This paper is part of the Proceedings of the 15th International Conference on Railway Engineering Design and Operation (CR 2016)
www.witconferences.com

Proposal of a software coding analysis tool using symbolic execution for a railway system

H.-J. Jo & J.-G. Hwang
Korea Railroad Research Institute (KRRI), Korea

Abstract

The railway system is being converted to a computer system from the existing mechanical device, and the dependency on software is rapidly increasing. Though the size and degree of complexity of software for railway systems are slower than the development speed of hardware, it is expected that the size will gradually grow bigger and the degree of complexity will also increase. Accordingly, the validation of reliability and safety of embedded software for the railway system started to become an important issue. Accordingly, various software tests and validation activities are highly recommended in railway software related international standards. In this paper, we present a software coding analysis tool using symbolic execution for a railway system, and present the result of its implementation.
Keywords: railway system, software validation, reliability, safety.

1 Introduction

Recently, the functions and complexity of embedded software (SW) for a railway system have been rapidly increasing, and the risk cost caused by the occurrence of SW error is relatively increasing. Especially since the fatal error due to the malfunction of SW during railway operation is directly connected to human accidents, the validation on SW shall be performed using various methods and its evaluation and verification must be available. Procedures for validating functional safety among railway system developments are being standardized as IEC 61508 and IEC 62279, etc., and especially, in the case of IEC 62279, it requires railway SW development and the safety management

/evaluation to which this standard was applied by strengthening the safety validation more and revising it recently, etc. [1, 2].

However, despite the fact that the type approval for rolling stock SW became mandatory in accordance with the revision of the Railroad Safety Act, it is still suspended because of realistic difficulties of domestic railway manufacturers in technical levels for SW development and validation to satisfy the level required by existing international standards [3]. And, because the result of safety evaluation by international standards becomes the index of product reliability, there is difficulty in domestic operation and overseas expansion if evaluation is not accomplished. Therefore, not only document validation on the safety activities required by international standards, but also concrete supporting of technology development to cope with analysis and validation through SW test are very badly needed for the actual embedded SW development product for the railway system.

Especially, in the case of the test coverage of a SW code which can derive quantitative test results among railway SW validation items required by international standards, it is prescribed as a 'HR Highly Recommended' condition, as a verification and testing item in the vital railway system SW whose railway SW Safety Integrity Level (SIL) grade is mostly classified as 3 or 4 [2]. Accordingly, in the case of a railway system's SW embedded devices, certification on safety can be obtained only if the test coverage of development SW is accomplished nearly up to 100%. To enhance test coverage, this paper wants to propose a SW source code analysis tool by using a symbolic execution method and show its developed results.

2 Proposed SW source code analysis tool

2.1 Testing technology based on the SW source code

Generally, there is a control flow analytical method as the testing technology for the developed SW source code, and the coverage can be measured and reported through this method as to whether the area, branch and conditional statement, etc. of the source code can be executed. Coverage is used as the representative measure which can validate SW qualitatively even in the other industry fields, and furthermore, it makes SW quality guessed at [4]. According to IEEE Std. 1008-1997, it is stated that all of the SW must be validated by test cases in the SW unit test stage, and the statement and branch coverage of SW codes shall be satisfied by the test case [5]. To measure the test coverage with unit test performance, this standard recommends using automated means. In addition, it is recommended using methods such as statement and branch coverage, etc. for the validation on SW implementation stage in the IEC Std. 60880-2006 also [6], and in actual industrial fields too, the statement and branch coverage is widely utilized already.

And, in RTCA/DO-178B which is the standard for the aviation industry, the SW grade was classified according to the degree of importance in accordance with the kind of accident verified through system safety evaluation, and the code coverage requirements were defined according to the grade [7]. According to [7], it is defined that the grade with a higher degree of importance to which safety is required must satisfy the modified condition/decision coverage (MC/DC), and the cases that prove the applicability are presented also [8, 9]. Therefore, the modified condition/decision coverage must also be satisfied in the vital railway system SW whose SIL grade is classified as 3 or 4. For this purpose, this paper would like to inspect compatibility with SW requirements and the practicability of source code routes and to apply the symbolic execution method which can enhance the coverage in accordance with the reinforcement of test cases. The implementation of the railway system SW source code analysis tool developed by this paper is that to which the symbolic execution method was applied [10, 11], and the concrete contents developed are as shown in sections 2.2. and 2.3.

2.2 Symbolic execution method to enhance source code test coverage

The symbolic execution method which we would like to propose, when the partial function of the source code was performed, extracts the conditional expression with which each variable must satisfy, and provides the function that can obtain various analytical results on source codes by obtaining the solution by adding conditional expressions which will be used for the analysis together with the conditional expression extracted as the additional function. Those functions for applying a symbolic execution method to enhance the source code test coverage are arranged as shown in Table 1.

Table 1: List of functions for symbolic execution application method.

Function name	Description of function
Sequential program input	It checks if the given input is suitable for the form of sequential program, and provides the result of inspection. In the case of the sequential program, it provides the parsed result so that the corresponding program can access this developed analysis tool.
Calculation of symbolic state	Starting from the beginning syntax of a given function of the sequential program to the ending syntax, it obtains the condition of route to reach each syntax sequentially, and collects values of symbolic variables.
Inspection on practicability to program route	Until the end of a given function of the sequential program, it inspects if execution itself is possible from the symbolic state collected for performance.
Inspection on compatibility with requirements	It inspects if it is satisfied with the conditional expression which wants to check matters after final performance of a given sequential program.
Report on results	In cases where the conditional expression used in the inspection is satisfied, it provides input values of variables which satisfy the corresponding conditional expression.

Syntax for control flow does not exist in the sequential program input from the developed analysis tool, and all of the decision conditions existing in the route to be performed originally are modified in the form of "assume(<condition>)". For example, if the following C-language program is assumed, Figure 1(a) is the source code before being converted to the sequence language, and Figure 1(b) is the source code of the converted sequential program. That is, when assuming the route which passes through the if-then clause in the exercise, the sequential program which can be drawn is the result of Figure 1(b).

```
void foo(int a)
{
   int b = 1;
   int c = 3;
   if ( a > b ) {
     b = a;
   } else {
     a = b;
      b = c;
```

(a)

```
void foo(int a)

{

     int b = 1;

     assume(a > b);

     b = a;

     return a + b;
```

(b)

Figure 1: (a) Exercise source code before conversion to sequence language. (b) Source code of converted sequential program.

From the above result, we may see that the if-statement was replaced with the condition of assume-function. In the same method, the route condition that passes through this route can be generated easily by gathering conditions within the assume-statement. And for the variables not used, we may see that slicing is accomplished. That is, since the C variable is used in the if-else only, variables not used in the sequential program for if-then are excluded. Each description due to the main functions of this analysis tool is classified as follows.

1) Generation of project
This module is expected to carry out repetitive tasks to find suitable values by performing it to the given sequential program and the conditional expression to be validated. Thus, it supports repetitive tasks by composing and managing the project for the sake of input factors. Although the project is restricted to not edit the content after generation, it can support repetitive tasks sufficiently because the conditional expression to be inspected is allowed to be edited whenever it is performed.

2) Input(editing) of conditional expression
It resets the project being used currently by setting the new conditional expression or editing, modifying the existing conditional expression.

3) Sequential program parsing
It converts the sequential program source code to the form of AST that can be recognized in this analysis tool by parsing the sequential program source code in the state of C-language to be input for the purpose of symbolic performance. Sequential program can use all of the expressions excluding control phrases (if, for, goto, switch, case, break, etc.) of general C-language. Therefore, sufficient parsing ability to support programming languages is provided. And in the case of the program for which parsing is impossible, the reason for non-recognition is provided in the sequential program parser.

4) Performance of symbolic execution based on the parsing result
It obtains the function which is subject to be performed from the parsed sequential program which was already analyzed, and performs symbolic execution in accordance with each syntax that composes the function. After a syntax was carried out, it renews the route condition to perform up to the corresponding syntax and symbolic values used until now. The current route condition generates a new route condition by combining route conditions up to the previous syntax with conditions occurring in the course of performing the current syntax. Symbolic values reset new values through calculation on the basis of symbolic values of previous syntax.

5) Calculation of values satisfying the conditional expression
- To obtain the value which satisfies the symbolic route condition: After completing the performance by using the constraint verifier library, final symbolic values and final values satisfying the route condition shall be obtained. In cases where there is any value that satisfies the corresponding condition, corresponding input values are provided.
- Inspection on compatibility of conditional expression: After completing symbolic performance, by inserting the given conditional expression together with the route condition, additional conditional expressions are generated, and obtains if there is any value satisfying it through the constraint verifier.

2.3 Result of development of the SW analysis tool for a railway system

The result of developing a source code analysis tool developed by applying the proposed symbolic execution method is as follows, and first of all, implemented contents of screen design for analysis tool and user interface implementation method are as shown in Figure 3. Implementation of the screen for a symbolic execution application analysis tool largely consists of four views, such as the Project View, Properties View, Content View and Output View.

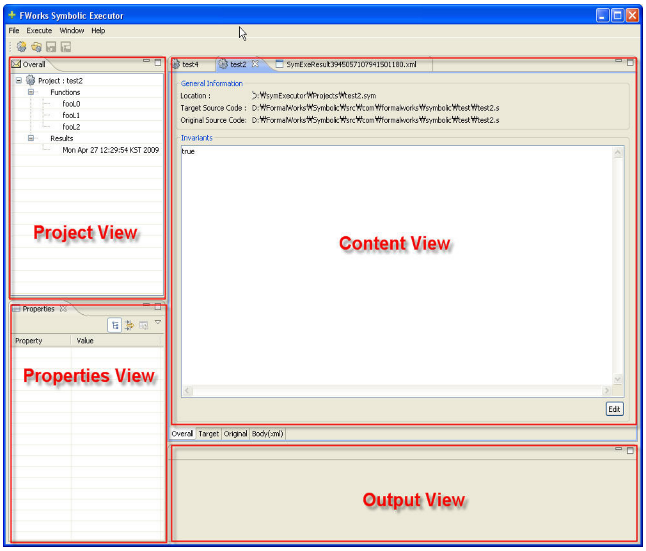
- Project View

It is the screen showing the outline of the project defined in this tool, and it provides the tree-based screen. It considers the project as the highest node, and has the child nodes called as Functions and Results. The Functions node has the function defined in the sequential program that wanted to be performed in the project as its child. The Results node has the result of symbolic execution carried out for this project as its child. Children have the performed date and time as their names. In cases where the Project which is the highest node and the result of each performance are selected in this screen, Project Content View or Result Content View is shown in the Context View area.

– Properties View
Properties View is the UI component provided by the Eclipse, and it shows the property defined for selected item if the property for Project and Result is defined.

– Output View
Output View is the view showing the simple text, and it shows various messages occurring during the work of this module.

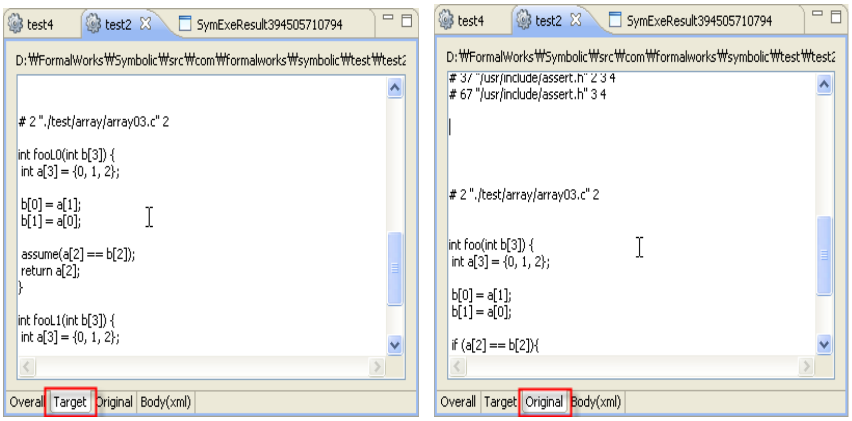
– Content View
Content View consists of four views such as the Project Info View, Target Source View, Original Source Code View and Project File View again, and the Info View presents the content of Invariant established in the Project together with general information, such as the location of file corresponding to the Project on the screen. Target Source View shows the content of source code which is subject to be performed and established in the Project symbolically, and the Original Source View shows the input file if any input file before being modified to the sequential program exists. Finally, Project File View shows the content itself of the Project in the form of XML.



Location : D:₩symExecutor₩Projects₩test2.sym
Target Source Code : D:₩FormalWorks₩Symbolic₩src₩com₩formalworks₩symbolic₩test₩test2.s
Original Source Code: D:₩FormalWorks₩Symbolic₩src₩com₩formalworks₩symbolic₩test₩test2.s

true

Figure 2: Result of implementing screen design for the source code analysis tool.



```
D:₩FormalWorks₩Symbolic₩src₩com₩formalworks₩symbolic₩test₩test2

# 2 "./test/array/array03.c" 2

int fooL0(int b[3]) {
 int a[3] = {0, 1, 2};

 b[0] = a[1];
 b[1] = a[0];

 assume(a[2] == b[2]);
 return a[2];
}

int fooL1(int b[3]) {
 int a[3] = {0, 1, 2};
```

```
D:₩FormalWorks₩Symbolic₩src₩com₩formalworks₩symbolic₩test₩test2
# 37 "/usr/include/assert.h" 2 3 4
# 67 "/usr/include/assert.h" 3 4



# 2 "./test/array/array03.c" 2


int foo(int b[3]) {
 int a[3] = {0, 1, 2};

 b[0] = a[1];
 b[1] = a[0];

 if (a[2] == b[2]){
```

(a) (b)

Figure 3: (a) Target source view screen. (b) Original source view screen.

The source code analysis tool was prepared in Java language, and it is prepared as an Eclipse RCP (Rich Client Platform) application program. Therefore, UI is composed using SWT (graphic library) which is used in the Eclipse. The generated application program is designed so that it is not dependent on the OS, and basically, the result on Microsoft Windows OS is verified. The driving layer of the user interface is as shown in Figure 4. This tool is divided into the SymbolicEngine system to perform the symbolic execution from the sequential program and the SymbolicExecutorRCP system which is the UI system that manages the project by using it and provides interaction to perform with the user. SymbolicExecutionRCP is in charge of interaction with the user and simple external communications, and the SymbolicEngine system is in charge of the important internal communications, such as the ConstraintVerifier, etc.

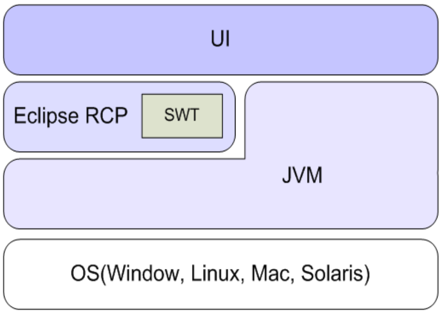


Figure 4: User interface driving layer of the developed tool.

SymbolicEngine performs symbolic execution in the sequential program and given text-based conditional expression, and it provides the value which satisfies the conditional expression. It links to the ConstraintVerifier library in the course of obtaining the value, and finds the value through it. SymbolicEngine is reconstructed into three grouped sub-organizations for this purpose. It can be divided into the parser function which analyzes the input source program, the expression function which parses, and stores it by converting parsed structures to meaningful syntax, and the execution function which performs converted syntax symbolically.

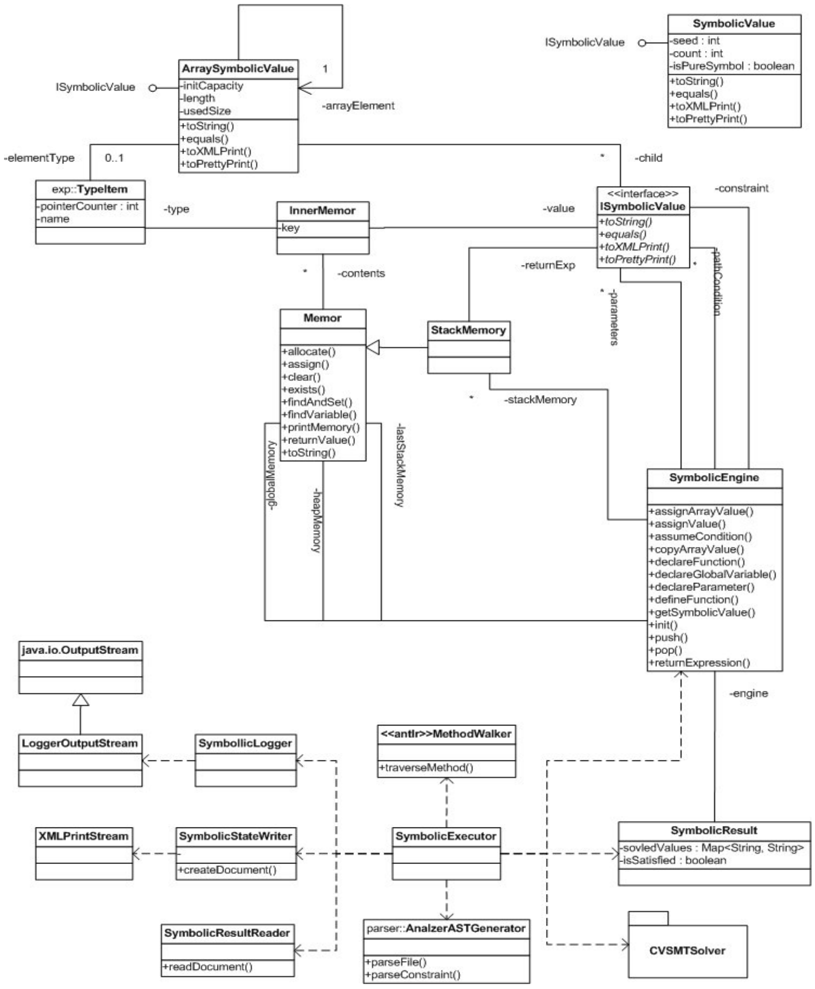
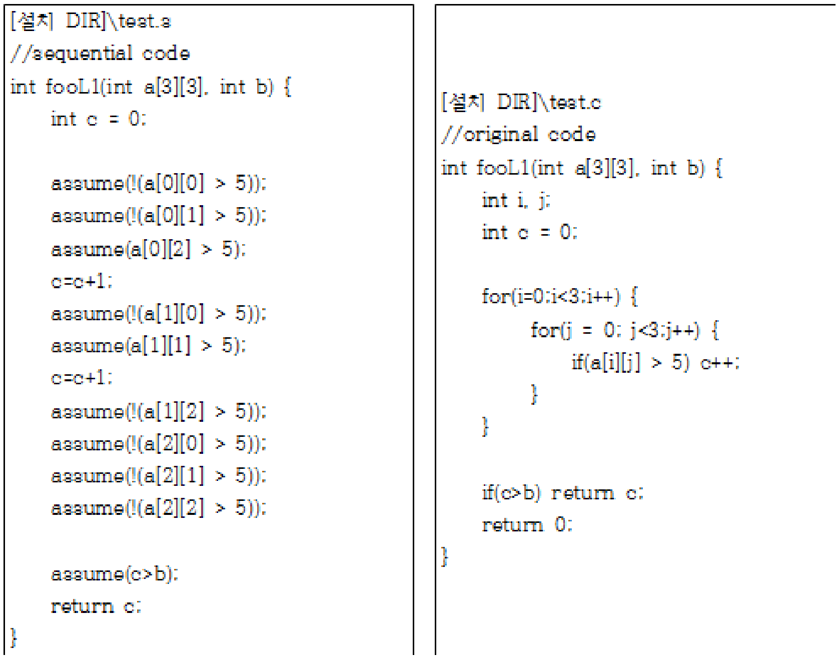


Figure 5: Class diagram in relation to symbolic execution.

Figure 5 is the class diagram in relation to the symbolic execution, and it establish main initialization works and UI-related works through SymbolicExecutor class, and the actual performance occurs through SymbolicEngine class. Memory and SymbolicValue classes for the internal performance are defined, and they can obtain the value if the result of symbolic performance is available through ConstraintSolver which is the external system. ConstraintVerifier is the independent external library, and it is linked to this developed tool through a generated XML document.

The source code analysis tool to which the symbolic execution method was applied is generated as the Eclipse RCP application program which is not dependent on the operation system, and it is linked to the internal SymbolicEngine and external ConstraintVerifier, and it was designed and implemented to perform functions specified previously. By designating one sequential exercise program and target function, it performs validation on the result of development for the corresponding implementation of function through the process that finds and verifies actually a satisfactory value by providing a conditional expression that wants to be verified. The exercise program (see Figure 6) was applied as the input to the developed tool. It verifies if it can perform up to the end of the corresponding function only so that the conditional expression of the exercise program can be "TRUE".



```
[설치] DIR]\test.s
//sequential code
int fooL1(int a[3][3], int b) {
    int c = 0;

    assume(!(a[0][0] > 5));
    assume(!(a[0][1] > 5));
    assume(a[0][2] > 5);
    c=c+1;
    assume(!(a[1][0] > 5));
    assume(a[1][1] > 5);
    c=c+1;
    assume(!(a[1][2] > 5));
    assume(!(a[2][0] > 5));
    assume(!(a[2][1] > 5));
    assume(!(a[2][2] > 5));

    assume(c>b);
    return c;
}
```

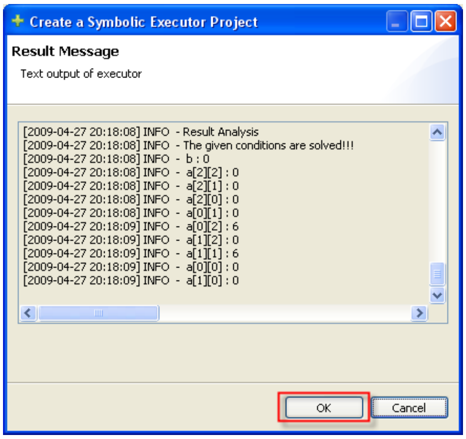
```
[설치] DIR]\test.c
//original code
int fooL1(int a[3][3], int b) {
    int i, j;
    int c = 0;

    for(i=0;i<3;i++) {
        for(j = 0; j<3;j++) {
            if(a[i][j] > 5) c++;
        }
    }

    if(c>b) return c;
    return 0;
}
```

Figure 6: Exercise program applied for validation on the developed tool.

If symbolic execution is carried out in the developed tool, performance-related messages are output onto the screen, and if the result of performance is generated after completion of all of the progresses, a new result of performance will be added to the Project View. The result performed through SymbolicResultEditor is shown in Figure 7 and the value which satisfies the conditional expression for the given sequential program exists, and with the value at that time, 3X3 array and input values of integer variables are

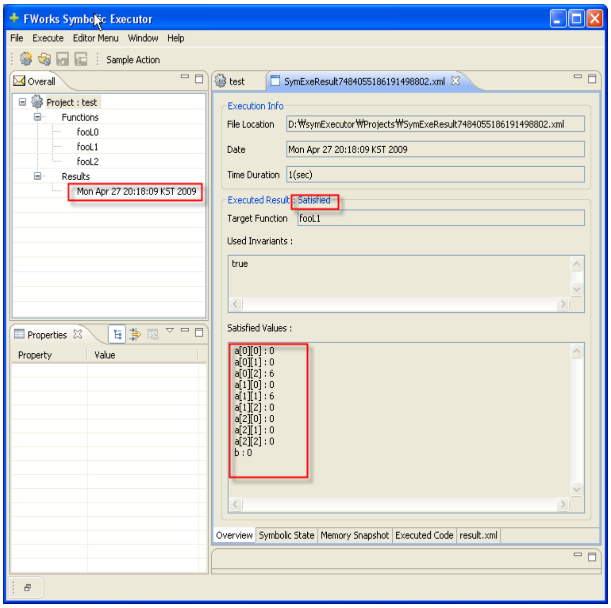


Result Message

Text output of executor

```
[2009-04-27 20:18:08] INFO  - Result Analysis
[2009-04-27 20:18:08] INFO  - The given conditions are solved!!!
[2009-04-27 20:18:08] INFO  -  b : 0
[2009-04-27 20:18:08] INFO  -  a[2][2] : 0
[2009-04-27 20:18:08] INFO  -  a[2][1] : 0
[2009-04-27 20:18:08] INFO  -  a[2][0] : 0
[2009-04-27 20:18:08] INFO  -  a[0][1] : 0
[2009-04-27 20:18:09] INFO  -  a[0][2] : 6
[2009-04-27 20:18:09] INFO  -  a[1][2] : 0
[2009-04-27 20:18:09] INFO  -  a[1][1] : 6
[2009-04-27 20:18:09] INFO  -  a[0][0] : 0
[2009-04-27 20:18:09] INFO  -  a[1][0] : 0
```

Figure 7: Screen for completion of symbolic performance.



Project : test
- Functions
  - fooL0
  - fooL1
  - fooL2
- Results
  - Mon Apr 27 20:18:09 KST 2009

Execution Info

File Location: D:₩symExecutor₩Projects₩SymExeResult7484055186191498802.xml

Date: Mon Apr 27 20:18:09 KST 2009

Time Duration: 1(sec)

Executed Result : Satisfied

Target Function: fooL1

Used Invariants :

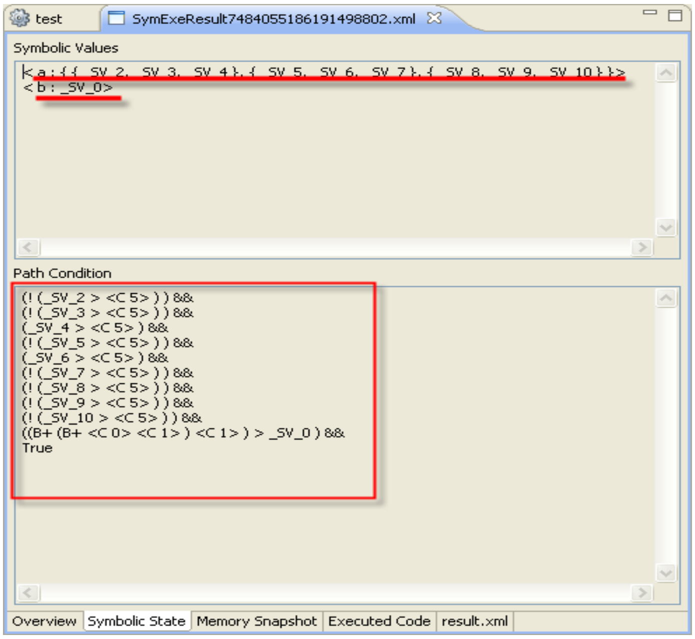
```
true
```

Satisfied Values :

```
a[0][0] : 0
a[0][1] : 0
a[0][2] : 6
a[1][0] : 0
a[1][1] : 6
a[1][2] : 0
a[2][0] : 0
a[2][1] : 0
a[2][2] : 0
b : 0
```

Figure 8: Screen for the result of completed symbolic execution.

provided as Figure 8. The performed symbolic state at this time is (as shown in Figure 9), a variable of the 3X3 integer variable and it shows that each value was changed to the specific symbolic value. In the case of the final route condition, we may see that it consists of complex conditions. Like this, symbolic performance was progressed through a given exercise program, and the referable symbolic value, memory and route conditions could be verified. On this basis, ConstraintVerifier made functions of tool verified finally by finding values suitable for conditional expression and by showing that the value satisfying the route of a given sequential exercise program exists.



```
<a : { { _SV_2, _SV_3, _SV_4 }, { _SV_5, _SV_6, _SV_7 }, { _SV_8, _SV_9, _SV_10 } }>
< b : _SV_0>
```

```
(! (_SV_2 > <C 5> ) ) &&
(! (_SV_3 > <C 5> ) ) &&
(_SV_4 > <C 5> ) &&
(! (_SV_5 > <C 5> ) ) &&
(_SV_6 > <C 5> ) &&
(! (_SV_7 > <C 5> ) ) &&
(! (_SV_8 > <C 5> ) ) &&
(! (_SV_9 > <C 5> ) ) &&
(! (_SV_10 > <C 5> ) ) &&
((B+ (B+ <C 0> <C 1> ) <C 1> ) > _SV_0 ) &&
True
```

Figure 9: Screen for symbolic state.

3 Conclusion

Recently, according to the development of computer technology, the dependence on computer SW of railway systems has been increasing rapidly, and in accordance with this technical development, high reliability and safety is required for the vital railway SW. Accordingly, SW testing and validation are required as mandatory in the railway system SW related international standards, and they require to derive the result of quantitative source code test coverage among SW validation items being required by these international standards. Accordingly, although quantitative validation is performed generally as the control flow analytical method that utilizes SW test cases, it is necessary to reinforce test cases through the practicability of source code route to enhance the test coverage up to about 100%.

Accordingly, this paper proposed a SW source code analysis tool using the symbolic execution method and showed the result of its implementation, and verified functional validation with the result implemented actually. The proposed analysis tool is the one applying a symbolic execution method that can enhance the coverage according to the reinforcement of test cases by inspecting compatibility with SW requirements and practicability of source code route, and detailed contents on design of this developed tool which enable verification up to the sequential program input, calculation of symbolic state, inspection on practicability of program route, and the result of compatibility with requirements were described in the main subject. Likewise, the result of screen implementation for the source code analysis tool to enhance measurement of test coverage for railway SW validation developed in this paper and the result up to the validation on function of tool through the result of symbolic performance completion via actual exercise program were presented also.

Basically, this source code analysis tool to enhance railway SW test coverage is the tool which will be utilized remarkably by sources of demand such as the railway operation agency, etc. for SW validation of railway system, and at the same time, it is considered that the degree of its utilization can be sufficiently high even in the unit or consolidated testing stage for corresponding developed products in the SW development process of railway-related industries also. In addition, finally, since test coverage can be enhanced remarkably so that the result of measurement on code-based test coverage can be matched to the SWSIL grade wanted by the user, it may maximize the efficiency at the actual industrial site of railway fields. If we use the developed source code analysis tool widely in the SW validation and development stage, it is considered that it may contribute to secure the safety and reliability by preventing errors in advance by detecting inherent errors of vital railway SW through it.

References

[1] IEC 61508, “Railway Applications - The specification and demonstration of RAMS”, 1998.

[2] IEC 62279, “Railway Applications – Communication, signalling and processing systems – SW for railway control and protection systems”, 2015.

[3] Railroad Safety Act [Law No. 13436], Partial revision 2015. 07.

[4] J. Burnim and K. Sen, “Heuristics for scalable dynamic test generation”, In ASE’08, 2008. 9.

[5] IEEE Std. 1008-1997, “SW Unit Testing”, 1997.

[6] IEC std. 60880-2006, “SW aspects for computer-based systems performing category A functions”, 2006.

[7] RTCA/DO-178B, “SW considerations in airborne systems and equipment certification”, 1992.

[8] Arnaud Dupuy and Nancy Leveson, “An Empirical Evaluation of the MC/DC Coverage Criterion on the HETE-2 Satellite SW”, Proceedings of DASC (Digital Aviation Systems Conference), Philadelphia, 2000. 10.

[9] Peter G Bishop, “MC/DC based estimation and detection of residual faults in PLC logic networks”, 14th IEEE International Symposium on SW Reliability Engineering(ISSRE), Denver, Colorado, 2003. 11.

[10] S. Bucur, V. Ureche, C. Zamfir, and G. Candea. “Parallel symbolic execution for automated real-world SW testing”, In EuroSys’11, 2011. 4.

[11] Cristian Cadar and Koushik Sen, “Symbolic execution for SW testing: three decades later”, Magazine Communications of the ACM, 2013. 2.