International Journal of Reconfigurable and Embedded Systems (IJRES)
Vol. 8, No. 2, July 2019, pp. 86~98
ISSN: 2089-4864, DOI: 10.11591/ijres.v8.i2.pp86-98

Integration testing based on indirect interaction for embedded system

Muhammad Iqbal Hossain[1], Woo Jin Lee[2]
[1]Department of Computer Science and Engineering, BRAC University, Bangladesh
[2]School of CSE, Kyungpook National University, South Korea

Article Info

Article history:

Received Jan 22, 2019
Revised Mar 29, 2019
Accepted Apr 20, 2019

Keywords:

Embedded system
Fault injection
Indirect interaction
Integration testing

ABSTRACT

Embedded systems comprise several modules that exchange data by interacting among themselves. Exchanging wrong resource data among modules may lead to execution errors or anomalies. Interacting resources produce dependencies between two modules where any change of resources by one module affects the functionality of another module. Several investigations of the embedded system such as aerospace or automobile system show interaction faults between modules are one of the major cause of critical software failures. Therefore, interaction testing is an essential phase to reduce the interaction faults and minimize the risk. The direct and indirect interaction between modules generates interaction faults where indirect interaction is made underneath the interface in which data dependence relationship with resources may cause a different outcome. We investigate errors based on the indirect interaction between modules and introduce a new test criterion for finding errors detectable by existing approaches in unit level but not in integration level. In this paper, we propose a noble approach to generate an interaction model using indirect interaction pattern and design test criteria based on different interaction errors to generate test cases. Finally, we use fault injection and data flow coverage techniques to evaluate the feasibility and effectiveness of our approach.

Copyright © 2019 Institute of Advanced Engineering and Science.
All rights reserved.

Corresponding Author:

Muhammad Iqbal Hossain,
Department of Computer Science and Engineering,
66 Mohakhali, BRAC University, Dhaka, Bangladesh.
Email: Iqbal.hossain@bracu.ac.bd

1. INTRODUCTION

Embedded systems have permeated in every aspect of our everyday life. From complex safety-critical systems like automobile, medical system to home appliances, cellular phones even toothbrushes is controlled by embedded software. So embedded system testing became a serious concern in the product development lifecycle. A study dispatched by the National Institute of Standards and Technology (NIST) found that every year software errors cost the US economy $59.5 billion. It is estimated that around $22.2 billion, could be eradicated by improving test techniques [1]. Unlike the systems of other domains, an embedded system is a combination of sensors, actuators, processors with massive deployment and exhaustive interaction with the environment and resources. Also, the procedure is complex and changes to software interfaces and hardware are common, which makes testing challenging. A number of investigations of aerospace problems show functional interactions among components and inadequate specifications causes serious software failures in aerospace missions. Lutz examined 387 software errors uncovered during integration and system testing of the Voyager and Galileo spacecraft [2]. In 1997, an error was introduced during the evolution of the Minimum Safe Altitude Warning software system (MSAW) where an aircraft crashed at the Guam International Airport [3].

In hazardous sectors, embedded systems need an exhaustive testing process. First of all, each software module is tested distinctly as a unit and then combined to proceed with integration testing. The integration testing has the goal of demonstrating whether developed features work together well enough for the software to submit for system testing. When joining all modules together, errors can emerge from their interactions. There can be direct and indirect interaction between modules and depend on these interaction types, execution paths are generated which ultimately covered by test cases. Therefore, all execution paths are needed to be tested to detect interaction faults. It is not possible to test all paths and till now there is a lack of standard pattern and model for representing indirect interaction and generating test cases. So, a new approach is proposed here for generating an interaction model for representing direct and indirect interaction and test paths are generated for covering indirect interaction of resources.

Embedded system comprises several modules and these separated modules exchange resource data by interacting among themselves. These resource data can flow across software layers between modules within layers. As a result, any changes in resources by one module affects the functionality of another module. Therefore, interaction testing is a vital phase to decrease 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. For example, A module calls B and C modules in its body then test cases must cover all relation between A-B and A-C. But there can be other interactions between module B and C. It is very difficult to test all interactions among them. So the proposed approach is designed to cover only those interaction which is done by resources. This type of interaction is called indirect interaction. There are several cases for the indirect direction that can be done by resources like a shared variable, file, database, device etc. which are described in details in the later part of this paper.

For generating test cases, many existing approaches use black/white box testing technique to find the interaction between two modules by the interface or prototype of the module. This technique can only be applicable for the unit level, not in integration level. Indirect interaction is indistinct for embedded system and still, there is no standard model for addressing this issue. As a result, existing approaches do not consider this interaction while generating test cases. It can produce errors by exchanging wrong resource data and may lead to critical errors or anomalies. It is very difficult to test every interaction among modules of the embedded system so a compact test suite is customized that assurances to resolve a subset of interaction. In this paper, a noble approach is proposed to generate an interaction model using indirect interaction pattern and then design test criteria based on different interaction errors for generating test cases. A brand new aspect of white box testing is proposed which takes account of indirect interaction while generating test cases. Several kinds of indirect interaction are investigated that causes errors through shared resources, file, device, database etc. denoted as interacting variable throughout this paper. For data flow based technique, D-U (Definition-Use)/W-R (Write-Read)/R-T (Receive-Transmit)/I-D (Insert-Delete) as represented as "interaction chain" of the interacting variable, are produced by analyzing the source code and generate test paths according to the sequence of the interacting modules. Interacting variables propagate between modules without parameter or return value and produce an indirect dependency not having any information in the declaration. The key contributions of this paper are:

a. Present a new type of model to represent the indirect interaction between the modules which are called interaction model. It represents how the modules of the system interact among themselves.
b. Specifies the abnormal indirect resource interaction pattern and categorize different fault types.
c. Test cases are generated by symbolic execution to cover the indirect interaction between resources.
d. Case studies show that the proposed approach is very effective for detecting indirect interaction related faults.

2. RELATED WORK

Related work is divided into two segments. At first, related work on path-based integration testing based on the indirect interaction of modules is discussed and then present several fault injection techniques for finding indirect interaction errors. There are few works on integration testing of the embedded system, which consider 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 do not consider internal execution paths of integrated modules for detecting function interaction faults.

A Coupling-based testing technique is proposed here [4] that requires the program execute from definitions of actual parameters through calls to uses of the formal parameters. Coupling based test paths are generated to cover last-def-before-calls, first-use-in-callee, last-def-before-return etc. They described three

kinds of coupling paths as parameter coupling path, shared data coupling path, and external device coupling path. Mostly the test focus is on parameter coupling and uses 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 more complex systems. 21 faults are inserted into Mistix, which does not reflect the integration/interaction relationship of modules. This survey paper in [5] identifies one of the major challenges in integration testing in component-based software engineering is identifying the dependencies. The author investigates how to observe the 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. The contribution of this paper [6] is one of the foundations of integration testing using white box approach. Later many researchers use this concept to develop their own techniques. Here errors are classified into domain error and computational error. A domain error occurs when a specific input follows the wrong path due to an error in the control flow of the program and a computation error exists when a specific input follows the correct path, but an error in some assignment statement causes the wrong function to be computed for one or more of the output variables. Their experience has shown that for most modules it is not possible to detect all the integration errors, even when all paths in the module are examined. Furthermore, they showed that these errors could be detected by examining the normal outputs of the subsystem, without requiring intermediate values or extraneous quantities to be examined. However, for indirect interaction, it is necessary to examine the intermediate value of the variable. Also, the number of paths is quite high and they suggest that a reduction in the number of the path should be examined. A study is done to solve the problem of building test suites for software interaction testing [7]. They have developed a model for the variable strength covering array and have provided some initial bounds and methods for constructing these. It is also shown that this type of model to gain a stronger interaction test suite without increasing the number of test configurations. They use greedy algorithms to make a decision on how to select components for interaction while the goal of testing is to cover as many component interactions as possible. They did not take into account the internal structure of the components and how resources can create interaction between two components. A verification pattern-based approach is developed to generate test scripts quickly for an embedded system [8]. The VP approach classifies system scenarios into patterns. For each scenario pattern (SP), the test engineer can develop a script template to test all the scenarios that belong to the same pattern. But the verification framework is a functional testing framework because it is requirements-driven. So it does not consider the internal behavior of the system. Also, the operational scenarios are generated from the requirements and firmly depends on the engineer's experience.

Fault injection/Mutation-based technique is 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 [9]. They illustrated how incorrect values entering and exiting a unit call and causes erroneous output. Here, only the actual parameter, global variable, and return value are considered. One of its weakness is that it is a mutation operator based technique and imposes a higher cost at every location in the program where the global variable used/defined is a potential location for mutation. This paper introduces an improved, simple and easy technique of interface faults insertion using AspectJ for Java component-based applications [10]. 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 is to propose a fault injection strategy to test the interaction among components [11]. For that reason, interface faults are introduced by corrupting input data as well as interface output data. However, almost every case researcher focuses on interface information and generate faults according to the input and output of the module. However, 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.

3. INDIRECT INTERACTION

Embedded systems encompass a broad range of hardware and software systems where the software system is divided into several modules, which are developed by several vendors or different developer teams. An interaction takes place when two or more modules have a calling relationship among them and accessing the same resources from several modules. Although some researchers use the same term to classify feature interaction, human-computer interaction, interaction testing etc. which is quite distinct from our work. For example, the interaction testing focused on how components interact with 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. 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 [12].

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 the 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 resource. The main difference between integration testing and interaction testing is that in integration testing, data transactions are visible such as parameter (variable, file, memory) return value etc. but in interaction testing, data transactions are not visible from the abstract view of the system.

3.1. Abnormal scenarios by indirect interaction

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

IDI 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 variables to communicate among them and related modules. Shared variables make data available from one module to another or among multiple processes, but have no call relation. It is very difficult to identify this interaction because shared data information is not present in the 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 needs to be compared to avoid value or type mismatch. It is done to make sure that there is no intermediate modification of the value. We use data flow based testing techniques to find all definition and use information of a shared variable and generated test paths. Any faults in data flow will be resolved by it. For example, Figure 1 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.

IDI 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, files are 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 are used inside modules. However, it is very important to test how the files are being used or whether the files are performing according to specification. While interacting, it is needed to test whether two modules follow that same file structure or not. For example, a file may contain an integer value instead of a floating number.

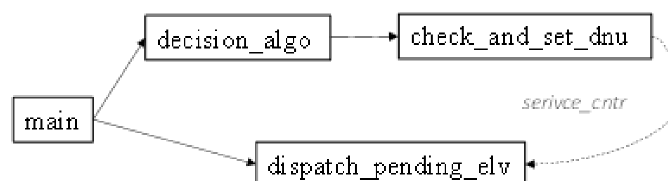

Figure 1. Indirect interaction by Shared variable

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 a file is not present in a directory. For this reason,

these abnormal cases during interaction should be tested. For example, Figure 2 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.

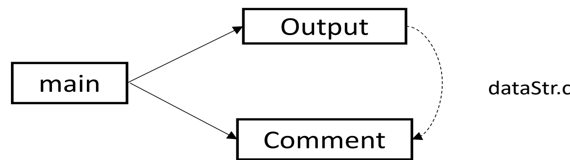

Figure 2. Indirect interaction by file

IDI by I/O device: Embedded systems contain extensive applications 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 parameters. Therefore, the tester does not test how devices are handled inside modules. However, it is very important to test how the devices are being used or whether the devices are performing according to specification. A device may have wrong state, timing failure, fault handling etc., which may lead to critical errors during interaction.

For example, Figure 3 represents indirect interaction using the *level sensor* in a water level monitoring system. Here, level sensor continuously reads the water level to start/stop the motor and in particular level, it triggers an alarm.

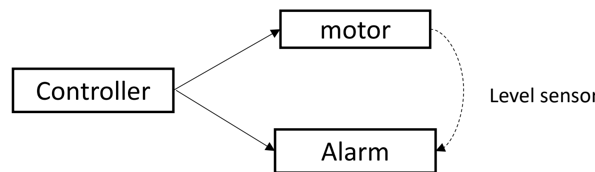

Figure 3. Indirect interaction by device

IDI by database: Embedded systems often need to use the database for storing configuration data, init data, trace data, error log data etc. Whenever a certain action is performed in on the module of an application, a corresponding CRUD (Create, Retrieve, Update and delete) action gets invoked. Another module may perform another action. So we need to test the data integrity. This means that following any of the CRUD operations, the updated and most recent values/Status should appear in another module. When a certain event takes places on a certain table in a module, a trigger can be auto instructed to be executed on another table. Some other event may take place at the later table in another module. As a result, an event in one module can indirectly affect another module. For example, in Figure 4 a module inserts purchase orders, and the product is removed or updated by another module, future events will have to fail.

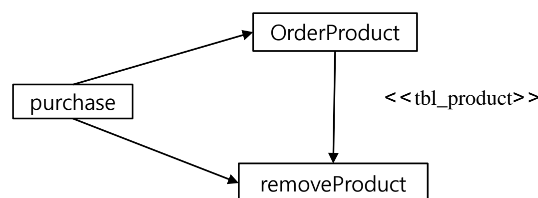

Figure 4. Indirect interaction by database

3.2. Formal model for indirect interaction

Modular interaction is done by the clearly well-defined interface through parameter or return value and most of the existing works focused on faulty message/data passing through modules. The functional interface contains the required information to interact with another module. Most of the time interfaces are poorly documented and only contain information related to direct interaction, not an indirect one. Finding indirect interaction is complicated for deficiency of standard pattern and model. An indirect interaction is pictured 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. An indirect interaction is represented by the interaction model generated by extending call graph in Figure 5.

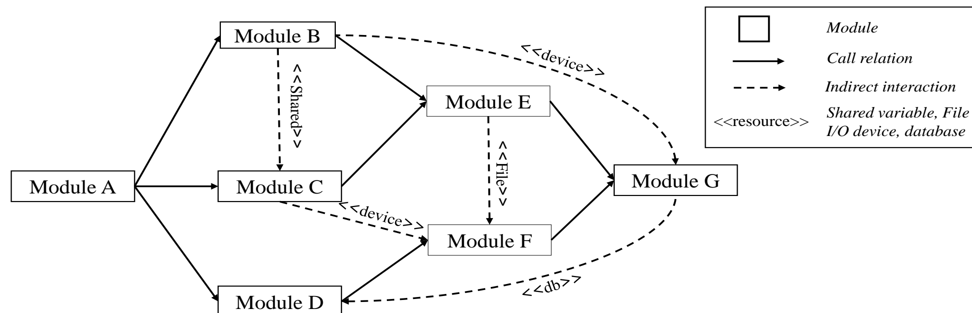

Figure 5. Sample interaction model

An indirect interaction is 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 call graph is generated automatically using the static analyzing tool and then find the indirect interaction between modules. Figure 5 represents module C and module B have an indirect interaction by the shared variable, module B, and module A has a call relation, module F and module E have indirect interaction through a file etc. The directed edges represent the calling sequences of the modules. Indirect interaction can be formally defined as follows,

Definition 2: Interaction model can be represented as G=(V, E); where V comprises finite set of modules and E contains a set of interactions, $E \subseteq V \times V$. Solid edges represent call relation and dashed edges represent indirect interaction where indirect interaction is the set of {Shared variable, File, Device, and database} and directed edges represent the calling sequences of V.

The proposed IDI approach comprises two phases. First phase interacting variables are found between two modules by generating an interaction model and define some new criteria where error may lie. In the second phase, test cases are generated efficiently for solving or preventing those errors. A tester should take account those new test criteria while generating test cases.

4. PROPOSED INDIRECT INTERACTION (IDI) BASED APPROACH

The proposed IDI approach comprises two phases. In the first phase, we find interacting variables between two modules by generating an interaction model and define some new criteria where the error may lie. In the second phase, test cases are generated efficiently for solving or preventing those errors. A tester should take account those new test criteria while generating test cases.

4.1. Interaction model generation

To generate an interaction model, as shown in Figure 6, Understand tools is used to parse the source code and then maintained in a database to store information dynamically for generating a call graph. Understand is used to analyzing the source code which understands and maintain large amounts of newly created source code. The IDE provides multi-language, maintenance-oriented, cross-platform features [13]. It has architectural features that support to produce hierarchical accumulations for units of source code. These units can be named and handle in various ways for further analysis such as control flow graph generation, call graph generation, locating declaration files, finding cluster calls etc.

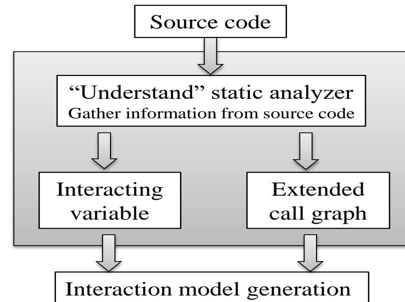

Figure 6. Overview of generating Interaction model

A list of the caller-callee relationship between two modules is acquired by generating a call graph as modules and arcs. Extraction of the interacting variable is a semi-automatic process, which can be done by the 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. Here the focus is on the resources accessed by two modules inside their body, which are not present in interface information. As discussed in section 3 that there are several kinds of indirect interaction that causes fault. It needs to find the following relation between the two modules:

- Same global variable defined or used.
- Same file open for read/write operation.
- Same device connected for receiving/transmitting signal.
- Same database access and perform query.

As a result, shared variable, file, device or database are found which are denoted as an interacting variable and their corresponding modules. The interaction model is generated by combining all the information. A flowchart of finding an interacting shared resource for generating interaction model is shown in Figure 7.

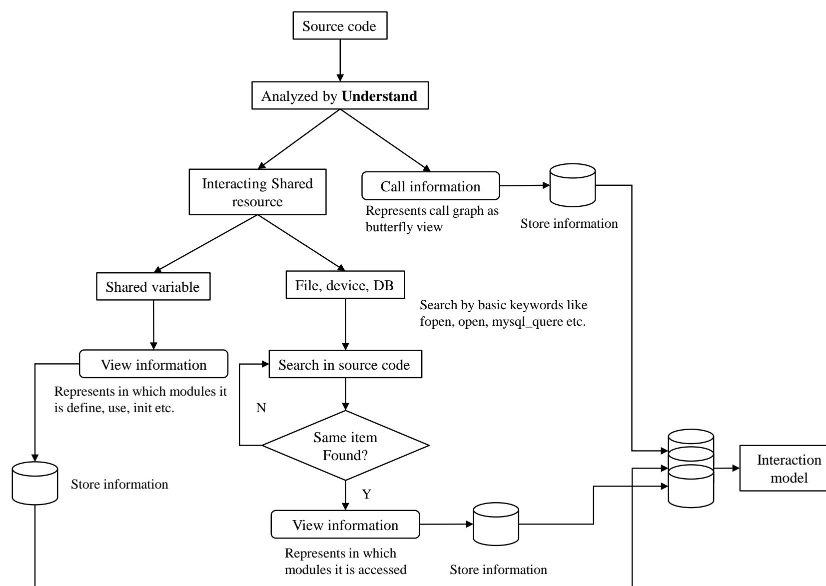

Figure 7. Flowchart to find Interacting shared resources

4.2. Generation of test case

In the second phase, based on the designed test template along with test adequacy criteria, test paths are generated for each indirect interaction. The white box testing approach provides a variety of test adequacy criteria such as a statement, branch or definition-use coverage. Our test adequacy criterion for three types of indirect interactions which are is already defined in section 3.

A test template is a basic overview of how test case generation procedure will proceed. A basic test template for interaction test path generation is given below where we use an elevator system as running example depicted in Figure 8. In elevator system, decision_algo decides which lifts to service which all requests, check_and_set_dnu checks which all lifts are servicing full requests and update their DNU status and add_service_request sets service level as requested.

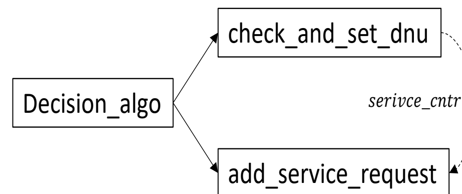

Figure 8. Running example of teaching the assistant system

Here, *decision_algo* have direct call relation with *check_and_set_dnu* and *add_service_request*. On the other hand, *check_and_set_dnu* and *add_service_request* module have shared variable *service_cntr*.

Step 1: Interaction model represents entire call information between modules of the system. From this model, two modules (leaf node) are identified which have indirect interaction and what type of interaction is present there. In the example, service_cntr is an interacting defined and used in check_and_set_dnu and add_service_request modules.

Step 2: After finding the modules which are interacting with them, it is needed to travel back to their parent node until there is a common ancestor. Here, the same interaction model is used where modules represented as nodes and interactions are represented as edges. At the end of this step, all traversal information is collected and create sub-tree where leaf nodes are interacting modules and there present a common root for them. Here, both *check_and_set_dnu* and *add_service_request* have a common root node *decision_algo*.

Step 3: In this step, the proper sequence of calling modules are generated from an ancestor node to a leaf node as represented in the sub-tree with control flow information. Usually, control flow uses to find the order in which module calls for an imperative program are executed. From program source code, control flow information is collected and find all sequences until each node visits from an ancestor. The sequence for the example is: $decision_algo \rightarrow check_and_set_dnu \rightarrow add_service_request$

Step 4: In the final step, first, interaction paths are generated using function call sequence. A test path is a sequence from the starting node to a terminal node of the control flow graph of a program and contains several paths for covering each module sequence. Secondly, from interaction type, which is found in step 1, test criteria is implemented and produce interaction chain. The chain represented as a series of nodes where the interacting variable is defined/used, read/write, transmit/receive or insert/delete. Only those paths are selected that are feasible by the chain and set injection point here. Proposed test path generation tool does test path selection procedure automatically. After that, test cases are generated by symbolic execution technique for executing those test paths. In the example, *service_cntr* is the interacting variable and its DU chain is as follows.

$DU\ chain$: 87 184

All the paths are generated according to module sequence and only those paths are feasible which are covered by DU chain. Some partial feasible paths (sequence of node number) are given below:

1. 203 204 205 84 85 86 87 95 96 97 98 206 207 208 209 210 211 212 213 214 176 177 178 179 180 181 182 183 184 185 187 188 190 191 192 193 215 216 217 218 223 224 225 230 231 232

2. 203 204 205 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 206 207 208 209 210 211 212 213 214 176 177 178 179 180 181 182 183 184 185 186 187 188 190 191 192 193 215 216 217 218 223 224 225 230 231 232

3. 203 204 205 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 206 207 208 209 210 211 212 213 214 176 177 178 179 180 181 182 183 184 185 187 188 190 191 192 193 215 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232

For each of these test paths, test cases are generated by symbolic execution technique. For example, we solve the first test path with symbolic execution and get the following path condition as shown in Table 1.

Table 1. Evaluating predicate condition for generating test cases

Serial	I	MAX_LIFTS	SERVICE_CNTR	DIR_UP	I2	MAX_REQUESTS	DIR_IDLE	DIR_DOWN	Result
1	7	2	92	2	7	15	0	2	fail
2	1	2	1	1	3	15	2	0	pass
3	2	2	84	1	3	15	1	0	fail
4	0	2	61	1	3	15	2	2	fail

(I<MAX_LIFTS) && (SERVICE_CNTR>=3) && (CUR_PROC_INP == DIR_UP) && (I2<MAX_REQUESTs) && (ELV_SERVICE_DIR !=DIR_IDLE) && (ELV_SERVICE_DIR== DIR_DOWN)

An algorithm is designed which randomly select input condition and execute with the path condition. Path condition contains the interacting variable along with other internal variables. If the path condition is satisfied then it is treated as a test input. As represented in Table 1, number 2 input condition fulfill the path condition, so it is a test case for that particular test path. Similarly, for all test paths, we generate the path condition and by evaluating it we get the test case. However, generation of test cases using symbolic execution is not covered here. Symbolic execution technique is well-understood, straightforward technique and many works already have published in many research journal [14, 15].

4.3. Fault injection technique

Fault injection technique is described as a deliberate injection of a fault into a running system during a test activity, to determine whether the system reacts well to off-nominal or exceptional conditions [16, 17]. 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 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, it needs to modify the code, add new code or delete part of the code. Figure 9 show the fault injection process is divided in,

1. Pre-injection analysis
2. Inject actual fault

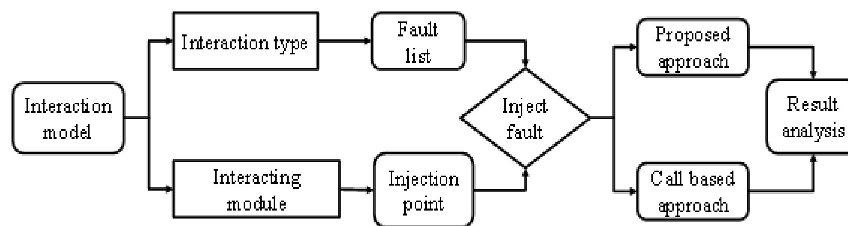

Figure 9. Overview of fault injection technique

The pre-injection analysis involves creating the fault according to test criteria. Test criteria are 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 can take place. After that, we inject the fault into the system and execute it. A tester observes the behavior of the system and compares with previous output. Faults have so many varieties that we cannot study every kind of their impact on software [18]. We select most relevant faults which may produce by indirect interaction and the list of faults is given in Table 2.

Fault injection technique is used to evaluate the proposed approach by finding the fault detection rate. The overview of the technique is given in Figure 9 and the steps are given below:

- Step 1: Like data flow based criteria, we generate interaction type and interacting modules from the interaction model.
- Step 2: According to the type of interaction, we select possible faults from the 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 the 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.
- Step 3: One of the important parts is finding the injection point. We analyze the interacting module and find execution paths in where injected fault will be executed. It is of no use if the fault is not triggered during execution.

- Step 4: Then we inject the fault into the system and run the program and observe the output/behavior of the system for activation of the fault. This fault activation process is done by our proposed approach and random fault activation technique and compares the fault activation rate for evaluation.

Table 2. Different faults by the interaction of resources

Type	Criteria
Shared Variable	Shared Variable exceed boundary value in one module
	Last value of first module is not equal to first value of second module
	Use definition use criteria for testing (DU testing)
File	File Removed in between two modules
	File data mi smatch between modules
	Required value is not present in file
	Garbage value handling
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
Database	Read data from empty table where data deleted by other module
	Write data to table which is altered by other module
	Top most data required but deleted by another module

5. CASE STUDY AND EVALUATION

Several case studies are performed on how to generate test cases for shared resource and timing constraints for indirect interaction. For shared resource based indirect interaction, number of test cases for covering all DU and indirectly interacting DU is compared. Also, for evaluating the fault detection rate, between the proposed approach and call based approach, fault injection technique can be used.

5.1. Comparison of number of test cases for DU coverage

In the first evaluation criteria, the required number of test cases are computed for covering all DU and interaction variable DU. All DU coverage means all definition-clear path for all the variables in that interaction. The number of paths is too high for the mid-level program and for the large system there will occur state explosion problem. It is not efficient to compute a large number of test cases which increase testing cost and time at a high rate. So focusing on indirectly interacting variable DU only to reduce the overhead for generating test case.

Table 3 represents the comparison between number of test cases required for all DU coverage and all interacting variable DU coverage. As shown in the first case, the number of test cases required for covering all DU is 3192 where the number of test case required for covering interacting variable DU coverage is 324 which is comparatively lower and realistic than all DU coverage. Also, it covers 10.15% of all DU coverage

Table 3. Comparison between numbers of test cases required for all DU and all interacting variable DU coverage

System	Number of test cases		Coverage
	All DU	Interacting DU	
decision_algo → check_and_set_dnu (lift1, lift2); decision_algo → add_service_request (elevators.id, current_proc_input); serivce_cntr	3192	324	10.15%
main → dispatch_pending_elv; main → decision_algo (current_proc_input); decision_algo → add_service_request (elevators.id, current_proc_input); serivce_cntr	2880	576	20.00%
main → dispatch_pending_elv; main → decision_algo (current_proc_input); decision_algo → check_and_set_dnu (lift1, lift2); serivce_cntr	4376	1240	28.34%
find_css_set → find_css_set (cgroup_root_count); find_css_set → link_css_set (cset, c); tmp_links	126	32	25.4%

5.2. Evaluating through fault detection rate

To evaluate the proposed approach, the fault detection rate is computed for several systems. At first, the list of faults is specified which occur in the direct and indirect interacting. For direct interaction related fault, most general kind of faults are listed according to IEEE standard Classification for software anomalies - IEEE std 1044-2009 and IEEE Standard for Software and System Test Documentation-IEEE Std 829-2008 [19, 20]. Indirect interaction faults are designed by analyzing the behavior of the resources. All the inserted faults are shown in Table 4.

For each resource, both direct and indirect interaction faults are inserted and analyze how faults are detected by the call based approach and proposed approach. In call-based approach, test cases are generated covering all parameter and return value as stated in the interface. The interface does not contain any indirect interaction information so it is expected that faults generated by indirect interaction wouldn't detect here. On the other hand, the proposed approach has both interface and indirect interaction information so it should detect all possible indirect interaction errors. A comparison between numbers of direct and indirect faults detected by the call based approach and proposed approach is shown in Figure 10. Considering only student information system, total 44 faults (direct interaction faults 35, indirect interaction faults 9) are injected. Call-based approach only detects 33 direct interaction faults and none of the indirect interaction fault. The proposed approach detects 27 faults which contain 18 direct interaction fault and 9 indirect interaction faults.

Table 4. List of inserted faults for direct and indirect interaction

Resaurces	Fault type	Fault description
Shared variable	Direct interaction	A parameter in a function call was missing
		In complete expression was used as parameter
		Wrong information was passed to a function call (Value, expression result. Etc)
	Indirect interaction	Shared variable exceed boundary value in one module
		Last value of first module is not equal to first value of second module
File	Direct interaction	No input file in present in directory
		Wrong file name
		Invalid parameter while opening the file
	Indirect Interaction	File removed in between two modules
		Required valueis not present in file
		Garbage value handling
Database	Direct interaction	Modify SQL statement
		Modify database connection information
		Lost database connection
		Modify column information in query
	Indirect Interaction	Read data from empty table where data deleted by other module
		Write data to table which is altered by other module
		Top most data required but deleted by another module

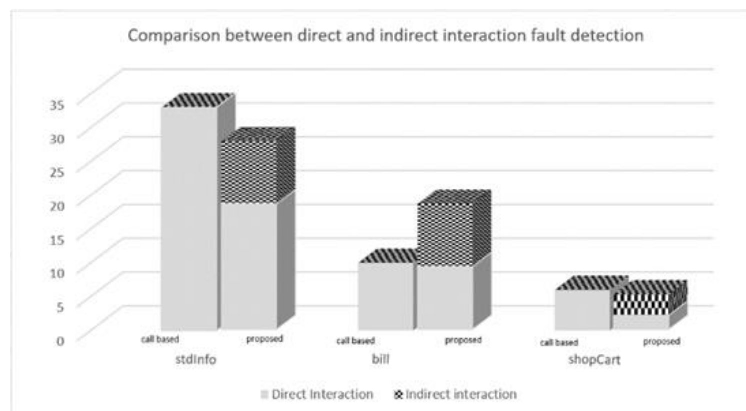

Figure 10. Comparison of the number of direct and indirect interaction fault detection

As shown in Table 5, total 80 faults are injected in the source code in several systems where 59 of them are direct interaction faults and 21 indirect interaction faults. As expected, call based approach did not detectany faults generated by indirect interaction. It only detects 49 faults which are direct interaction faults. The proposed approach detects 21 indirect interaction faults along with 29 direct interaction faults.

The proposed approach detects 100% indirect interaction fault in every case and in addition it also detects direct interaction fault. For example, for the *stdInfo* system, call based approach detect 94.29% direct interaction fault and none of indirect interaction fault. The proposed approach detects 100% indirect interaction faults along with 51.43% direct interaction fault. The result clearly shows how efficiently the proposed approach detects indirect interaction faults.

Table 5. Fault detection by call based and proposed approach

System	Inserted faults			Faults detected by call based approach			Faults detected by call based approach		
	DI faults	IDI faults	Total	DI faults	IDI faults	Total	DI faults	IDI faults	Total
StdInfo	35	9	44	33 (94.29%)	0	33 (75%)	18 (51.43%)	9 (100%)	27 (61.36%)
TellBill	16	9	25	10 (62.29%)	0	10 (40%)	9 (56.25%)	9 (100%)	18 (72%)
shopCart	8	3	11	6 (72.00%)	0	6 (54.55%)	2 (25%)	3 (100%)	5 (45.45%)
Total	59	21	80	49 (83.05%)	0	49 (61.25%)	29 (49.15%)	21 (100%)	50 (62.50%)

6. CONCLUSION AND FUTURE WORK

The paper presents a general specification of an interaction model including the indirect interaction between modules of the embedded system and proposes test adequacy criteria which can be included with data flow driven integration testing approach to generate test cases efficiently. Also, a fault injection technique is used to test the fault tolerance system based on indirect interaction error.

In our research, we identified different indirect interactions that are considered specifying an interaction model and then designed test criteria for each type of interaction. Using data flow analysis by the specialized tool, the source code is parsed to collect required information and this information is used for generating an interaction model. Using several techniques interaction model is break down into sub-trees and find a common ancestor for each of the interacting modules. Using the sub-tree and control flow information of the source code, module sequence from ancestor to leaf node is generated. From the module sequence, the number of interaction paths is generated and compared with the tests created based on the test criteria. Those paths, that are feasible by designated test, are selected according to the DWRI-URTD chain of the interacting variable. After that, symbolic execution technique is used to generate test cases for each of the test path. On the other hand, some faults are listed according to different indirect interaction and those faults are injected into the source code and execute the program. The output of the original program is compared to the output generated after fault injection. If the outputs are the 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.

In future work, we are planning to implement our test technique as a tool suite to generate test data for interacting variables between modules automatically. In addition, for generating more efficient interaction model, we intend to undertake an in-depth study to find further interaction pattern which can be implemented in the larger embedded system.

REFERENCES

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

[2] National Transportation Safety Board, “Controlled Flight into Terrain, Korean Air Flight 801, Boeing 747-300, HL7468", Nimitz Hill, Guam, August 6, pp.212, 2000.

[3] National Transportation Safety Board, “Controlled Flight into Terrain, Korean Air Flight 801, Boeing 747-300, HL7468, Nimitz Hill, Guam, August 6, 1997,” p. 212, 2000.

[4] Z. Jin and A. Offutt, “Coupling-based criteria for integration testing,” *Softw. Test. Verify. Reliab*., vol. 154, no. July, pp. 133–154, 1998.

[5] H. Zhu and X. He, “A Methodology for Component Integration Testing,” Springer, pp. 239–269, 2005.

[6] A. Haley and S. Zweben, “Development and Application of a White Box Approach to Integration Testing,” *J. Syst. Softw.*, vol. 15, pp. 309–315, 1984.

[7] M. B. Cohen, “Designing test suits for software interaction testing,” The University of Auckland, 2004.

[8] W. Tsai and L. Yu, “Rapid Embedded System Testing Using Verification Patterns,” *IEEE Software*, vol. 22, no. 4, pp. 68–75, 2005.

[9] Â. E. Delamaro, J. C. Maldonado, and Aditya p. Mathur, “Interface Mutation: An Approach for Integration Testing,” *IEEE Trans. Softw. Eng*., vol. 27, no. 3, pp. 228–247, 2001.

[10] N. L. Hashim, H. W. Schmidt, and S. Ramakrishnan, "Interface faults injection for component-based integration testing," Comput. Informatics, 2006. ICOCI '06. *Int. Conf.*, pp. 1–6, 2006.
[11] R. Lúcia and D. O. Moraes, "Architecture-based Strategy for Interface Fault Injection," Work. Archit. Dependable *Syst. Int. Conf. Dependable Syst. Networks*, 2004.
[12] S. Apel and K. Christian, "An Overview of Feature-Oriented Software Development," *J. object Technol.*, vol. 8, no. 4, pp. 1–36, 2009.
[13] "Scitools-Understand (visualize your code)." [Online]. Available: www.scitools.com.
[14] J. C. King, "Symbolic Execution and Program Testing," Communication, vol. 19, pp. 385–394, 1976.
[15] J. Zhang, C. Xu, and X. Wang, "Path-Oriented Test Data Generation Using Symbolic Execution and Constraint Solving Techniques," *Int. Conf. Software Eng. Form.* Methods, no. 60125207, 2004.
[16] A. A. Samuel, N. Jayalal, B. Valsa, C. A. Ignatius, and J. P. Zachariah, "Software fault injection testing of the embedded software of a satellite launch vehicle," *IEEE POTENTIALS*, vol. 32, no. September, pp. 38–44, 2013.
[17] H. Ziade, R. Ayoubi, and R. Velazco, "A Survey on Fault Injection Techniques," *Int. Arab J. Inf. Technol.*, vol. 1, no. 2, pp. 171–186, 2004.
[18] J. F. and H. Q. N. C. Kaner, "Common software errors," in Testing Computer Software Second Edition, Dreamtech Press, pp. 1–89, 2000.
[19] IEEE Computer Society, IEEE Standard Classification for software anomalies (IEEE std 1044-2009). 2010.
[20] IEEE Computer Society, IEEE Std 829-2008, *IEEE Standard for Software and System Test Documentation*, vol. 2008, no. July. 2008.

BIOGRAPHIES OF AUTHORS

Muhammad Iqbal Hossain is currently an Assistant professor in the school of computer science and engineering at BRAC University, Bangladesh. His research interest includes embedded software engineering particularly testing and verification.

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.