Tunghai Science Vol. 3: 29−44
July, 2001

# Function Validation of a Program Logic Using Software Usage Testing

Wen-Kui Chang[*] Tzu-Po Wang[*]

## Abstract

Function validation of program logic is important and attentive by lots of system managers. This paper investigates issues on evaluating embedded control systems built by the function block diagram (FBD), which is a kind of graphical language used to create program logic that is analogous to circuit diagrams. Our testing FBD environment is embedded on the Tristation 1131 software system with the industry-standard IEC1131-3 conformance. In this research, software usage testing (SUT) is studied to verify if it is possible to be applied in such domain. Essentially, software usage testing first establishes a usage model, and then follows to statistically generate test cases. The usage model is represented as a graph in which the nodes represent usage states and the arcs represent stimuli that cause transitions between usage states. The FBD-primitive application can be transformed into a logical usage model to simulate the real execution scenarios. Based on the usage model, test cases can be randomly generated with the purpose of achieving complete test coverage and we may then quantitatively determine when to stop the testing process for saving the testing effort.

**Keywords:** Function block diagram (FBD), software reliability, software usage testing (SUT), test coverage

## 1. Introduction

In practice, it is expensive both in the model definition and in the model analysis of dependability evaluation on the program logic of the embedded control applications because of the structural complexity and the large system dimension. However, software usage testing is generally acknowledged as a fair and efficient methodology on such applications. Coverage

[*] Department of Computer Sciences & Information Engineering, Tunghai University, Taichung 407, TAIWAN

analysis is deliberately considered with aimed at checking the correctness of a system implementation. We extend the application of software usage testing to such kind of environment systems. The FBD-primitive application can be transformed into a logical usage model to simulate the real execution scenarios. Based on the scenario usage model, test cases can be randomly generated with the purpose of achieving complete test coverage and we may then quantitatively determine when to stop the testing process for saving the testing effort. In this paper, we apply SUT into this kind of environment systems, and then transform the FBD-based program into a usage model in order to simulate the real execution paths.

In this paper, we will introduce the developing platform in the following section and then discuss the SUT methodology in section 3. The detailed software usage testing approach is also provided. Section 4 shows a reactor protection system (RPS) example in the program logic of the embedded control applications domain for demonstration, and provides the rectification result in a quantitative analysis.

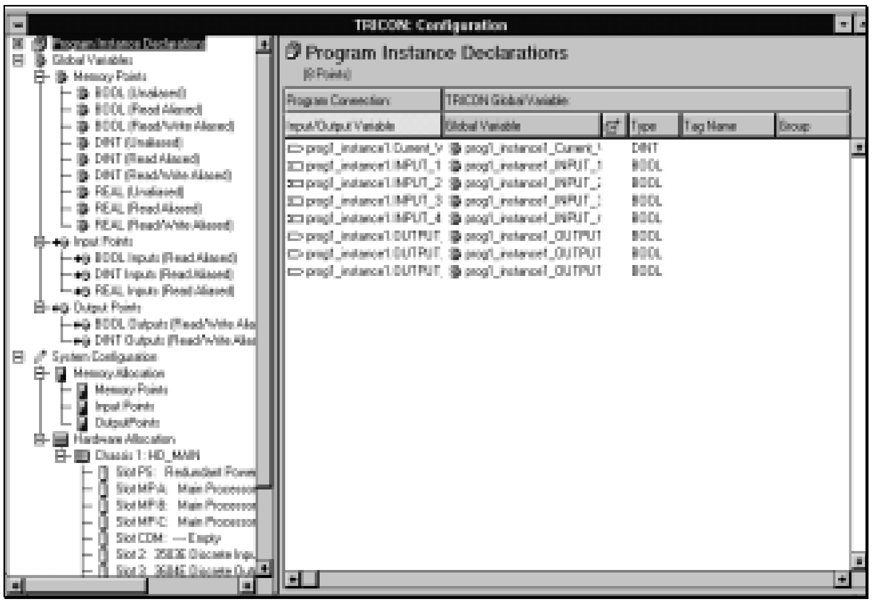
## 2. Materials and Methods

The TriStation 1131 tool kit is a Windows-NT-based programmer workbench for developing, testing and documenting process-control applications that execute in the TRICON controller [10]. All IEC1131-3 compliant languages are included in the workbench to develop, debug, and download programs including function blocks, and functions.

Briefly, the *Configuration Editor* that is activated by selecting "Edit Configuration" on the TRICON Menu, allows setting or displaying the following information for the current project element(s):

- Program instance declarations
- Global variable definitions
- System parameters
- Memory variables
- Hardware allocation

The configuration variables for the current project are displayed in the left half of the *Configuration Editor* screen as the *Configuration Tree* as shown in Fig. 1. The values or parameters for any of the variables or points can be set or edited on the right side of the screen when a variable is highlighted in the *Configuration Tree*.



TRICON: Configuration

- Program Instance Declarations
- Global Variables
  - Memory Points
    - BOOL (Unaliased)
    - BOOL (Read Aliased)
    - BOOL (Read/Write Aliased)
    - DINT (Unaliased)
    - DINT (Read Aliased)
    - DINT (Read/Write Aliased)
    - REAL (Unaliased)
    - REAL (Read Aliased)
    - REAL (Read/Write Aliased)
  - Input Points
    - BOOL Inputs (Read Aliased)
    - DINT Inputs (Read Aliased)
    - REAL Inputs (Read Aliased)
  - Output Points
    - BOOL Outputs (Read/Write Ali
    - DINT Outputs (Read/Write Alia
- System Configuration
  - Memory Allocation
    - Memory Points
    - Input Points
    - OutputPoints
  - Hardware Allocation
    - Chassis 1: HD_MAIN
      - Slot PS: Redundant Powe
      - Slot MP-A: Main Processo
      - Slot MP-B: Main Processo
      - Slot MP-C: Main Processo
      - Slot COM: --- Empty
      - Slot 2: 3503E Discrete Inpu
      - Slot 3: 3604E Discrete Ou

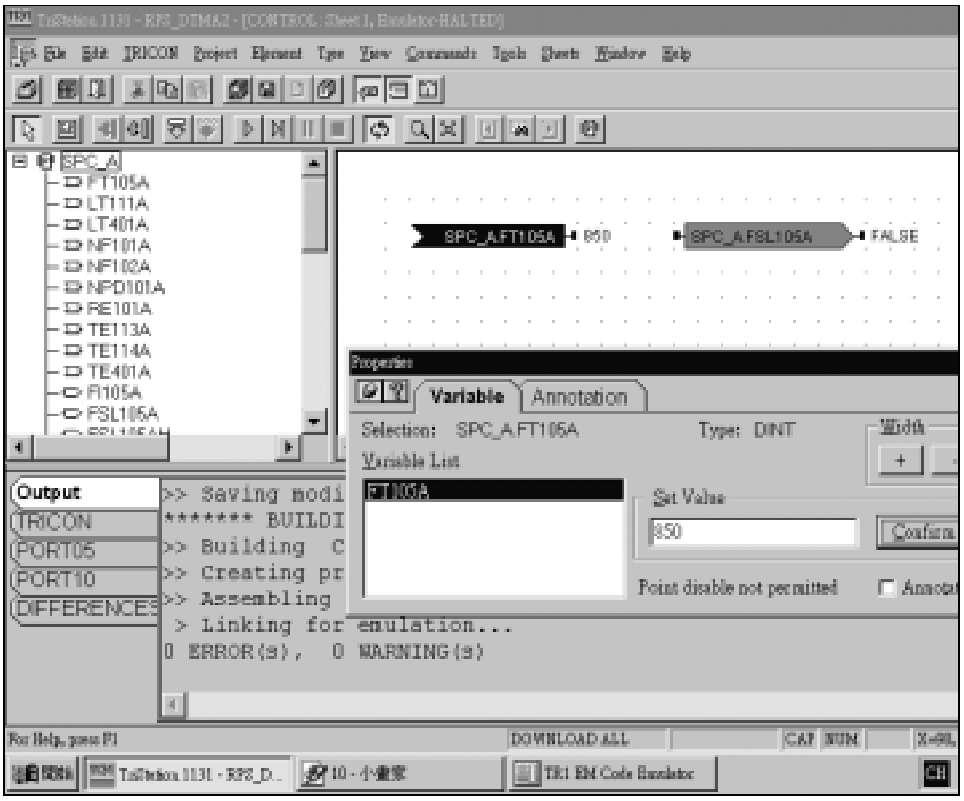
Program Instance Declarations
(8 Points)

| Program Connection | TRICON Global Variable | | | | |
|---|---|---|---|---|---|
| Input/Output Variable | Global Variable | | Type | Tag Name | Group |
| prog1_instance1.Current_V | prog1_instance1_Current_ | | DINT | | |
| prog1_instance1.INPUT_1 | prog1_instance1_INPUT_ | | BOOL | | |
| prog1_instance1.INPUT_2 | prog1_instance1_INPUT_ | | BOOL | | |
| prog1_instance1.INPUT_3 | prog1_instance1_INPUT_ | | BOOL | | |
| prog1_instance1.INPUT_4 | prog1_instance1_INPUT_ | | BOOL | | |
| prog1_instance1.OUTPUT_ | prog1_instance1_OUTPUT | | BOOL | | |
| prog1_instance1.OUTPUT_ | prog1_instance1_OUTPUT | | BOOL | | |
| prog1_instance1.OUTPUT_ | prog1_instance1_OUTPUT | | BOOL | | |

Figure 1. The configuration editor

Applications may start to run the current configuration from the TRICON Emulator. The configuration will run in the Emulator until it completes its execution or a Halt or Pause command is given. Clicking on the *Continuous Refresh* button will allow the display to change as values change in the variables executing.

After stopping execution and disconnecting from the Emulator, we can edit project elements or change the current configuration and reconnect to the Emulator to observe how it would run on the TRICON controller. Fig. 2 illustrates emulator control panel with the input value and executed result.



Selection: SPC_A.FT105A  Type: DINT

Set Value: 850

```
>> Saving modi
******* BUILDI
>> Building C
>> Creating pr
>> Assembling
 > Linking for emulation...
0 ERROR(s),  0 WARNING(s)
```

Figure 2. The emulator control panel

## 3. Evaluation technique

In this section, we provide the SUT approach for evaluation as following:

### 3.1 The rationale

Software usage testing is a stochastic testing mechanism. It begins at the software specification level, and transforms it into a software usage model. After the model analyzing and evaluation, the testers may test plan accordingly. It is an objective and acceptable software testing mechanism.

Software usage model represents infinite usage behavior with finite states. It could be created before the code process finishes. With the modeling process, it will promote the software improvement, and increase the testability.

### 3.1.1 Model analysis and usage modeling

The initial structure of a usage model follows directly from the software specification [6]. In particular [9,12,13], the canonical sequences identified during specification define the initial state space for the usage model.

A usage model may be represented as a graph in which the nodes represent the usage states and the arcs represent the stimuli that cause transitions between usage states. Developers and potential users, who often participate in usage model review, can understand graphical usage models easily. Graphical representation aids in system understanding but, generally, is only used for small systems or for high-level representation of large systems. Usage models for large systems are often defined abstractly at first, with automated support for model expansion through sub-models and transformation of abstract stimuli to associated atomic stimuli.

### 3.1.2 Test scripts generation and certification

After the usage model has been developed, test cases can be generated automatically by traversing the usage states of the model [9,12,15], guided by the transition probabilities associated with the exit arcs from each state. Because each arc is associated with a particular stimulus to the system, the traversal results in an accumulation of successive stimuli that represents a particular test case. The test cases constitute a script for use in testing. They may be annotated during test planning to include instructions for conducting and evaluating tests, and they may be annotated during testing to record results and observations.

## 3.2 Evaluation steps

In summary, a user-oriented testing framework for certifying FBD-primitive applications can be performed as following: building usage model with their transition distribution, generating test scenarios statistically, executing each test scenarios, collecting failure information, and analyzing certification result.

### 3.2.1 Building usage model with their transition distribution

Essentially a software usage model characterizes various operational uses of a software system [9]. Suppose that an operational use is a skeleton for the intended use of the software in an intended environment. Thus, all possible operational uses of a software system will constitute a population with a huge size. If a usage sample of test scripts is drawn statistically from the usage population, performance on this sample may then be used as a basis for the evaluation of software quality.

### 3.2.2 Generating test scenarios statistically

After the usage model and the analysis have been reviewed and determined to be a valuable basis for testing, test cases are generated according to statistical usage theorem [4]. The first test suite generated is usually the minimal arc coverage suite, which traverses the model in the fewest possible steps required to achieve model coverage. Model coverage testing accomplishes several goals. The model is further confirmed to be precise, the ability to evaluate all responses is confirmed, and the readiness of the software for random testing is established. Random testing enables measurement of the reliability of the software.

### 3.2.3 Executing each test scenarios

In this phase, a tester executes the test scripts manually. The first step, as above test script form, tester follow sequential steps of test script and execute those links.

There is an issue that we can do in our future work. When there are too many nodes of usage model, execution of test script will take several hours. To resolve this issue, we can design a tool that can execute these test scripts automatically. Using a tool to execute test scripts will save a lot of time and we can add more nodes in the usage models.

### 3.2.4 Collecting failure information and analyzing certification result

Once test scripts are generated statistically and executed, some failures may be found and collected to delineate their resultant responses conflicting with the user expectations. By the analysis result derived from the usage model, partial certification is summarized.

Moreover, based on the collected failure information, the associate defects can be identified and recovered by tracking the failed test script back to the usage specification.

## 3.3 Certification assisting tool - toolSET_Certify

toolSET_Certify is a GUI-based UNIX application developed by Q-Lab[7]. It transforms software systems to usage model mathematically and provides the automated analysis and effective to the system operation in the future. It is designed to support testing according to the above scientific protocol. The tool supports the three phases of software usage testing: model development, model analysis and test planning, and certification testing. The testers build models that are usage models of the system capability; environments of use; and customer classes. The tool provides automated test case generation and continuous test effectiveness monitoring providing objective criteria for increment testing and ultimate product certification.

# 4. Demonstration results

A demonstration example will be discussed in the following:

## 4.1 Problem description

### 4.1.1 Target system description

The target system in this research is a reactor protection system (RPS), which is a kind of embedded control programs. The RPS includes 9 function categories, which are independent with each other. The first category is analogy signal transformation function and illustrated as in Fig. 3.

### 4.1.2 Analogy signal transformation function

In the analogy signal transformation function, the *AIN* program component will transfer the signal of the global variable FT105A into analog input function between the value of 819.0~4095.0, and then subtract 819.0 by the *SUB* component. The result is further multiplied by 3276.0 by the *MUL* component and finally we take its square root by the *SQRT* component. Furthermore, the previous result is multiplied by 0.7326 and adds the quantity 100.0 by the *ADD* component. If the final value is great than 935 using the *GE* component, the output value will be “true” and saved in the global variable FSL105A and FSL105AH. On the other hand, if the final value is less than 935, the output value will be “false”. Each input that gets from the sensor will be processed in each category, and the result will be computed finally. In addition, all processes follow the IEC 1131-3 standard and built by FBD.

Table 1 lists the usage specification of all program components that are used in RPS. For instance, the *ABS* component receives the integer input and then computes its absolute value, which may be any number. The stimulus and responses in RPS is also shown in Table 1.

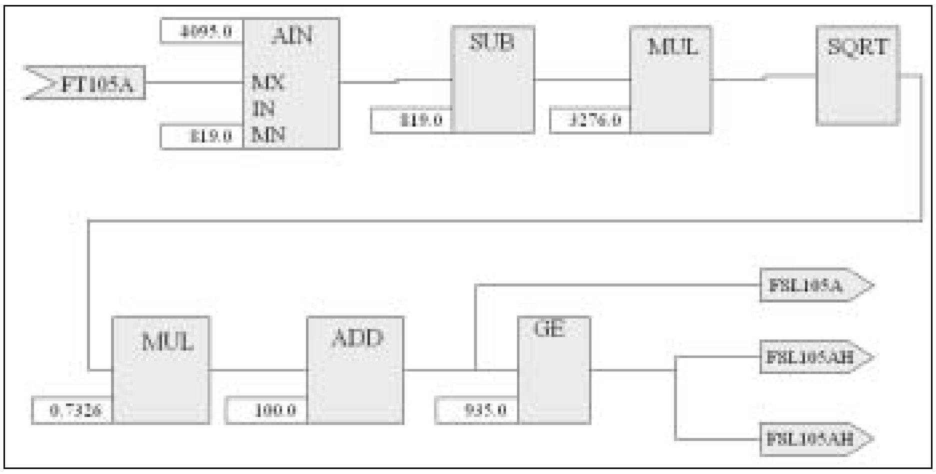


Figure 3. Analogy signal transformation function

Table. 1. Usage specification of the 1131-3 program components

| Program Component | Meaning | Stimulus | Responses |
|---|---|---|---|
| *ABS* | Absolute Value | INT | *ANY_NUM* |
| *AIN* | Analog Input Function | REAL | *REAL* |
| | | DINT | |
| | | REAL | |
| *ADD* | Add | ANY_NUM | *ANY_NUM* |
| | | ANY_NUM | |
| | | ANY_NUM | |
| *GE* | Greater Than or Equal | ANY_NOTE | *BOOL* |
| | | ANY_NOTE | |
| *EXPT* | Exponentiation | ANY_REAL | *ANY_REAL* |
| | | ANY_NUM | |
| *LE* | Less Than or Equal | ANY_NOTE | *BOOL* |
| | | ANY_NOTE | |
| *MUL* | Multiply | ANY_NUM | *ANY_NUM* |
| | | ANY_NUM | |
| | | ANY_NUM | |
| *SQRT* | Square Root | ANY_REAL | *ANY_REAL* |
| *SUB* | Subtract | ANY_NUM | *ANY_NUM* |
| | | ANY_NUM | |

**4.2 Usage model**

As stated in the previous section, the analogy signal transformation function of RPS example can be directly transferred into the SUT usage model as shown in Fig 4. All the usage models of other categories could be also generated at the same time.

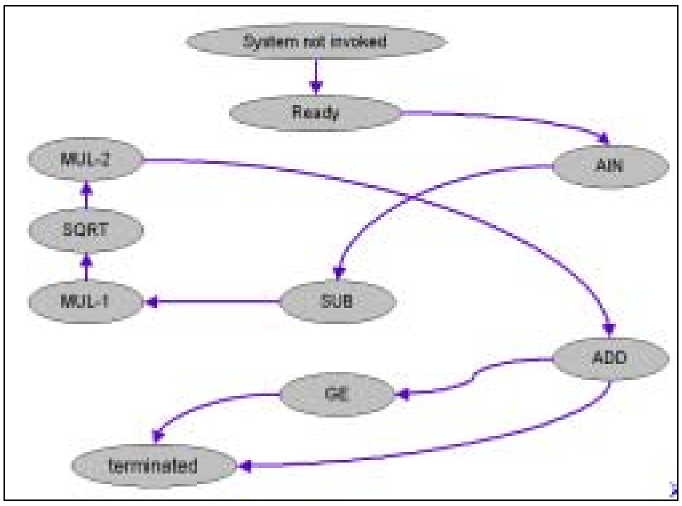


Figure 4. The usage model of analogy signal transformation function

The model is then analyzed by the toolSET_Certify [7], and the following table shows the result. The analogy signal transformation function contains 10 states and 10 active arcs. By the suggestion of toolSET_Certify, two test scripts are suggested to transverse the completed usage model. All the number of test scripts in other categories would be suggested at the same time.

Table. 2 Usage model analysis table

| | |
|---|---|
| Number of no deleted states | 10 |
| Number of active arcs | 10 |
| Expected script length | 9.5 |
| Min test scripts for least likely state coverage | 2.0 |
| Min test scripts for least likely transition coverage | 2.0 |

## 4.3 Test scripts

The test scripts may be generated from the usage model, and from the above table, it suggests that we have to execute two test scripts for the analogy signal transformation function, as illustrated in Table 3. In RPS, 20 test scripts are needed to achieve complete coverage.

Table 3. Paths of two test scripts

*Script 1:*

$$\text{Software Not Invoked} \xrightarrow[\text{to ready}]{} \text{ready} \xrightarrow[\text{to AIN}]{} \text{AIN} \xrightarrow[\text{to SUB}]{} \text{SUB} \xrightarrow[\text{to MUL -1}]{} \text{MUL - 1}$$

$$\xrightarrow[\text{to SQRT}]{} \text{SQRT} \xrightarrow[\text{to MUL -2}]{} \text{MUL - 2} \xrightarrow[\text{to ADD}]{} \text{ADD} \xrightarrow[\text{to GE}]{} \text{GE} \xrightarrow[\text{EXIT}]{} \text{Software Terminated}$$

*Script 2:*

$$\text{Software Not Invoked} \xrightarrow[\text{to ready}]{} \text{ready} \xrightarrow[\text{to AIN}]{} \text{AIN} \xrightarrow[\text{to SUB}]{} \text{SUB} \xrightarrow[\text{to MUL -1}]{} \text{MUL - 1}$$

$$\xrightarrow[\text{to SQRT}]{} \text{SQRT} \xrightarrow[\text{to MUL -2}]{} \text{MUL - 2} \xrightarrow[\text{to ADD}]{} \text{ADD} \xrightarrow[\text{EXIT}]{} \text{Software Terminated}$$

## 4.4 Expected output

The following table illustrates the expected output of analogy signal transformation function. It provides the information for testers to review the test result. In this table, the sequence shows the order of execution and the input is the stimulus of the sequence. Current state and reached state illustrates the transition path. If the result follows the description in this table, the test script will be labeled as "PASS". Otherwise, "FAIL" is labeled.

Table. 4. Expected output

| **Sequence** | **Input** | **Current state** | **Expected output** | **Reached state** |
|---|---|---|---|---|
| 1 | FT105A | *AIN* | 819<FT105A<4096 | *SUB* |
| 2 | 819<FT105A<4096 | *SUB* | FT105A-819 | *MUL1* |
| 3 | FT105A-819 | *MUL1* | $\sqrt{(\text{FT105A} - 819) \text{ X } 3276}$ | *SQRT* |
| 4 | $\sqrt{(\text{FT105A} - 819) \text{ X } 3276}$ | *SQRT* | $\sqrt{(\text{FT105A} - 819) \text{ X } 3276} \times 0.7326$ | *MUL2* |
| 5 | $\sqrt{(\text{FT105A} - 819) \text{ X } 3276} \times 0.7326$ | *MUL2* | $(\sqrt{(\text{FT105A} - 819) \text{ X } 3276} \times 0.7326)-100$ | *ADD* |
| 6 | $(\sqrt{(\text{FT105A} - 819) \text{ X } 3276} \times 0.7326)-100$ | *GE* | If $[(\sqrt{(\text{FT105A} - 819) \text{ X } 3276} \times 0.7326)-100] >= 906$, then 1 | |
| | | | If $[(\sqrt{(\text{FT105A} - 819) \text{ X } 3276} \times 0.7326)-100] < 906$, then 0 | |

## 4.5 Evaluation report

In this practical demonstration project, one functional error was found and located. The error is due to the *MUL* component. It failed because the programmer disabled its function. Such kind of failure will cause fatal damage. Sometimes it has to trace back from user requirements.

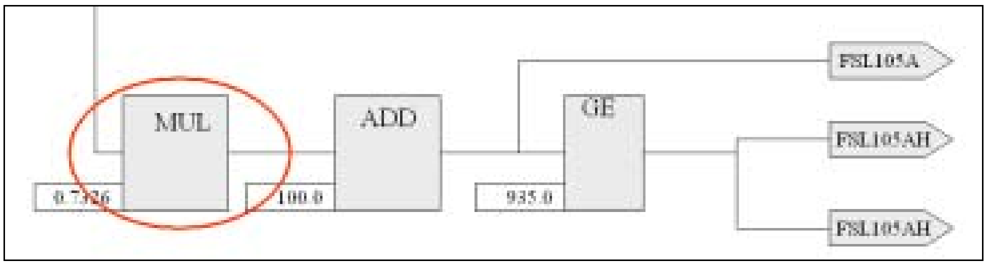


Figure 5. The error component

In summary, the certification result is given Table 5. Fig. 6 illustrates the distributions for reliability evaluation and arc coverage. It is noted that the reliability dropped for the 7th and 10th test scripts due to failures.

Table 5. Summary of certification analysis

| **Script #** | **Result** | **MTTF** | **Reliability** | **% States certified** | **% Arcs certified** |
|---|---|---|---|---|---|
| 1 | PASS | NA | 1.000000 | 15.625000 | 9.756097 |
| 2 | PASS | NA | 1.000000 | 37.500000 | 29.268291 |
| … | … | … | … | … | … |
| 7 | FAIL | 7.000000 | 0.857143 | 65.625000 | 58.536583 |
| … | … | … | … | … | … |
| 10 | FAIL | 5.000000 | 0.800000 | 71.875000 | 65.853661 |
| … | … | … | … | … | … |
| 19 | PASS | 9.499999 | 0.894737 | 93.750000 | 92.682930 |
| 20 | PASS | 9.999998 | 0.900000 | 93.750000 | 92.682930 |

Table6. Failure information

| Script # | Result | Failure ID | Transition # | Status |
|---|---|---|---|---|
| 7 | Fail | 1 | 6 | Halt |
| 10 | Fail | 2 | 6 | Halt |

In this experiment, two failures happened at the 7th and 10th test scripts are found. Functionally, the *MUL* component of the analogy signal transformation function in RPS normally receives one input and then multiples with 0.7326. However, the *MUL* input was disabled. Analytically, the programmer caused such failure as he set the FBD diagram.

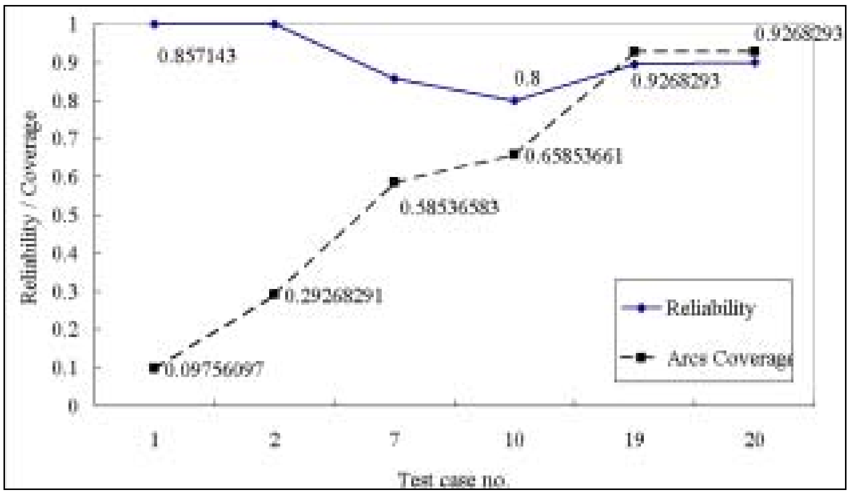


Figure 6. Reliability evaluation and arc coverage distributions

### 4.6 Rectification result

After rectifying the failures for the analogy signal transformation function, the *MUL* function is operated again for normal execution. Afterwards, all test scripts need to be performed again for regressing the repair process. Fig. 7 similarly illustrates the reliability evaluation and arc coverage information.

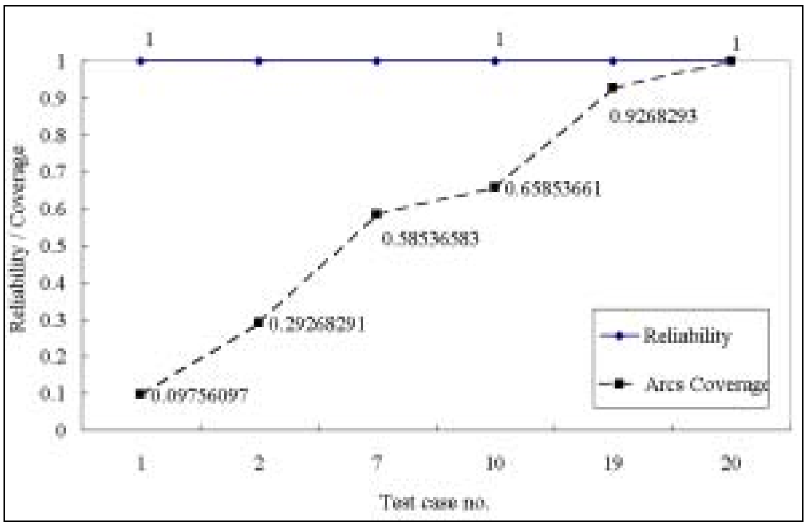


Figure 7. Rectification result

### 4.7 Benefits of the proposed approach

Software usage testing of a software system produces measurements of product and process quality for management decision making throughout the life cycle. Because a usage model is based on specifications rather than codes, the insights that result from model building can be used to make informed management decisions in the early stages of a project. It prevents problems getting larger and larger. The following are key benefits.

#### 4.7.1 User requirements validation

SUT begins at the software specification level, and transforms it into a software usage model. A usage model is an external view of the system specification. It is also comprehensible by system engineers, developers, customers, and end users. Interfaces and requirements are often simplified or clarified when the usage model is reviewed systematically in the context of operational use.

#### 4.7.2 Effective and efficient testing

Based on the statistical certification process, the fault in the target system could be found in the early stage. However, faults are not equally likely to cause failures. Faults that are on frequently traversed paths have a higher probability of causing failures than faults that are on infrequently traversed paths. This simple fact is the primary motivation for random testing: Faults are discovered in roughly the order in which they would cause failures in the field. The test budget is spent in a way that maximizes the increase in operational reliability resulting from testing.

#### 4.7.3 Quantitative test management

Software usage testing based on a usage model provides quantitative criteria for management decisions about completion of testing and system release. The sufficiency of testing can be measured as the statistical difference between expected usage and tested usage. By such information, the tester could decide when to stop testing.

# 5. Conclusions

Essentially, the rationale of software usage testing is to generate a set of complete test cases systematically, rather than the general ad hoc approaches. It may provide both complete testing coverage and quantitative analysis. This project is to investigate the feasibility of employing the software usage testing to software testing in the program logic of the embedded control applications domains. A testing framework is proposed to perform specification testing in this research. In addition, this research has demonstrated that, by a RPS example, the suggested mechanism is not only feasible but also efficient in locating and recovering potential defects existed in the FBD diagram in the program logic of the embedded control applications.

The most prominent advantage of the proposed framework is that testers can start their testing process without perceiving the complex internal system architecture. Accordingly, it may reduce the time effort and cost expended in the testing period.

Obviously, the program logic of the future embedded control applications will become more complicated continuously. Naturally, it will increase complexity of testing. In the future work, we hope to develop a new approach that can overcome these problems. On other hands, too many states in a usage model generate numerous test scripts; automatic execution of the designed test scenarios will be required for saving testing effort.

## Acknowledgements

This research is supported by the Institute of Nuclear Energy Research. (Grant 90-2001-INER-011: software reliability evaluation methodology development and application.) Further, the authors of this paper would also like to thank Mr. Tsung-Chieh Cheng for providing the RPS example at Section 4 in this project.

## References

[1] Chang, W.K., Wang, T.P., and Fu, C C. (1999) "A Study on Usage Modeling for Software Reliability Certification via Prototyping Simulation," *The 3rd Reliability & Maintainability Conference*, pp. 279-285.

[2] Fekete, A., Gupta, D., Luchangco, V., and Lynch, N. (1999) "Eventually-Serializable Data Services," *Theoretical Computer Science* **220**, pp. 113-156.

[3] Kim, M., Shin, J., Chanson, S.T., and Kang, S. (1999) "An Approach for Testing

Asynchronous Communication System," *IEICE Transition Communication* **E82-B**(1), pp. 689-701.

[4] Kone´, O., and Castanet, R. (2000) "Test Generation for Interworking Systems," *Computer Communications* **23**, pp. 642-652.

[5] Lin, Xuemin (1997) "A fully distributed quorum consensus method with high fault-tolerance and low communication overhead," *Theoretical Computer Science* **185**(2), pp.259-275.

[6] Prowell, S.J., Trammell, C.J., Liger, R.C., and Poore, J.H. (1999) *Cleanroom Software Engineering Technology and Process*, Addison Wesley, pp.46-109.

[7] Q-Labs: "toolSET_Certify User Guide." Version 4.0, 1999.

[8] Silberschatz, A., Galvin, P., and Gagne, G. (2000) *Applied Operating System Concepts*, John Wiley & Sons, Inc., pp. 501-563.

[9] Trammell, C.J. and Poore, J.H. (1995) "Process Control in Statistical Reliability Certification," *Proceedings of the Seventh Annual Software Software Technology Conference*, pp. 528-536.

[10] TriStation 1131$^{TM}$ Help System, Triconex Corporation, 1998.

[11] Özgür, Ulusoy (1995) "Research Issues in Real-Time Database Systems," *Information Sciences* **87**(1-3), pp. 123-151.

[12] Walton, G.H., Poore, J.H., and Trammell, C.J. (1995) "Statistical Testing of Software Based on a Usage Model," *Software Practice and Experience* **25**(1), pp. 97-108.

# 利用軟體使用測試方法驗證程式邏輯之功能正確性

張文貴[*]　　王咨博[*]

## 摘　　要

程式邏輯之功能驗證是非常重要且受許多系統管理者關心的議題，本文重點即在評估以功能方塊圖(FBD)描繪而成之嵌入式控制系統。功能方塊圖是一種圖形化語言，它可用來建構類比訊號的系統邏輯圖，我們的測試環境則是架構在符合 IEC1131-3 標準之 Tristation 1131 軟體系統中。

在本質上，軟體使用測試是先建立軟體使用模式並在統計學基礎下產生測試案例，在使用模式中，以節點來代表使用狀態並以連線代表會觸動使用狀態間轉換的激發動作。以 FBD 建構而成的系統可以轉換為一個邏輯化的使用模式並據以產生實際的測試案例，在此機制下，隨機產生的測試案例將可達到完整的測試涵蓋率，並決定何時停止測試過程以節省測試的成本。

關鍵詞：功能方塊圖、軟體可靠度、軟體使用測試、測試涵蓋。

[*] 東海大學資訊工程與科學系