TEST CASE GENERATION FOR INTEGRATING MEDICAL SYSTEMS
CONSIDERING FUNCTION CHARACTERISTICS

Youngsul Shin, Muhammad I. Hossain, Woo Jin Lee
School of Computer Science and Engineering, Kyungpook National University
1370, Sankyuk-dong, Buk-gu, Daegu, Korea
youngsulshin@gmail.com, milon_mi7@yahoo.com, woojin@knu.ac.kr

ABSTRACT
A medical system is complicated because it has many sub-modules and is connected with many kinds of other systems. Because software faults tend to occur in interoperation among modules, a safety-critical medical system should be tested strictly in terms of integration. At integration testing phase, there are too many execution path of software to test, that makes testing difficult. To deal with the problem, existing test approaches ignore complicated internal information of modules at integration testing phase. However, the existing approach may not find faults of interoperation among modules because they cannot define test cases that cover many execution paths. Our paper proposes a method to test interoperation among modules by abstraction of function characteristics and reuse of test cases. Abstraction of function characteristics generates test cases that bring internal information of functions. By reusing the generated test cases, integration testing can consider internal information of functions.

KEY WORDS
Complicated Medical Software, Required Interface, Abstraction of Function Behavior, Function Behavior Reduction, Test Case Reuse, Integration Testing.

1. Introduction

In late 2000 and early 2001, the overexposure of 28 radiation therapy patients was occurred at National Cancer Institute, Panama City [1]. Therapy control software was received the shielding blocks transferred by computer screen and miscalculated the dosage of radiation. The interoperation fault between the control component and the screen input component would be found if testing was effective.

However, testing complicated medical systems is very difficult, because medical systems have many sub-components and are connected with other systems. For example, IEEE 11073 standards define communication and connectivity among medical devices [2]. Moreover, Software faults tend to be biased to interoperation among components. In this case that many components are connected, integration testing is important to provide safety with medical systems.

Integration testing usually loses internal information of components [3]. Unit testing called white-box testing is used to test a component and can examine execution path of a component in detail [4]. Integration testing called black-box testing does not treat an internal behavior of a component because test paths are increased in geometric progression if components are integrated [5]. Therefore, test cases cannot be defined to strictly test components used by other components. For example, suppose that a caller function invokes a callee function, where the callee function has already passed testing. When a developer tests the caller function, the developer ignores internal execution paths of the callee function and treats the callee function as one statement although the callee function has many execution paths inside.

We propose a method to generate test cases considering internal characteristics of callee functions. Abstraction of callee functions generates test cases which reflect characteristics of the callee functions conforming with each of test coverage criteria. The generated test cases for the callee functions provide the characteristics of the callee functions with caller functions. Test cases for the caller functions can be generated considering the characteristics of the callee functions by reuse of the generated test cases for the callee functions.

The rest of the paper is organized as follows. Test cases reflecting characteristics of callee functions are described by an example in Section 2. A method to characterize callee functions is presented in Section 3. Testing using characteristics of callee functions is described in Section 4. A case study is presented in Section 5. Finally, a conclusion and future work are given in Section 6.

2. Background

2.1 Required Interface

Unified Modeling Language (UML) is a semi-formal language to model a system in an object-oriented approach. UML is a de facto standard as it supports a formal approach and is easy to use. A complicated system like a medical system has modular architecture that is divided into independent modules called a component in UML [6]. A component is a modular part of a system that is encapsulated its internal information by an interface. A component can communicate with other components through its interface that consists of functions. A
component has two kinds of interfaces, provided interface and required interface. Functions of a provided interface are called by other functions that consist of a required interface. Figure 1 shows an example of components with interfaces. The component A has a required interface represented by a half circle and the component B has a provided interface represented by a circle. Functions of the required interface of the component A invoke functions of the provided interface of the component B.

![Figure 1. Interfaces of components](image)

### 2.2 Testing a Required Interface

A callee function is usually tested earlier than a caller function. After testing the callee function, a caller function can use the callee function. When a developer tests the caller function, the developer cannot know an internal architecture of the callee function at integration testing phase and handles the callee function as a simple statement [7]. The developer may think that the behavior of the callee function is understood completely. The developer may test interoperation between the callee function and the caller function with scenarios of the callee function behavior that the developer understands.

The testing approach above cannot strictly test complicated medical systems because the function behavior used by other functions is ignored and handled as a black-box. A patient-controlled analgesia infusion pump (PCA infusion pump) is given as an example of a medical system. The PCA infusion pump infuses fluids into patients continuously. When the patients feel a high level of pain, they trigger a bolus dose by pressing a button. The pump is programmed with a basal rate and bolus dose using a user console input device by doctors [8]. Figure 2 shows a function to configure a basal rate of the pump controller. The `configPump()` returns three kinds of return values depending on a basal rate input.

```c
int configPump(int rate) {
    if(rate = 0.5) return 1;  // normal
    else if(0.5 < rate < 12) return 0;  // warning
    else return -1;  // exception
}
```

Figure 2. A function to configure the pump controller

Two functions to call the `configPump()` are shown in Figure 3. The function of `console1()` does not understand the function of `configPump()` completely and does not care the return value of warning status. A developer of the `console1()` may not test interaction behavior for warning status between `console1()` and `configPump()`. However, the function of `console2()` handles warning status as well as normal and exception status. A developer of the `console2()` can give safety to the PCA infusion pump system.

```c
int console1(int rate) {
    int result;
    ... result = configPump(rate);
    if(result == 1) {  // normal
        ...
    } else {  // exception
        ...
    }
}
```

```c
int console2(int rate) {
    int result;
    ... result = configPump(rate);
    if(result == 1) {  // normal
        ...
    } else if(result == 0) {  // warning
        ...
    } else {  // exception
        ...
    }
}
```

Figure 3. Functions to call the `configPump()`

Existing test approaches did not consider characteristics of callee function that can increase safety and decrease development risk of complicated medical systems. A developer should examine whether callee functions are used correctly. Our paper proposes a method to abstract and apply characteristics of callee functions to testing.

### 3. Characterizing Function Behavior

Several coverage criteria have been using for selecting test paths such as normal path selection, branch coverage, condition coverage, and etc. The normal path is a representative path of a function. In branch coverage, every branch of the graph is visited at least one time. In condition coverage, the conditions should be true for at least one test case. This paper proposes a method to generate test cases to cover return value coverage criteria by abstracting function characteristics.

Return value coverage is based on return statements. Return values of a function present a function behavior in terms of other functions that exchange data with it. Test paths can be also reduced on the basis of return statements. Therefore, test cases for the function can be generated...
based on return statements. A procedure to abstract function characteristics is shown in Figure 4.

Figure 4. A procedure to abstract function characteristics

The PCA infusion pump is given as an example. Figure 5 shows the flow chart of the `configPump()` source code to configure the pump controller with parameters. All blocks of the flow chart are transformed to nodes of the control flow graph as shown in Figure 6 to generate the control flow graph from the flow chart [9][10].

Figure 5. Flow chart of the `configPump()`

The control flow graph has nodes that do not affect the value of the return variables [11][12]. A slicing technique is used to identify and eliminate the unnecessary nodes [13]. Figure 7 shows the graph from which the unnecessary nodes are eliminated on the basis of the return statement of the node 28.

Figure 6. Control flow graph of the `configPump()`

After eliminating the unnecessary nodes, the depth first search algorithm generates all possible paths from an entry node to the return statement node. Figure 8 is the possible set of all path generated by the depth first search algorithm.

The proposed method to abstract function characteristics selects the shortest path with maximum input coverage. Among the generated paths by the depth first search algorithm, the abstraction method selects minimum number of paths which contains less number of nodes, i.e. shortest path. The abstraction method also uses maximum
input parameter coverage criteria to make test paths effective to find faults. The shortest path can be found by using Dijkstra’s algorithms [14].

\[
\begin{align*}
1 &\rightarrow 2 \rightarrow 4 \rightarrow 20 \rightarrow 22 \rightarrow 24 \rightarrow 28 \\
1 &\rightarrow 2 \rightarrow 3 \rightarrow 2 \rightarrow 4 \rightarrow 20 \rightarrow 22 \rightarrow 24 \rightarrow 28 \\
1 &\rightarrow 2 \rightarrow 3 \rightarrow 2 \rightarrow 4 \rightarrow 20 \rightarrow 21 \rightarrow 23 \rightarrow 25 \rightarrow 23 \rightarrow 21 \rightarrow 20 \rightarrow 22 \rightarrow 24 \rightarrow 28 \\
1 &\rightarrow 2 \rightarrow 4 \rightarrow 20 \rightarrow 21 \rightarrow 23 \rightarrow 25 \rightarrow 23 \rightarrow 21 \rightarrow 20 \rightarrow 22 \rightarrow 24 \rightarrow 28 \\
1 &\rightarrow 2 \rightarrow 3 \rightarrow 2 \rightarrow 4 \rightarrow 20 \rightarrow 21 \rightarrow 23 \rightarrow 25 \rightarrow 23 \rightarrow 21 \rightarrow 20 \rightarrow 22 \rightarrow 24 \rightarrow 28 \\
1 &\rightarrow 2 \rightarrow 4 \rightarrow 20 \rightarrow 21 \rightarrow 23 \rightarrow 25 \rightarrow 23 \rightarrow 21 \rightarrow 20 \rightarrow 22 \rightarrow 24 \rightarrow 28 \\
\end{align*}
\]

…

Figure 8. Generated paths by depth first search

Once the shortest paths are decided, test cases are generated to cover the decided paths by symbolic execution. The test case in terms of the return statement of the node 28 covers a path as follow:

\[
1\rightarrow 2\rightarrow 4\rightarrow 20\rightarrow 21\rightarrow 23\rightarrow 25\rightarrow 23\rightarrow 21\rightarrow 20\rightarrow 22\rightarrow 24\rightarrow 28.
\]

The abstraction method can also generate test cases to satisfy existing test coverage, for example, statement coverage, path coverage, branch coverage, decision coverage, condition coverage, because it can change the abstraction levels by using the slicing of the control flow graph. Table 1 presents test cases generated by the abstraction, which satisfy a variety of test coverage.

Table 1. Test cases satisfying a variety of test coverage

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Test Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>P_1</td>
<td>1,2,4,20,22,24,18</td>
</tr>
<tr>
<td>P_2</td>
<td>1,2,3,2,4,20,22,24,28</td>
</tr>
<tr>
<td>P_3</td>
<td>1,2,3,2,4,20,22,24,27,24,28</td>
</tr>
<tr>
<td>P_4</td>
<td>1,2,4,20,22,24,27,24,28</td>
</tr>
<tr>
<td>P_5</td>
<td>1,2,3,2,4,20,22,24,27,24,27,24,28</td>
</tr>
<tr>
<td>P_6</td>
<td>1,2,3,2,4,20,22,24,27,24,27,24,28</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>

A test case can satisfy multiple test coverage because the test coverage has a coverage hierarchy. For example, test cases for branch coverage may be replaced by a test case for condition coverage. The abstraction method is assigned test coverage by a developer. The test coverage may be the proposed return statement coverage or existing test coverage. The abstraction method generates new test cases according the assigned test coverage or selects some test cases among test cases that have already generated.

Test cases according to abstraction levels are shown in Table 2. Once a developer chooses the condition coverage, test cases are generated according to the condition coverage. There are 9 test cases for the condition coverage in Table 2. If the developer wants to increase abstraction level to the branch coverage, the proposed abstraction method may generate test cases for the selected abstraction level. However, test cases have already generated to satisfy the selected abstraction level and the abstraction method decides a test case of \( P_4 \) among the test cases for the selected abstraction level.

Table 2. Test cases according to abstraction levels

<table>
<thead>
<tr>
<th>Normal Path Coverage</th>
<th>Return Value Coverage</th>
<th>Branch Coverage</th>
<th>Condition Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>P_1</td>
<td></td>
<td></td>
<td>P_1</td>
</tr>
<tr>
<td>P_2</td>
<td></td>
<td></td>
<td>P_2</td>
</tr>
<tr>
<td>P_3</td>
<td></td>
<td></td>
<td>P_3</td>
</tr>
<tr>
<td>P_4</td>
<td></td>
<td></td>
<td>P_4</td>
</tr>
<tr>
<td>P_5</td>
<td></td>
<td></td>
<td>P_5</td>
</tr>
<tr>
<td>P_6</td>
<td></td>
<td></td>
<td>P_6</td>
</tr>
<tr>
<td>P_7</td>
<td></td>
<td></td>
<td>P_7</td>
</tr>
<tr>
<td>P_8</td>
<td></td>
<td></td>
<td>P_8</td>
</tr>
<tr>
<td>P_9</td>
<td></td>
<td></td>
<td>P_9</td>
</tr>
</tbody>
</table>

4. Test Using Function Characteristics

Function characteristics provide internal information of functions and reusability of test cases. A procedure to generate test cases and execute testing using function characteristics is shown as Figure 9. The source code of the callee function is written and abstracted by the testing criteria. Abstraction of the callee function generates test cases for the callee function. The generated test cases are used to examine the callee function. After the callee function passes testing, the callee function can be used by other functions.

Once a developer writes the source code of the caller function that invokes the callee function, the caller function should be tested to examine interaction with the callee function using the generated test cases for the callee function. After test cases are selected among the test cases for the callee function by the test criteria, test cases for the caller function are generated using symbolic execution and the caller function is tested by the generated test cases. However, expected values of the test cases are not generated. The expected values should be defined by a developer by reference to a specification of the caller function. If the caller function is invoked by other function after passing testing, the caller function is a callee function of other functions.

The function of console() delivers values of configuration parameters for the PCA infusion pump from the user console to the pump controller as shown in Figure 9. The console() invokes the function of configPump() to configure the pump with the parameters. When the source code of the console() is written, the test cases for the configPump() are given because the configPump() passes testing before the console() is written. Therefore, the console() can be tested considering the callee function of the configPump() by reusing the test cases for the configPump().
When testing the `console()` function, a developer can understand an internal characteristic of the `configPump()` function because the developer is given the abstraction table as shown in Table 2. The abstraction table brings the generated test cases for the `configPump()`. The test cases represented by the abstraction table bring the characteristics of the `configPump()`. Once test cases for the `configPump()` are selected from the abstraction table by the return value coverage criterion, test cases for the `console()` are generated to execute the test cases selected for the `configPump()` according to the return value coverage criterion. The `console()` passes interaction testing if all of the selected test cases are executed successfully and the expected values are delivered.

```c
int console(int rate) {
    int result;
    ...
    result = configPump(rate, loading, bolus);
    if(result == 1) {  // normal
    ...
    } else if(result == 0) {  // warning
    ...
    } else {  // exception
    ...
    }
}
```

Figure 10. A function `console()` to call the `configPump()`

5. Case Study

Openhealth is a research project for development solutions of eHealth inside of mobile environments. These solutions are based on the management wireless biomedical devices in body area networks under the IEEE 11073-20601 standard and open mobile terminal platform, which are implemented in Java [17]. This paper proposes a method to reuse test cases in unit testing. Since we did not write the source code of Openhealth project, we should have analyzed the source code thoroughly with the IEEE 11073-20601 standard for case study. We take `decodeEnum()` that has a complicated control flow to effectively show the proposed method. The function of `decodeEnum()` is called by two functions as shown in Figure 11. The `decodeEnum()` also invokes other functions.

To verify the source code of the `decodeEnum()` at the phase of unit testing, the `decodeEnum()` is inspected by a developer to write the source code of the `decodeEnum()`. The developer can generate test cases based on the control flow graph of the `decodeEnum()` as shown in Figure 12. The control flow graph has 20 nodes which are code blocks. An initial node is the node which label is 1. The two filled nodes are final nodes which labels are 11 and 20. The `decodeEnum()` returns a class type of `DecodedObject` in normal status or `null` value in exception status.

```c
Decoder.decodeClassType()
ASN1EnumMetadata.decode()
Decoder.decodeEnum()
```

Figure 11. The function of `decodeEnum()`

Figure 12. Control flow graph of the `decodeEnum()`
The abstraction method of a function behavior generates test cases that satisfy test coverage selected by the developer using symbolic execution and linear programming. We generated test cases of the `decodeEnum()` according to condition coverage, branch coverage and return statement coverage as shown in Table 3. Each of test paths is executed by one test case. The function of `decodeEnum()` has the same number of test cases for condition coverage and branch coverage. The test cases for return coverage are the TP1 and the TP3. The test case that executes the test path of TP2 covers the same return statement with the test case for the TP1. However, the return coverage assumes the maximum coverage of input parameters. Therefore, the test case for TP1 is not included in return coverage.

<table>
<thead>
<tr>
<th>Test Coverage</th>
<th>Test Path</th>
<th>Number</th>
<th>Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Condition Coverage</td>
<td>TP1</td>
<td>1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 13, 12, 13, 14, 15, 16, 17, 18, 19, 20</td>
<td></td>
</tr>
<tr>
<td>Branch Coverage</td>
<td>TP1</td>
<td>1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 13, 12, 13, 14, 15, 16, 17, 18, 19, 20</td>
<td></td>
</tr>
<tr>
<td>Return Coverage</td>
<td>TP1</td>
<td>1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 13, 12, 13, 14, 15, 16, 17, 18, 19, 20</td>
<td></td>
</tr>
</tbody>
</table>

After the function of `decodeEnum()` passes unit testing, it is referenced by the function of `decodeClassType()`. At unit testing phase of the `decodeClassType()`, the developer of the `decodeClassType()` should verify interaction with the function of `decodeEnum()`. However, the developer does not know the internal information of the `decodeEnum()` and cannot write significant test cases. We solved the problem by reusing test cases generated by abstraction of a function behavior. When testing the `decodeClassType()`, we selected branch coverage and return coverage. The three test cases have been generated already and have satisfied branch coverage and return coverage. Abstraction of a function behavior is also applied to generate test cases for the `decodeClassType()`, which execute the selected test cases for the `decodeEnum()`.

We concluded that the abstraction method of a function behavior generates test cases according to test criteria and the reuse of the generated test cases helps developers verify the source code with significant test cases. However, existing constraint solvers should be enhanced to generate more reliable test cases.

6. Conclusion and Future Work

Our paper proposed a testing method to consider characteristics of callee functions. The proposed method enables strict testing for complicated medical systems because developers can examine interaction among modules with internal information of them. The method includes abstraction of functions and reuse of test cases. Abstraction of callee functions generates test cases by test criteria. The generated test cases deliver internal information of the callee functions to the caller functions. When testing the caller functions at integration testing phase, a developer can test the caller functions considering the characteristics of the callee functions because the test cases for the callee functions are given. The case study shows that the abstraction method and the reuse of test cases are useful for unit testing and testing tool with enhanced constraint solvers is necessary.

Our future work is to handle multiple callee functions at the same time. It is difficult to generate test cases that cover multiple callee functions by reusing the given test cases. Handling multiple callee functions makes testing more reliable. We will apply the proposed method to several domains of medical systems.

Acknowledgements

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (2011-0004759), and Basic Convergence Technology R&D Program through the Daegu Technopark.

References


