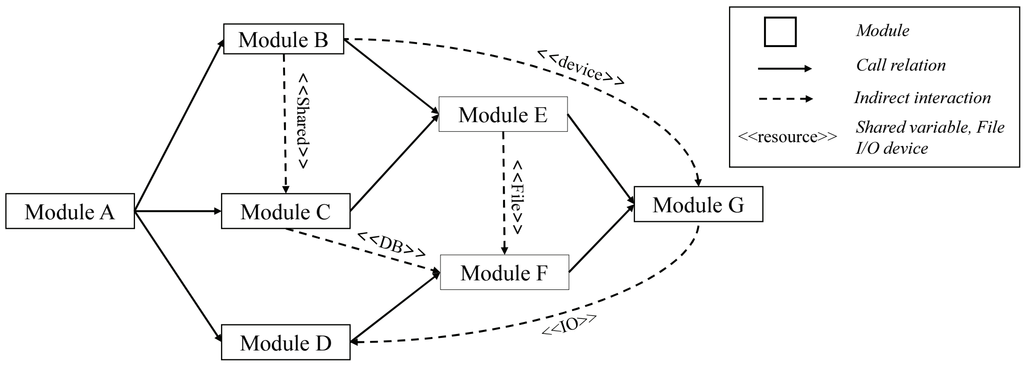


Fig. 1. Interaction Model Representation

call graph is generated automatically using the static analyzing tool and then find the indirect interaction between modules. Fig. 1 represents module A and module B have a call relation, module B, and module C have an indirect interaction by the shared variable, module E and module F have indirect interaction through a file and so on. The directed edges represent the calling sequences of the modules. The interaction model is first introduced in our prior work which uses data flow which is used for testing embedded system [12]. A formal definition of indirect interaction is given below.

Definition 1: Interaction model

Interaction model is represented as G = (V, E) comprising finite set of modules, called nodes V and a set of interactions, called edges E, where $E \subseteq V \times V$. Solid edges represent call relation and dashed edges represent indirect interaction where indirect interaction is the set of {Shared variable, File and Device} and directed edges represent the calling sequences of V.

3.2 Abnormal Scenarios by Indirect Interaction

We have identified three basic types of interactions, which are designated as test adequacy criteria, causes indirect interaction (IDI) error. Each of the types is described in details here.

Case I: Indirect Interaction by Shared Variable

In an embedded system, especially in the interrupt service routine (ISR), memory management unit (MMU), task management unit (TMU) etc. use shared variable to communicate among them and related modules. Shared variable makes data available from one module to another or among multiple processes, but has no call relation. It is very difficult to identify this interaction because shared data information is not present in module declaration. It can easily be defined and used in several modules. Any error or change of shared variables in one module affects another module. Therefore, it is essential to trace shared variables and confirm their correctness. The value of a shared variable while exiting the first module and after entering the second module need to compare to avoid value or type mismatch. It is done to make sure that these is no intermediate modification of the value. We use data flow based testing technique to find all definition use information of a shared variable and generated test paths. Any faults in data flow will be resolved by it.

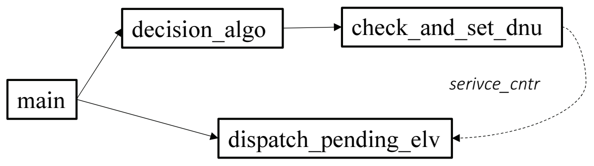


Fig. 2. Indirect Interaction by Shared Variable

For example, Fig. 2 shows the shared variable in elevator system where service_cntr is a shared variable defined and used in check_and_set_dnu and dispatch_pending_elv modules.

Case II: Indirect Interaction by File

Many embedded systems have a block of non-volatile RAM of which the kernel can maintain no memory page descriptor to mount a read/write filesystem. In addition, some embedded OSs provide memory management support for a temporary or permanent file system storage scheme. Usually, file is used to get input into a program or to display/store data from a program. MMU processes a file for temporal/permanent storage of data, which can be read, write or append by several modules. A module can open a file anywhere in its body and perform required actions without passing file information through the parameter of a module interface. Therefore, the tester does not test how files processes inside modules. However, it is very important to test how the files are processing or whether the files are performing according to speci-fication. While interacting, it is needed to test whether two modules follow that same file content structure or not. For example, a file may contain an integer value instead of a floating number. So, while reading an integer value from a file, although the file contains a float value, produces an error. There can be cases where the file system is empty or required file is not present in a directory. For this reason, these abnormal cases during interaction should be tested.

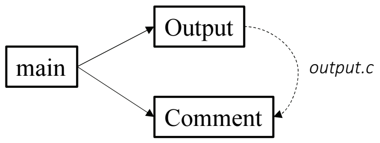


Fig. 3. Indirect Interaction by File

For example, Fig. 3 represents indirect interaction using the file in a project called "simulating a preprocessor using file," Here dataStr.c file is read in output module and write in comment module.

Case Ⅲ: Indirect Interaction by I/O Device

Embedded system contains extensive application running on different devices and these are used to receive data into a program or to transmit output data from a program. For example, in microwave oven system, the door sensor and heating elements interact with its software system and execute according to their operations. This device corresponds to a real world physical object that interacts with the system via sensors and actuator. A module can enable any sensors and actuator anywhere in its body and perform required actions. It is not needed to pass device information through a parameter. Therefore, the tester does not test how devices process inside modules. However, it is very important to test how the devices are being processed or whether the devices are performing according to specification. There can be a wrong state, timing failure, fault handling etc. problems while a device, which may lead to critical error, does the interaction.

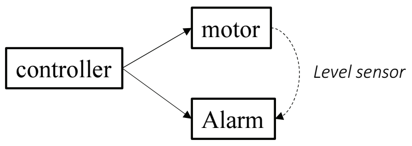


Fig. 4. Indirect Interaction by Device

For example, Fig. 4 represents indirect interaction using the level sensor in a water level monitoring system. Here, level sensor continuously read water level to start/stop the motor and in particular level, it triggers an alarm.

4. Proposed indirect interaction based approach

Our proposed indirect interaction approach comprises two phases. In the first phase, we find interacting variables between two modules by generating interaction model and define different faults by the interaction of resources. In the second phase, we use fault injection technique to verify the correctness of each indirect interaction.

4.1 Interaction Model Generation

At first, using source insight tools [13], the source code is parsed and maintained in a database to store symbolic information dynamically to generate a call graph. A list of modules and arcs i.e. caller-callee relationship between two modules are acquired by generating a call graph. Extraction of the interacting variable is a manual process, which can be done by developer or tester by analyzing source code. Many techniques use interface information to find the interaction, which can be erroneous or incomplete, and several works have already done testing this kind of interaction. Our focus is on the resources accessed by two modules inside their body, which are not present in interface information. Overview of finding interacting variable is shown in Fig. 5.

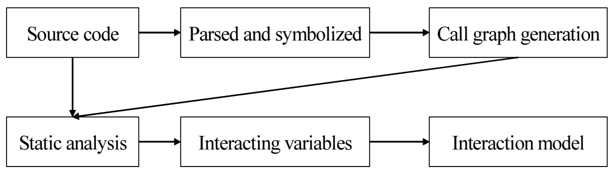


Fig. 5. Overview of Finding Interacting Variable

4.2 Fault Injection Technique

Fault injection technique is described as intentional injection of a failure condition into a running system during a test activity, to determine whether the system reacts well to off-nominal or exceptional conditions [14] [15]. Faults that injected into the system represent the actual faults that occur within the system. A tester creates a list of faults and injects those faults into the

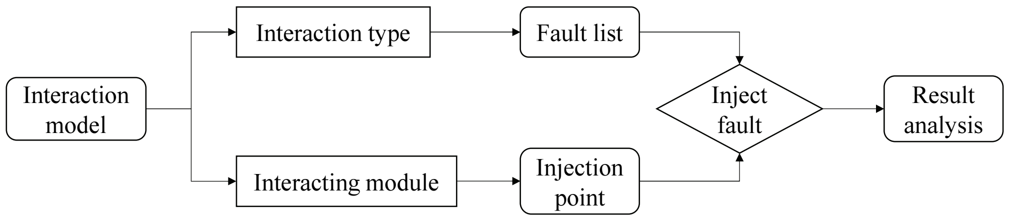


Fig. 6. Overview of Fault Injection Technique

system. The final report sent to the developer to correct the code so that faults can be handled correctly. To inject fault in the source code, we may modify the code, add new code or delete part of the code. Fault injection process is divided into two parts.

1) Pre-injection analysis

The pre-injection analysis involves creating the fault according to test criteria. Test criteria is based on the behavior of interacting variables, software design, and experience of a tester. A tester should have proper knowledge of the source code and a clear idea of where and how the fault might take place.

2) Inject actual fault

After completing the analysis, faults are injected in specific place, which is accessible by the system and execute it. A tester observes the behavior of the system and compares with previous output. Faults have so many varieties [16] that we cannot study every kind of their impact on software so we select most relevant faults which may produce by indirect interaction and the list of faults is given in Table 1.

We use fault injection technique to evaluate our approach by finding mutation score. Mutation score is used to measure the quality of a test suite detecting the introduced faults in the mutants. The main idea behind this is to observe how newly defined errors by our criteria are discovered. In the call-based technique, no error will discover without indirect interaction. The overview of the technique is given in Fig. 6 and the steps are given below:

Step 1: Generate interaction type and interacting modules from the interaction model.

Step 2: According to the type of interaction, we select possible faults from fault list. As we have already discussed that faults generated by indirect interaction, which is not studied yet. There are some existing works, discussed in related works, but does not contain standard model or representation. We have analyzed indirect interaction and make a list of errors, which can produce during run-time in the previous section.

Table 1. Different Faults by Interaction of Resources

Type	Fault description
Shared variable	Conjugative definition of the modular variable
	Last value of first module is not equal to first value of second module
	Different type in different module. Integer type in the first module and float type in the second module.
	Shared variable exceeds the boundary value leaving or entering module
File	File removed in between two modules
	File data mismatch between modules
	File modified incorrectly in one module
	Required value is not present in file
	Garbage value handling
I/O Device	Interacting device not found
	Wrong device connected
	Wrong data receive/transmit from device from another module
	Device is in wrong state while interacting
	Timeout between modular interaction

Step 3: One of the important parts is finding the injection point. We analyze interacting module and find execution paths, where injected fault will be executed. It is of no use if the fault is not triggered during execution.

Step 4: After injecting the fault, we run the program and observe the output/behavior of the system. We compare the output with the original output and make report according to that.

4.3 Measure Metrics

For measuring the effectiveness and feasibility, a path based fault injection technique is implemented to find mutation score by existing call-based approach and indirect interaction based approach. Every testing criteria have its own advantages and disadvantages. Rather than comparing with existing approaches, it is useful to show that indirect interaction related faults, undetected by call-based technique, could be killed by proposed technique resulting higher mutation score. Fault injection technique considers whether injected fault causes a change in output or not. In call based fault injection techniques, fault generated by module interface or declaration information and does not identify indirect interaction errors. However, our focus is to find indirect interaction and produce possible faults for it to see whether the existing approach can detect those faults. Mutation score is calculated by the ratio between the number of killed mutants and total injected fault as formulated in Equation (1). We use previous work [12] to generate test path for finding fault injection point, which can minimize time and cost of the testing process in a high rate.

$$Mutation\,score = \frac{No\,of\,killed\,Mutant}{Total\,injected\,fault} \times 100\% \qquad (1)$$

5. Case Study and Evaluation

The most common question asked about any testing technique is whether the technique has more coverage than existing one or whether the technique is effective at detecting faults. To demonstrate the effectiveness of the proposed approach, several experiments conducted on various systems to find indirect interaction. Those interactions cannot be found by traditional approaches where the potential cause of errors may lie. It is very difficult to find a single system carrying all indirect interactions together. Therefore, we use a whole program or some parts of the program for analysis. We consider the following research question, which will be addressed after case study.

RQ: Do the test suite identify indirect interaction based faults compare to call based approach? If so, does it increase mutation score?

During the case study, different embedded systems like room heating system, USB control system and general purpose system like student management system, telecom billing system is used. General purpose system is used for the unavailability of open source embedded system. However, we consider that the test result would be somewhat similar. Every system is divided into several modules and has critical interaction among them through interacting variable, which cannot be found in a traditional software system.

For these test programs, we have chosen 44 faults based on different indirect interaction. These faults are injected manually into the source code and executed the programs. A detailed description of the case study is given in Table 2. It represents whether the faults are found or not. For better understanding faults are described in details. We compare previously recorded output (without injecting the fault) with the output generated by injected fault. If the outputs are same, then either the test case is not adequate, or the program is unable to identify the fault. For new indirect interaction, faults should not be detected by existing call-based criteria because they do not consider indirect interaction. From Table 3, it can be seen that for several systems 44 faults are injected and among them, 23 faults are detected which are not detected by the call-based approach.

The goal of this empirical study is to observe if our injected faults are detected by the system or not. If faults are not detected then it means, in software development process, a developer does not concern about particular indirect interaction. We have created 44 mutants and observe how many of them are killed. Call based criteria killed only 21 mutants and 23 mutants are live which are killed by our IDI based approach. As shown in Fig. 7, average mutation score is around 50% by existing call-based criteria, so half of the faults are not detected by the system where our test criteria can cover remaining faults. Thus, we can conclude that there is a huge necessity to take account indirect interaction while performing integration testing and by merging call based approach with IDI based approach, we can get 100% mutation score. *The answer to the RQ is that test suite identifies indirect interaction based faults where call based approach failed to do so and it increases mutation score around 50%.*

Table 2. Detail Description of Faults, How it is Produced, Where It is Placed with (a) Shared Variable, (b) File and (c) Device

Fault description by shared variable / system	**Room heating** (dT)	**Stdinfo1** (numofstudent)	**Stdinfo2** (classMean)
Shared variable exceed the boundary value in one module.	Not found	Not found	Not found
Last value of first module is not equal to first value of second module.	Found	Found	Found
Different type in different module.	Found	Found	Found
Conjugative definition of variable.	Not found	Not found	Not found
Delete one assignment statement.	Not found	Not found	Not found

(a)

Fault description by File / system	**Stdinfo1** (Loadfile-savetofile)	**Tellbill** (Addrecord-listrecord)	**Tellbill** (Search-modify)
File not present in directory.	Found-found	Not found-found	Found-found
Empty file in directory.	Not found-not found	Not found-not found	Found-not found
Directory name is not given.	Found-found	Found-found	Found-found
Wrong operation. Change read file to write file.	Not found-found	Not found-not found	Not found-not found
Incorrect file name in directory.	Found-not found	∞ - found	Found-found

(b)

Fault description by device / system	**Scenario**	**Result(readFromDevice-writeToDevice)**
Device permission changed while interacting	After calling Module A (transmit data to device), change permission of device(only read) before calling Module B(transmit data to device).	Segmentation fault (core dumped) Input lost
Port number changed while interacting	After calling Module A, change port number of device before calling Module B.	No effect
Interacting device not found	After calling Module A, remove device before calling Module B.	Segmentation fault (core dumped) Input lost
Wrong data receive/transmit from device from another module	Module A transmit float number to device and Module B receive integer data from device	Float data transform to integer value

(c)

Table 3. Comparison Between Call Based Approach and Our Approach

System	Total Mutants	No of killed Mutant		Mutation score	
		Call based approach	IDI approach	Call based approach	IDI approach
RoomHeat	5	2	3	40%	60%
StdInfo1	10	4	6	40%	60%
StdInfo2	10	6	4	60%	40%
TelBill	19	11	8	58%	42%
Usb ctrl	4	2	2	50%	50%

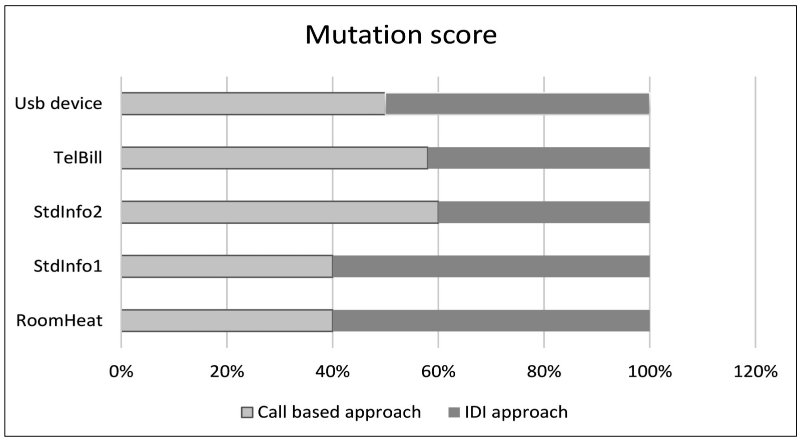


Fig. 7. Mutation Score Between Call Based and IDI Based Approach. For Room Heating System Call Based Approach Cover 40% Faults Where IDI Based Approach Cover 60% Faults.

6. Conclusion and Future Works

The paper presents a general specification of an interaction model including the indirect interaction between modules of the embedded system and proposes a fault injection technique to test fault tolerance of system based on indirect interaction error.

In our research, we identified different indirect interactions that are considered specifying an interaction model and listed different types of faults according to different indirect interaction. Those faults are injected into the source code and the whole or part of the program is executed. The output of the original program is compared to the output generated after fault injection. If the outputs are same, either then the test case is not adequate, or the program is unable to identify the fault. To show the feasibility and effectiveness of the proposed approach, some case studies are done and conducted qualitative experiments on several systems. The result indicates that there is a huge necessity to test indirect interaction while performing integration testing.

Future work will focus on implementing a tool suite of our test technique that automatically generates test data for interacting variables between modules. In addition, we intend to undertake a depth study to find further interaction pattern for feature-oriented software development and perform timing interaction.

References

[1] US Department of Commerce, N., “Updated NIST Software Uses Combination Testing to Catch Bugs Fast and Easy,” 2010.

[2] N. G. Leveson, “Role of Software in Spacecraft Accidents,” *J. Space. Rockets*, Vol.41, No.4, pp.564-575, 2004.

[3] B. Liscouski, and W. Elliot, “U.S.-Canada Power System Outage Task Force,” System 40, 238, 2004.

[4] Honda Admits Software Problem, Recalls 175,000 Hybrids IEETimes [Internet], www.eetimes.com/document.asp?doc_id =1323061, 2014.

[5] M. E. Delamaro, J. C. Maldonado, and A. P. Mathur, “Interface mutation: An approach to integration testing,” *IEEE Transactions on Software Engineering*, Vol.27, No.3, pp.228-247, Mar., 2001.

[6] N. L. Hashim, H. W. Schmidt, and S. Ramakrishnan, “Interface faults injection for component based integration testing,” *International Conference on Computer Informatics*, 2006.

[7] R. Moraes and E. Martins, “An architecture-based strategy for interface fault injection. Workshop on Architecting Dependable Systems,” *IEEE/IFIP Int. Conf. on Dependable Systems and Networks*, Italy, 2004.

[8] Z. Jin and A. Offutt, “Coupling-based criteria for integration testing,” *Software Testing Verification Reliability*, pp.133-154, 1998.

[9] H. Zhu and X. He, “A Methodology of Component Integration Testing,” *Springer*, pp.239-269, 2005.

[10] M. B. Cohen, “Designing test suites for software interaction testing,” Ph.D. thesis, University of Auckland, New Zealand, 2004.

[11] Brady J. Garvin and Myra B. Cohen, “An Overview of Feature- Oriented Software Development,” *Journal of Object Technology*, Vol.8, No.4, Jul., 2008.

[12] H. M. Iqbal and W. J. Lee, “Data Flow Based Integration Testing for Embedded System Using Interaction Model,” *21st Asia-Pacific Software Engineering Conference*, pp. 423-429, Jeju, 2014.

[13] Source Insight [Internet], http://www.sourceinsight.com/ (accessed 12.7.15), 2012.

[14] A. A. Samuel, N. Jayalal, B. Valsa, C. A. Ignatious, and J. P. Zachariah, “Software fault injection testing of the embedded software of a satellite launch vehicle,” *IEEE Potentials*, Vol.32, No.5, pp.38-44, 2013.

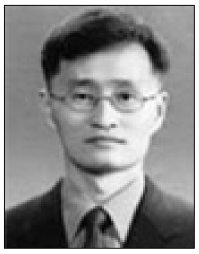
[15] H. Ziade, R. Ayoubi, and R. Velazco, “A survey on fault injection techniques,” *The International Arab Journal of Information Technology*, Vol.1, No.2, pp.171-186, Jul., 2004.

[16] C. Kaner, J. Falk, and H. Q. Nguyen, “Testing Computer Software,” 2nd Edition, Dreamtech Press, 2000.

Muhammad Iqbal Hossain

e-mail : iqbal@knu.ac.kr

Muhammad Iqbal Hossain is a Ph. D. student in the school of computer science and engineering at Kyungpook National University, South Korea. His research interest includes embedded software engineering particularly testing and verification.



Woo Jin Lee

e-mail : woojin@knu.ac.kr

Woo Jin Lee is currently a professor in the school of computer science and engineering at Kyungpook National University, South Korea. He received the Ph. D. degree in Computer Science from Korea Advanced Institute of Science and Technology in 1999. His research interest includes embedded software testing, modeling and verification of embedded software, and component-based software development.