Data flow based integration testing for embedded system using interaction model

Hossain Muhammad Iqbal  
School of CSE  
Kyungpook National University  
Daegu, South Korea.  
iqbal@knu.ac.kr

Woo Jin Lee  
School of CSE  
Kyungpook National University  
Daegu, South Korea  
woojin@knu.ac.kr

Abstract— Embedded system covers a large range of highly sensitive area where a slightest error can damage large number of resources even cost human life. So testing embedded system is extremely essential concern in computer science industry. Errors can lies between interacting software and hardware module of embedded system. This interactions can be modeled in terms of data flows between subsystems, i.e. module, component etc. Tracking dataflow across different components is important because applications use this to activate interactions that acquire different values and resources in ways that may ultimately exercise faults. Integration testing is a verification technique of the interfaces among system parts. The integration test of software components is especially concerned with the detection and uncover of interface errors, which are not detectable during unit testing. Existing integration testing approaches typically focused on direct interaction of subsystems which is done by function call but there exists other interactions for indirect dependencies between subsystems, which may occur faults in the system. There are no specific general models for representing all interactions. In this paper we propose an interaction model to cover all possible direct and indirect interaction and provide an integration testing approach using data flow analysis to generate minimal test cases efficiently.

Keywords— Integration testing, Interaction model, Dataflow analysis, Definition-Use chain.

I. INTRODUCTION

Embedded system covers a large range of computer systems from small computer-based devices to large systems monitoring and controlling complex processes. Embedded systems such as nuclear plant, automotive system, medical system and so on. These are highly sensitive area and any fault can damage large number of resources even cost human life. So testing embedded system is extremely essential to reduce the faults as well as minimize risk. A study commissioned by the National Institute of Standards and Technology found that software errors cost the US economy $59.5 billion every year. The study estimated more than a third of that amount, $22.2 billion, could be eliminated by improved testing [1].

Embedded system includes several sub systems which consist of two parts: hardware and software components. A software component is a software package or a module that encapsulates a set of related functions and hardware component is the electrical device itself. The software components have direct or indirect interaction between them. The interactions in embedded systems can be modeled in terms of data flows between components. Interaction testing approach test all the possible interaction to resolve the faults. To do so tracking data flows across different subsystems is important because applications use this to activate interactions that acquire different values and resources in ways that may ultimately exercise faults. Data flow can be considered to be a form of structural testing which focuses on the variables used within a program. Variables are defined and used at different points within the program; data flow testing allows the tester to follow the changing values of variables within the program. Integration testing is used as a verification technique of the interfaces among system parts. Integration test of software components is especially concerned with the detection of interface error and uncovers errors which are not detectable during unit testing.

Most of the existing testing technique focuses on basic interaction properties separately like calling function with parameter, execution order of input parameters, interaction statement. These are based on direct coupling relationships that exist among different variables across different call sites in functions. But there exist many indirect interactions between the subsystems where fault can occur. But existing testing techniques only concern particular interaction properties and overlook indirect or other interaction between subsystems while generating test execution path. To solve this problem, we propose an interaction model which include all direct and indirect interactions and use data flow based integration testing between subsystems to generate minimal test cases efficiently.

The rest of the paper is organized as follows. In section 2, we include related work and briefly discuss about interaction model in section 3. Section 4 includes implementation technique and section 5 includes case study and evaluation. Finally in section 6 we provide conclusion and future works.

II. RELATED WORK

There are a few works on interaction testing which are basically focused on execution ordering of input parameters, interaction statement in threading, wrong parameter ending etc. A novel approach for automated test data generation for coupling based integration testing of object-oriented programs using genetic algorithm is presented in [2]. This approach takes
the coupling path as input, containing different sub paths, and
generates the test data using genetic algorithm. This approach
can be implemented when one procedure passes parameters to
other procedure, which means when parameter couplings
occur. It is a kind of direct interaction between two subsystems
and it assumes that there are no dependencies between other
subsystems. The main disadvantage of this approach is other
than the two interacting subsystem there may be some
dependencies with other subsystem. While performing
coupling based interaction then this fault cannot be caught.

A path-based integration testing method is discussed in [3].
Here integration test cases are generated based on two outputs
from interface net and mapping table. To analyze the interface
net and mapping precisely, they proposed a forward slicing
technique to obtain the dependency relation for input and
outputs. Strategies for class test order for integration testing
have been discussed in [4], as integration test order specifies
the estimated cost for developing stubs and drivers. Semantics
for UML diagram is enable the dependency analysis of
components for integration testing. An integration testing
based on selection of number of parameter and number of
values of each parameter is being discussed in [5] [6]. This
ultimately results to generate large number of test cases to
cover large number of interaction. Suppose if there are n
parameter and each parameter have d value then total
combination is nd, which is impossible for large system to
exhaustively test all of these combinations of parameter values.
To reduce the large number of combination many existing
works adopt t-way strategies to select number of parameter
using bee’ s algorithm, ant colony algorithm etc. But still t-way
testing can generate a large number of tests, which makes it
impractical to execute the tests and evaluate their results.
A formal specification-based integration testing approach is
proposed in [7]. In this paper, Data flow paths are derived from
rigorous CDFDs, and formal functional scenario sequences are
extracted from the paths as the foundation for test case
generation. Usually formal specification approach used in unit
testing and it is difficult to specify inter modular relation of
the system. On the other hand, for constructing functional scenario
sequence only directly interacting input and output variables of
function are considered. But data those flows between modules
by indirect interaction is not consider in this approach. So fault
generated by indirect interaction cannot be found.

The existing approaches do not have a general model for
representing interactions between subsystems. We realize that
there is a necessity of specifying an interaction model that will
contain all possible direct and indirect interaction between
subsystems. On the other hand, number of test cases is an
important issue from cost and time perspective. In existing
works minimizing number of test cases are ignored so number
of test cases is quite high. We use data flow based approach
to generate minimal test cases. Usually data flow based technique
is used for testing a unit, procedure or function etc. but we will
use it in interaction testing.

III. INTERACTION MODEL DEFINITION

Integration testing technique focuses on basic interaction
properties like calling function with parameter, execution order
of input parameters, interaction statement etc. These are based
on coupling relationships that exist among different variables
across different call sites in functions. But there exist many
indirect interactions where faults can occur by inappropriate
data flow between the subsystems. To solve this problem we
find all interaction pattern between subsystems to generate an
interaction model and implement a data flow based testing
technique to generate minimal test cases efficiently. There may
exist several kinds of direct and indirect interaction between
subsystems. According to interaction pattern we find definition
use of interacting variables between subsystems and generate
DU chain. Test path will be generated by the DU of interacting
variables. We are concentrating on building a interaction model
that represent all possible interaction between sub systems.

A. Interaction patterns between subsystems

Interaction pattern represents how data flows across
different Subsystems. Based on these coverage criteria we find
the following interaction patterns between subsystems. Some
of these patterns are similar with existing approaches because
we are looking forward to produce a general model of
interaction.

1) Direct Interaction: Direct interaction between subsystems
can be done by variables, which flows directly from
subsystem to subsystem. A variable can flow through
subsystem in four ways; parameters, call by value, call by
references and return value.

a) Parameter: Some variable can directly interact from
one subsystem to another by parameter.

b) Call by value (in): The call by value method of
passing arguments to a function of another subsystem
copies the actual value of an argument into the
formal parameter of the function.

c) Call by reference (in-out): The call by reference
method of passing arguments to a function of another
subsystem copies the address of an argument into the
formal parameter.

d) Return variables (out): A module of a subsystem can
return some value to another subsystem.

2) Indirect interaction: Interacting variables of one
subsystem can be defined by other local variable of a
subsystem and used by another local variable of another
subsystem. So there produce a dependency among
the variable and indirect interaction between the subsystems.
Suppose V1 is defined by variable d1 of subsystem S1 and
used by another variable u1 of subsystem S2, in that case
S1 and S2 have indirect dependency by (d1, u1).

```
1. int p1, p2, p3, p4;
2. p1 = randID();
3. p2 = randID();
4. p3 = sub1.addByTen(p1, p2);
    ...
    ...
8. p4 = sub2.getSquare(p3);
    ...
    main
```

```
int sub1
    int addByTen(int f1, int f2){
        f1 = f1+10;
        f2 = f2+10;
        int res = f1*f2;
        return res;
    }
```

```
int sub2
    int getSquare(int a){
        int res = a * a;
        return res;
    }
```

Figure 1. Sample source code with three subsystems (main, sub1 and sub2)
Figure 1 represents sample code containing three subsystems {main, sub1, sub2}. In this code main subsystem call a function, addByTen() in line 4, of sub1 by parameter p1, p2. addByTen() function of sub1 return a value after some computation and main subsystem put that value to p3. Again main subsystem called another function, getSquare() in line 8, of sub2 with parameter p3 which is actually returned by sub1. Sub2 get an actual parameter p3 to formal parameter a. So in this case res is defined in sub1 and a is used in sub2 creates an indirect dependency.

3) Shared resources: In some system it is needed to share same resources between many subsystems for its accessibility and availability. Usually multi-threading or concurrent programming need to share same data between processes. File or devices are some kind of shared resources where these can be used by several subsystems.

A file can be read/write in several subsystems creates an indirect dependencies. Memory or register allocates some space in the system. The address of this allocated space can be used in several subsystems. So two subsystems can interact between themselves through register variables.

B. Interaction Model representation

Interaction model represents all kind of interaction between subsystems using interaction patterns which are described earlier. Generally it shows interaction of direct calling function, indirect dependencies and shared resources between subsystems.

Definition 1. Interaction model, IM, is a nonempty set which includes 4-tuples (S, V, DD, ID) where,

a) S is a set of subsystems. If there are n subsystems then S can be represented as {S_1, S_2…S_n}.

b) V is a set for interacting variables \{V_1 \cup V_2 \cup …V_n\}; where \text{V_i}\ the variables of subsystem is S_i.

c) DD \subseteq S \times S, which is the direct dependency between two subsystems. If subsystem S_1 calling some method of subsystem S_2, can be represented by solid arrow between them, S_1\rightarrow S_2.

d) ID \subseteq S \times S, which is the indirect data dependency between two subsystems. If variable id_1 is defined in subsystem S_1 and used in subsystem S_2 can be represented by dashed arrow between them, S_1\rightarrow S_2. ID has a set of interacting definition-use pairs between subsystems which can be represented by DU(id_1) = \{(v_1, v_2) | \text{there is definition-use relationship between } v_1 \in V_1 \text{ and } v_2 \in V_2\}.

Figure 2 shows an interaction model of Figure 1 which contains three subsystems {main, sub1, sub2}. main have a direct dependency with sub1 and sub2. On the other hand, sub1 and sub2 have indirect dependency between them by [res, a] which depicts that the interacting variable is defined by variable res of sub1 and used by variable a of sub2.

C. Dataflow abstraction of module

Abstraction is a cognitive means according to which, in order to overcome complexity at a specific stage of a problem solving situation, we concentrate on the essential features of our subject of thought, and ignore irrelevant details. Abstraction is especially essential in solving complex problems as it enables the problem solver to think in terms of conceptual ideas rather than in terms of their details. In our approach we create abstraction of all the subsystems on the basis of the definition-use criteria. Because all the subsystems are already unit tested so we do not consider the internal dataflow or intra-procedural analysis. We abstract the subsystem in terms of input space and output space. We considered following criteria for abstraction mechanism.

a) Function parameters are mapped as input space and return variables are mapped as output space. From Figure 1, f_1, f_2 are mapped in as input and res is mapped as output.

b) If reference variables are used then they should mapped in both input and output space. Because reference variables cannot be used as returned variable so we include in output space.

c) File can be mapped as its characteristics i.e. if read from file operation occurs then it should include in input space and if write to file operation occur then it is included in output space.

d) If subsystem return a numeric value then numeric value implicitly assigned to a temporary variable and then return that value. If a subsystem return 0 then we implicitly include a temp variable, assign 0 and return the temp variable.

IV. IMPLEMENTATION PROCEDURE

From interaction model we get interacting variables between subsystems. After defining interacting variables from indirect dependencies, we generate definition-use of them. Definition-use of variable is used to find where variables are defined and used. The generation of definition-use is already done in our previous work [8]. Definition-Use is an essential part to find the data flow of the variables [9]. When data flow of interacting variable is found, the behavior of interacting subsystems can be well understood. By testing the data flow information errors can be found in interacting subsystems. To do so we generate DU chain for each of the interacting variables, which is the partial path information of indirect dependencies. A definition-use chain consists of all definition-use of variable according to the sequence of their appearance in...
the source code. Finally DU chain is used to generate test path for testing each of the particular interacting variables. This procedure is described in details below:

A. Find indirect dependencies

In order to find indirect dependencies we need to identify those variables which are passed as parameter or return from one subsystem to another. In this case we make an abstraction of the subsystems and assume that these subsystems are already unit tested. Indirect dependencies can be extracted by analyzing the source code. First we convert the source code to xml format. We use Doxygen, an open source documentation generation tools, to convert C source code to xml code. After that we parse the xml file with PHP xml parser and extract the variables which causes indirect dependencies [10]. We also use another specializing tools, Cscope which enables us to browse source code and find any particular item from the source code [11]. We have modified Cscope to find the indirect and implicit interacting variable. The algorithm is given below:

Algorithm 1: ComputeInteractingVar(S, G)

Input:
S: set of abstracted Subsystems
G: flow graph of the system
Output:
interactingVariable: array of interacting variables

Declare:
definiedBy: array of defined variables
usedBy: array of used variables
returnVar: array of returned variables
currentLine: Current position, initialized to zero
totalLine: LOC of source code

Begin:
While currentLine is not equal to totalLine do
Analyze each node of G to find function call from S, to

if function call is found then

get the parameter and defined variable of S, and
return variable of S

definedBy[k] := S, parameter
usedBy[k] := S, returnVariable
returnVar[k] := S, definedVariable

for each m from 0 to k do
for each n from 0 to k do
if definedBy[m] = returnVar[n] then
add (returnVar[n], interactingVariable)
endif
endfor
endif
k := k+1
endif

Increase currentLine by 1
endwhile
end ComputeInteractingVar

B. Definition Use of interacting variable

Before generating definition use of interacting variables, we first find the occurrence of each of the interacting variables. Occurrence is nothing but the statements where the interacting variables exists. Then analyzing the statement we generate the status of that variable. A variable can be defined, used or both defined-used in a statement. We construct an algorithm that take each statement and find the status of that variable. We have considered the following criteria:

- When a variable, x, is on the RHS of an assignment statement, such as \( v = x+y \) then it is a definition of variable \( v \).
- When a variable, \( v \), is on the both of an assignment statement, such as \( v = v+y \) then it is a definition-use of variable \( v \).
- When a variable, \( v \), use incremental symbols such as \( v++, ++v, v-- \) then it is a definition-use of variable \( v \).
- A variable, \( v \), pass as a value as a function parameter then it is a use of the variable. If \( v \) is passed as reference as a parameter then it is a definition of variable \( v \).
- When a variable, \( v \), is in other occasion like: function parameter, loop or switch condition etc. then it can be called as use of variable \( v \).

C. Partial path generation

Partial path is a definition-use chain that consists of a use, \( U \), of a variable, and all the definitions, \( D \), of that variable that can reach another use without any other intervening definitions. DU chain can be created by data flow analysis. Usually the chain consist of use and definition node. Two or more use node of variable can present successively but two definition node cannot be used successively. While generating definition-use of interacting variable we also extract the sequence of the occurrence of the interacting variables which is nothing but the line number of source code. In this case represent line number as a node. The Pseudo code of DU Chain generation is described as follows:

Step 1:
Set definition node, \( D = \sum_{i \in R} n_d \); where \( i \in R \) and \( n_d = number \ of \ definition \ node \).
Set use node, \( U = \sum_{i \in R} n_u \); where \( i \in R \) and \( n_u = number \ of \ use \ node \).

Step 2:
For each \( i=1 \), find first definition and add in chain.

Step 3:
For \( i>1 \), find sequential Use/Definition node such that it doesn’t contain successive definition node.
Make a link among definitions and use.
Set the node \( n_u \), as definition node and increase \( i \) by 1.
Kill previous definition.
Repeat step 3.

D. Test path generation

A path represents the flow of execution from the start node to end node. Path testing designed to execute all or selected paths through a computer program. The nature of software represented through graph, and it is essential to test all nodes and edges under the graph, for this type of verification, path testing is more reliable and suitable. Usually paths are generated from the control flow graph of source program [12] [13]. There exists many manual and automatic tools to construct a program’s control flow graph which are used to determine the basis set of linearly independent paths which contain each of the partial path.
V. CASE STUDY AND EVALUATION

In order to implement our technique we choose a patient monitoring system which has four subsystems and a main control unit. They are blood pressure, heart beat monitoring system, panic button control system and alarm/buzzer system. Each of the subsystems has their own functionality. Blood pressure unit calculate the blood pressure and heart beat monitoring system monitor the heart rate of the patient. If the patient has some emergency situation they can press the panic button and signal goes to the alarm or buzzer. The function of alarm system is to send signal to alarm or pager. All of the subsystems are control by central software unit and interact with each other through the main unit. For simplicity we eliminated hardware code from the source code and make an abstraction of the subsystems. In Figure 3 we have shown interaction model of patient monitoring system. This model includes all subsystems, direct calling, indirect dependencies and relationship by shared resources among them.

Figure 3. Block diagram of Interaction model of patient monitoring system

Here, \( S = \{BPMS, HBMS, PBCS, AS, SR\} \).

Table I shows necessary information of variables which have indirect dependency with other subsystem. For example: \( Systolic\_Pressure \) exists in several function of main, alarm and blood pressure module. Also we get the exact point where the variables is defined or used. This information is extracted automatically from the source code to calculate partial path information of interacting variables. For generating DU chain we analyze the sequence of function where the interacting variable are defined and used.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Module</th>
<th>Function</th>
<th>Line no</th>
<th>Expression</th>
<th>Def/Use</th>
<th>Partial Path info</th>
</tr>
</thead>
<tbody>
<tr>
<td>Systolic_Pressure</td>
<td>main</td>
<td>Measure_Systolic</td>
<td>163</td>
<td>( Systolic_Pressure = \text{Convert_Voltaget} )</td>
<td>U</td>
<td>12-163-313-314-214-15</td>
</tr>
<tr>
<td></td>
<td>alarm</td>
<td>alarmMain</td>
<td>15</td>
<td>( \text{if}(Systolic_Pressure &gt; 160</td>
<td></td>
<td>\text{Diastolic_Pressure} &gt; 160) )</td>
</tr>
<tr>
<td></td>
<td>bloodPressure</td>
<td>Measure_Systolic</td>
<td>169</td>
<td>return ( Systolic_Pressure )</td>
<td>U</td>
<td></td>
</tr>
<tr>
<td></td>
<td>bloodPressure</td>
<td>Calculate_Diastolic</td>
<td>214</td>
<td>( \text{Diastolic_Pressure} = 1.5*\text{MAP} – \text{Sys} )</td>
<td>U</td>
<td></td>
</tr>
<tr>
<td></td>
<td>bloodPressure</td>
<td>bloodPressureMain</td>
<td>313</td>
<td>( Systolic_Pressure = \text{Measure_Systolic} )</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>Diastolic_Pressure</td>
<td>main</td>
<td>Calculate_Diastolic</td>
<td>214</td>
<td>( \text{Diastolic_Pressure} = 1.5*(\text{MAP} - \text{Sys}) )</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td></td>
<td>alarm</td>
<td>alarmMain</td>
<td>15</td>
<td>( \text{if}(\text{Systolic_Pressure} &gt; 160</td>
<td></td>
<td>\text{Diastolic_Pressure} &gt; 160) )</td>
</tr>
<tr>
<td></td>
<td>bloodPressure</td>
<td>Calculate_Diastolic</td>
<td>214</td>
<td>( \text{Diastolic_Pressure} = 1.5*(\text{MAP} - \text{Sys}) )</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td></td>
<td>bloodPressure</td>
<td>Calculate_Diastolic</td>
<td>221</td>
<td>( \text{Xd} = \text{Diastolic_Pressure} )</td>
<td>U</td>
<td></td>
</tr>
<tr>
<td>iterationBloodPressure</td>
<td>main</td>
<td>int iterationBloodPressure;</td>
<td>15</td>
<td>( \text{int iterationBloodPressure;} )</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td></td>
<td>main</td>
<td>bloodPressureMain</td>
<td>20</td>
<td>( \text{bloodPressureMain(Diastolic_Pressure)} )</td>
<td>U</td>
<td></td>
</tr>
<tr>
<td></td>
<td>bloodPressure</td>
<td>bloodPressureMain</td>
<td>271</td>
<td>( \text{iterationBloodPressure++;} )</td>
<td>U/D</td>
<td></td>
</tr>
<tr>
<td>iterationHeartRate</td>
<td>main</td>
<td>int iterationHeartRate;</td>
<td>16</td>
<td>( \text{int iterationHeartRate;} )</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td></td>
<td>main</td>
<td>finalPulse = heartRateMain(iterationHeartRate);</td>
<td>22</td>
<td>( \text{finalPulse = heartRateMain(iterationHeartRate++)} )</td>
<td>U</td>
<td></td>
</tr>
<tr>
<td>heartRate</td>
<td>heartRateMain</td>
<td>int iterationHeartRate;</td>
<td>372</td>
<td>( \text{iterationHeartRate++;} )</td>
<td>U/D</td>
<td></td>
</tr>
</tbody>
</table>
In similar fashion we generate partial path information of interacting reference variable in TABLE II. In this case we also find the referred variable because when reference of interacting variable is used, variable name can be changed.

Finally, for data flow testing, we test path that cover all or a threshold of all DU chains. Sometime it is not possible to cover all DU chain for semantic reason so we can break the chain in parts and generates test cases for them.

Figure 4 shows the number of DU chain covered by random testing approach and our proposed approach. For example, for a variable, \textit{Systolic\_Pressure}, random testing approach covers 3 DU pair where our approach covers 5 DU pair. So, it clearly shows that using our approach DU coverage rate is always higher than random testing.

![Figure 4. Comparison of number of covered DU pair](image)

TABLE III. COMPARISON OF DU COVERAGE BETWEEN RT TECHNIQUE AND PROPOSED APPROACH

<table>
<thead>
<tr>
<th>Approach</th>
<th>No of Variable</th>
<th>Interacting module</th>
<th>DU pair</th>
<th>Covered DU pair</th>
<th>DU coverage (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RT</td>
<td>5</td>
<td>4</td>
<td>21</td>
<td>13</td>
<td>61.9%</td>
</tr>
<tr>
<td>Proposed</td>
<td></td>
<td></td>
<td></td>
<td>21</td>
<td>100%</td>
</tr>
</tbody>
</table>

DU coverage = \( \frac{\text{No. of DU covered}}{\text{Total No. of DU}} \times 100\% \) (1)

Figure 4 depicts that for 5 interacting variable we have found 21 definition-uses. Existing random testing approach covers 13 DU chain while using our proposed approach cover all 21 DU chain. According to equation (1), DU coverage for random testing approach is 61.9% whereas our approach has 100% DU coverage. TABLE III shows the comparison between random testing approach and proposed approach. Finally we have shown a qualitative comparison between our approach and other related works in TABLE IV.

TABLE IV. QUALITATIVE COMPARISON WITH EXISTING WORKS

<table>
<thead>
<tr>
<th>Approach</th>
<th>Interaction type</th>
<th>Number of test cases</th>
<th>Coverage criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td>W. Miao and S. Liu</td>
<td>Direct</td>
<td>Moderate</td>
<td>Scenario based</td>
</tr>
<tr>
<td>H. Jueliang, D. Zuohua and P. Geguang</td>
<td>Direct</td>
<td>High</td>
<td>Path based</td>
</tr>
<tr>
<td>S.A Khan and A. Nadeem</td>
<td>Direct</td>
<td>Moderate</td>
<td>Coupling based</td>
</tr>
<tr>
<td>The proposed approach</td>
<td>Indirect</td>
<td>Minimal</td>
<td>Data flow based</td>
</tr>
</tbody>
</table>

VI. CONCLUSION AND FUTURE WORK

In this paper we presented a general specification of an interaction model, which includes all possible direct and indirect interaction between subsystems, and use data flow driven integration testing approach to generate minimal test cases efficiently of a patient monitoring system. At first we consider all direct and indirect interaction to specify an interaction model. From this interaction model we obtain the interacting variable between subsystems. After that for each of the interacting variable we produce definition use information and generate partial path information i.e. DU chain of them. For generating DU chain we extract several information i.e. calling module, calling function, function sequence etc. from the source code. Finally test cases can be generated to cover the
test path that cover all or a threshold of DU chain. To show the effectiveness of the proposed approach we applied our technique to the patient monitoring system. And in the evaluation part we have shown that DU coverage of our approach 100% where random testing approach has 61.9%.

In the future work, we are planning to implement our test technique as a tool suite that automatically generates test data for interacting variables between subsystems of an embedded system. Also we are looking forward to find more interaction pattern to generate more efficient interaction model that can be implemented in larger embedded system.

ACKNOWLEDGMENT

This research was supported by the MSIP (Ministry of Science, ICT & Future Planning), Korea, under the C-ITRC (Convergence Information Technology Research Center) support program (NIPA-2014-H0401-14-1004) supervised by the NIPA (National IT Industry Promotion Agency) and the IT R&D program of MSIP/IITP. [10041145, Self-Organized Software platform (SoSp) for Welfare Devices].

REFERENCES