The 6th International Conference on Nuclear Thermal Hydraulics, Operations and Safety (NUTHOS-6)
Nara, Japan, October 4-8, 2004.
Paper ID. N6P298

Direct Control Flow Testing on Function Block Diagrams

Junbeom Yoo, Suhyun Park, Hojung Bang, Taihyo Kim, Sungdeok Cha
Korea Advanced Institute of Science and Technology(KAIST) and AITrc/SPIC/IIRTRC,
Dept. of Electrical Engineering and Computer Science,
373-1, Guseong-dong, Yuseong-gu, Daejeon, Republic of Korea
{jbyoo, suhyun, hjbang, taihyo, cha}@salmosa.kaist.ac.kr

KEY WORDS

Software Testing, Control Flow Testing, Function Block Diagram, Programmable Logic Controller

ABSTRACT

In this paper, we propose a testing technique that can directly test FBD programs without generating intermediate code for testing purpose. The previous PLC-based software testing generates an intermediate code such as C, which is equivalent to the original FBD, and targets an intermediate code. In order to apply unit and integration testing techniques to FBDs, we transform FBD program into a control flow graph and apply existing control flow testing coverage criteria to the graph. With our approach, PLC based software designed in FBD language can be tested cost-efficiently because we do not need to generate intermediate code. To demonstrate the usefulness of the proposed method, we use a trip logic of BP(Bistable Process) in DPPS(Digital Plant Protection System) RPS(Reactor Protection System), which is currently being developed at KNICS (KNICS, -) in Korea.

1. INTRODUCTION

Testing is an indispensable process in software development for assuring quality of software. In the area of nuclear power plant control systems, testing on software becomes more important as existing analog systems based on RLL (Relay Ladder Logic) are replaced by digital systems controlled by software (US NRC, 1997). The software is implemented on PLC (Programmable Logic Controller) (Mader, 2000), which is widely used industrial computer to implement real-time safety critical software. The software is designed using PLC programming languages such as LD (Ladder Diagram) or FBD (Function Block Diagram) (IEC, 1993).

PLC programs written in such languages are compiled into PLC machine code automatically by engineering tools provided by PLC vendors. During the multi-compilations, C or other programming code are generated as an intermediate output of the process. Previous works on PLC based software testing usually applies software testing techniques to the intermediate code. It is due to the fact that the testing on PLC machine code is so complicated that we can hardly perform it.

In this paper, we propose a testing technique that can directly test the FBD programs without generating the intermediate code. We assume that translation process from FBD programs into PLC machine codes, which is conducted by engineering tools automatically, produces no errors. This assumption is reasonable in that the translation processes have been validated for a few decades by many developers and end-users. At first, we define the software testing in terms of FBD programs, which are composed of networks of many sub-function blocks. We need to define the concept of unit and module in terms of networks of function blocks. We then transform the FBD program into a kind of control flow graph, and apply the existing control flow testing coverage criteria to the graph. With our approach, the testing of PLC based software written in FBDs can be processed cost-efficiently because we do not need to generate intermediate code. To demonstrate the effectiveness of the proposed method, we introduce an example of trip logic of BP (Bistable Process) in DPPS (Digital Plant Protection System) RPS (Reactor Protection System), which is presently being developed at KNICS in Korea.

The remainder of the paper is organized as follows: Section 2 briefly introduces the FBD and software testing. In Section 3, we define the unit and module of FBD programs for unit and integration testing. Section 4 describes the unit testing process on FBD programs and introduces a real case study, which is currently being developed in Korea. Conclusion and future work are described in Section 5.

2. BACKGROUND

The target of testing in this paper is neither PLC machine code nor intermediate code translated from FBD. PLC machine code is too complex to test itself. We apply software testing techniques directly on FBD instead of generating intermediate code from FBD. The behaviors of FBD is similar to procedure or function of software in that FBD gets inputs and emits outputs according to input values. Although this similarity makes software testing techniques useful in FBD, there is no systematic way to apply software testing technique to FBD itself. This section addresses basic issues in FBD and software testing.

2.1 Function Block Diagram

Programmable Logic Controllers (PLC) (Mader, 2004) are widely used in diverse control systems in chemical processing plants, nuclear power plants or traffic control systems. A PLC, an industrial computer specialized for real-time applications, is an integrated system containing a processor, main memory, input modules and output modules that are coupled together by a common bus.

There are several PLC programming languages. The IEC 61131-3 (IEC, 1993) standards include five: Structured Text (ST), Function Block Diagram (FBD), Ladder Diagram (LD), Instruction List (IL) and Sequential Function Chart (SFC). The FBD is one of the most widely used languages because of its graphical notations and usefulness in applications with a high degree of data flow between control components. Such a data or information flow between control components can be designed as a network of software blocks.

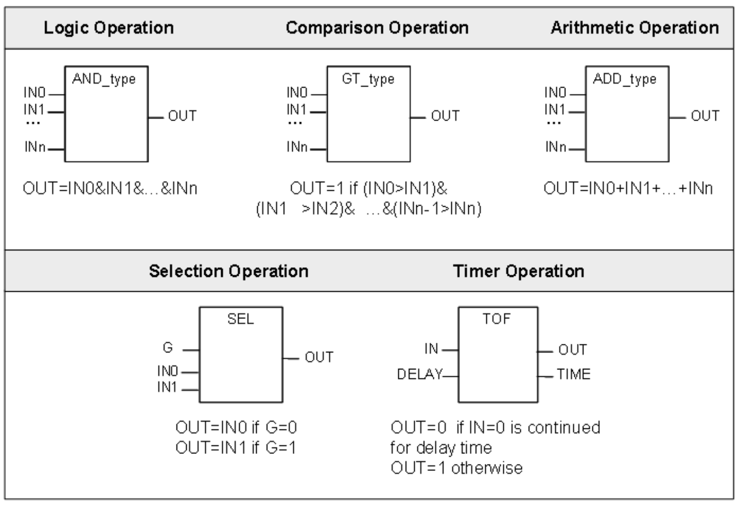


Figure 1. Categorized example of FBD function blocks

FBD design expresses system behavior in terms of flow of signals among function blocks. Functions between input and output variables are graphically represented by a collection of function blocks "wired" together in a manner of a circuit diagram. A function block is depicted as a rectangle and is connected to input/output variables. Function blocks are classified into several categories according to the operations they perform. Figure 1 shows some of the groups of function blocks and example blocks in each group. The RPS being developed at KNICS (KNICS -) is programmed using only the 5 categories shown in Figure 1. Using only these groups increases readability and understandability, and consequently enhances the software safety in nuclear domains.

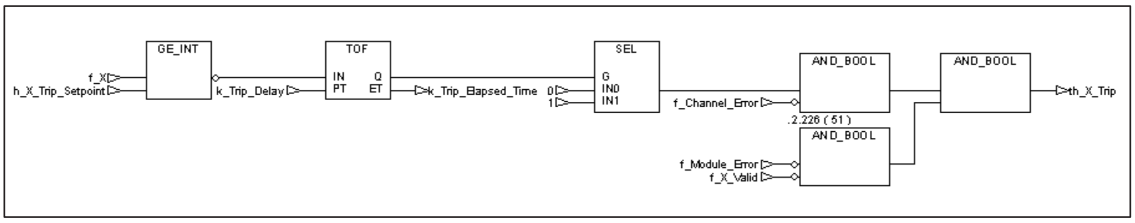


Figure 2. FBD example

Figure 2 shows a network of function blocks. The output *th_X_Trip* is produced by the combination of the function block operations. First, the GE_INT function compares inputs *f_X* and *h_X_Trip_Setpoint*. The result is inverted and given as an input to the TOF function. Next, the TOF outputs a result according to the input from the preceding GE_INT block and the delay time *k_Trip_Delay* based on its function in Figure 1. The output is given to the following SEL function block. The SEL outputs either 0 or 1 based on the output from the TOF function. Finally, the result from the SEL function and the inverted values of *f_Channel_Error*, *f_Module_Error* and *f_X_Valid* are logically AND-ed. The AND-ed result is stored to the final output variable *th_X_Trip*.

2.2 Software testing

The objective of software testing is to make a judgment about quality of software and to discover problems (Jorgensen, 1995). In order to test the software, we need test input and expected output, which consist of test cases. After executing the software with the test input, tester decides whether actual output is same as the expected one. How to choose test cases can highly affect the result of testing.

There are two fundamental approaches to identifying test cases. One is functional testing, which is based on the view that any programs can be considered to be a function that maps values from its input domain to values in its output domain. The other is structural testing, which allows testers to identify test cases based on how the function is actually implemented. In short, structural testing identifies a set of test cases based on actual code of software while functional testing identifies them based on the specification of the software.

In this paper, we adopt structural testing approach. The kinds of structural testing approach are twofold. One is control flow testing, which focuses on control flow of the software. The other is data flow testing, which focuses on where the variable is defined and used in the software. The proposed approach uses control flow testing technique, which provides a variety of test coverage criteria. A set of test cases should exercise all nodes, all edges or all paths of control flow graph.

3. GRANULARITY OF FBD TESTING

FBD is a network of function blocks, so it is important to define the concept of units and modules clearly. If we define the unit as one function block, we do not need unit testing since we can assume that each function block always operate correctly. On the other hand, if we define the unit by the number of function blocks for convenience, the interaction of variables in a unit may bring about the issues of integration testing.

We define the unit of FBD program as a meaningful function block, which computes a primary output. Primary output is stored in the memory of PLC for external output or internal uses of other units. If the output variable is used just for programming convenience, we do not consider it as a unit. For example, Figure 3 shows a part of KNICS RPS trip logic. It is pre-trip set-point calculation part for manual reset variable set-point falling trip logic. Although the output *Pk_PTSP_Satat0* of upper block seems to be primary output, it is just used for programming convenience. *Pk_PTSP_Satat0* is stored in the memory and then internally used as the second input for *MUX* function block in lower block. Therefore, the upper block is not defined as an individual unit. Both upper and lower blocks are defined as an individual unit in that they perform a function and compute an external outputs *f_X_PTSP*. In summary, the upper block is a part of the lower one, and the whole block, which outputs *f_X_PTSP*, is one unit.

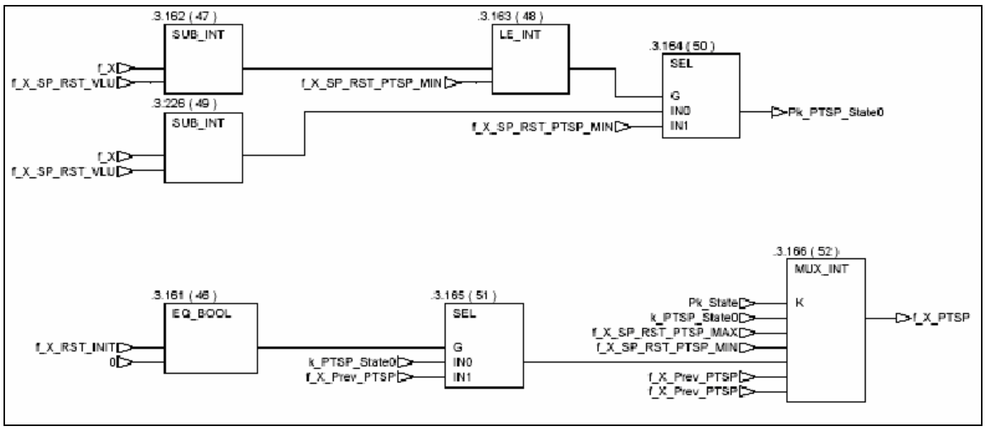
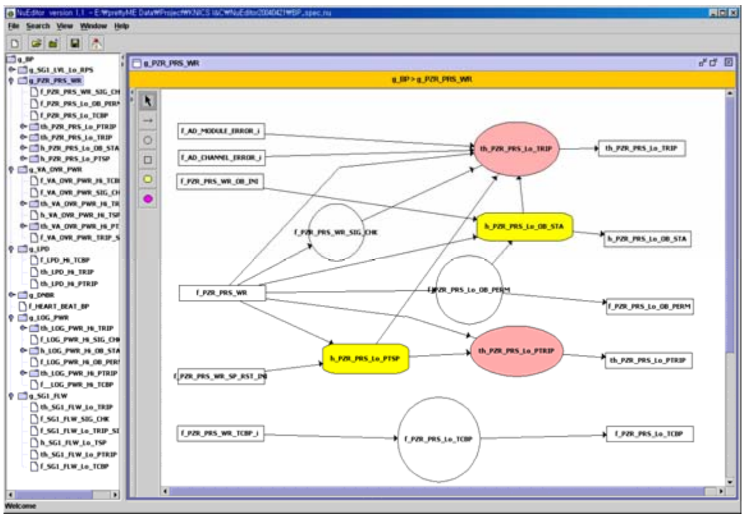


Figure 3. FBD unit example

We define the module of FBD program as a set of units, which performs a meaningful function. Each trip logic block in KNICS RPS BP can be regarded as a module, and a module is composed of several units. KNICS project uses NuSCR formal specification language (Yoo, 2003) as software requirements specification (SRS) to increase of the safety of software system. In such case, we can get a guideline from the SRS written in NuSCR. Figure 4 (a) below is an FOD (Function Overview Diagram) for a manual reset variable set-point falling trip logic named *g_PZR_PRS_WR*. It is screen-captured from the NuSCR specification and verification assistant tool (Cho, 2004). FOD, notation similar to the data-flow diagram, captures dependency among various nodes hierarchically so that complex requirements can be specified in a divide-and-conquer fashion. Each node of FOD in Figure 4 (a) is designed as an individual FBD unit. The whole FBD is a module, which consists of 7 units, gets 6 inputs and emits 5 outputs. This module can be shown in the upper hierarchy of FOD described in Figure 4 (b). FOD in Figure 4 (a) also shows many interactions among units in the module *g_PZR_PRS_WR*.

We can also define the software system of FBD program as the whole block of modules. The software system gets inputs from the outside of the system and emits outputs to the outside. In Figure 4 (b), the stand-alone node named *g_BP* is the software system. The input variables listed on the left are the software system inputs, and the output variables listed on the right are the software system output. In this way, we define the concept of unit, module, and software system in terms of FBD program testing. As we mentioned, the division of each units can be accelerated if we use the formal notation such as NuSCR. Even if we use the SRS written in natural language, this information can be easily identified.


(a) FOD for *g_PZR_PRS_WR* in KNICS RPS BP

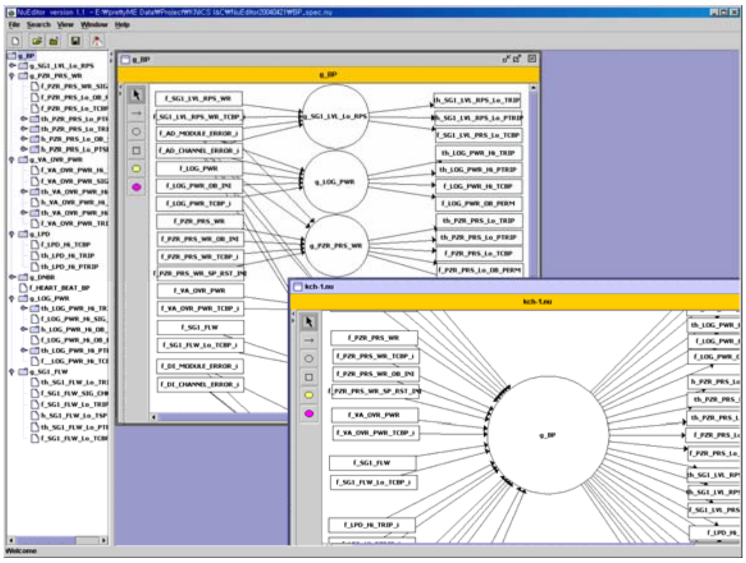

(b) FOD for *g_BP* in KNICS RPS BP

Figure 4. FODs for KNICS RPS BP

4. UNIT TESTING ON FBD PROGRAMS

In this section, we explain the overall process for FBD testing. To test the FBD program without generating intermediate code, we need to transform the FBD unit into corresponding control flow graph. The transformation is the preliminary and major step of the proposed testing process and enables to apply the existing flow graph based testing techniques to the FBD programs. After transformation, we choose such test coverage criteria as all nodes, all edges, and all paths coverage. Finally we generate a set of test cases satisfying the selected criteria. In this work, we targets only unit testing and adopt control flow coverage criteria (Jorgensen, 1995). Figure 5 describes the proposed FBD unit testing approach.

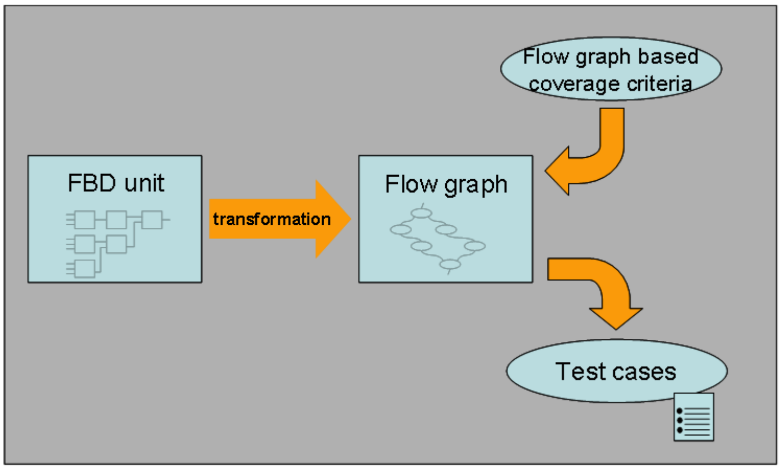


Figure 5. Overview of FBD unit testing

4.1 Control Flow Graph Transformation

We need to know the execution mechanism of FBD program for transforming the FBD unit into a flow graph. Figure 6 shows a unit FBD which calculates *th_X_Pretrip* variable. This unit FBD is a part of module *g_PZR_PRS_WR* in Figure 4. It gets a pre-trip set point value from the other unit in Figure 3 and decides the pre-trip value.

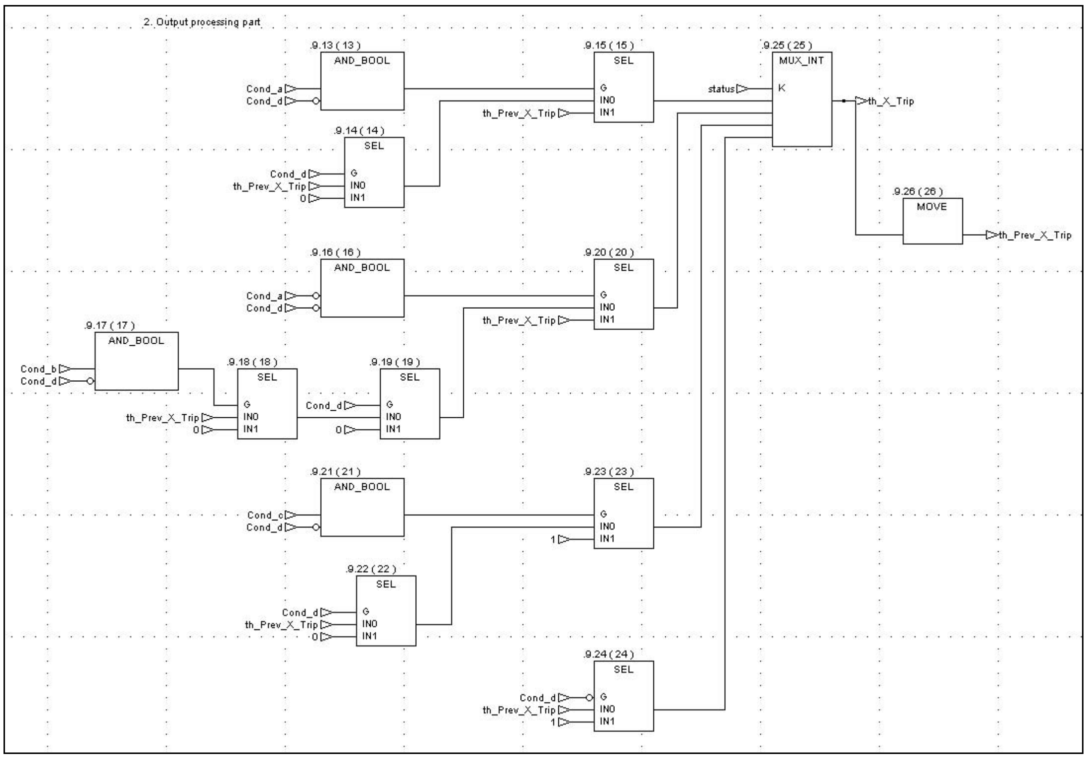


Figure 6. FBD unit for *th_X_Pretrip*

All function blocks in FBD have their own execution orders. On every scan cycle, all of them are executed sequentially according to their execution orders. The parenthesized number on the top of the each function block means its execution order. For example, AND_BOOL function block numbered (13) in Figure 6 is executed at first, and MOVE function block numbered (26) is executed at last. Before the final function block is executed, the primary output *th_X_Pretrip* is emitted. Transformed flow graph should reflect these sequential execution orders sufficiently because generated graph focuses on the flow of control and gets used to apply control flow testing coverage criteria.

The flow graph is a kind of control flow graph. Figure 7 shows transformed control flow graph. Each node represents statements of source code, which is equivalent to the behavior of function block. Arrow means the control flow of FBD program. Arithmetic operations of function blocks such as ADD_INT or MUL_INT do not make the control divergence. For example, the AND block numbered (13) in Figure 6, does not make the control to branch in Figure 7. It just computes AND-ed value of Cond_a and Cond_d'. Logical, comparison, and selection operations make the divergences. For example, selection function block numbered (14), makes the control split into two branches in the graph. One of two branches is taken according to the value of Cond_d. MUX_INT function block numbered (25) is transformed into a multi branched structure corresponding to switch statement. We need some temporary variables to store the output of each block if the calculation output does not have variable name and scheduled to be intermediately used. For example, the result of (13) function block is stored in an intermediate variable, (14) SEL function block is executed, and then the intermediate variable is used for (15) SEL function block. The result of (15) SEL function block is also stored for the input in (25) function block.

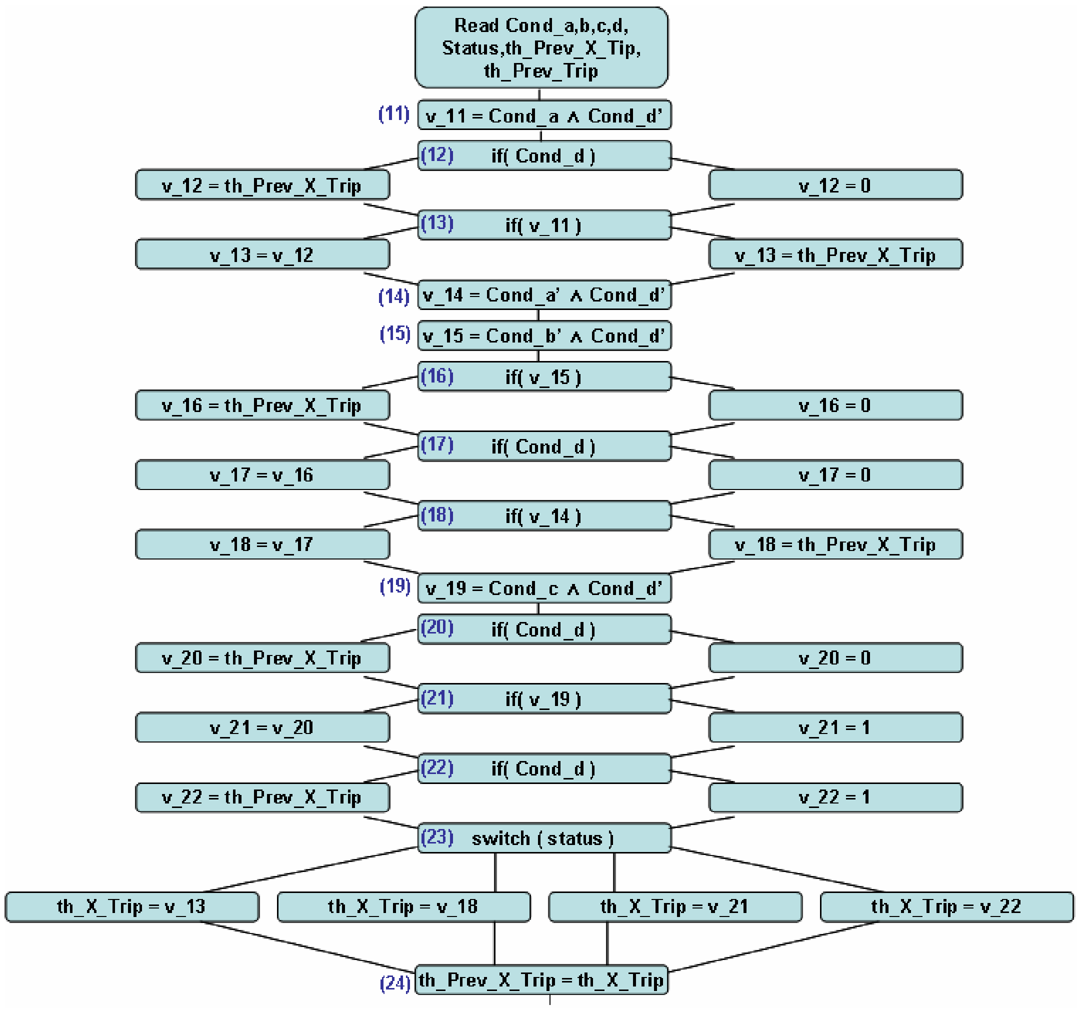


Figure 7. Control flow graph for *th_X_Pretrip* FBD unit

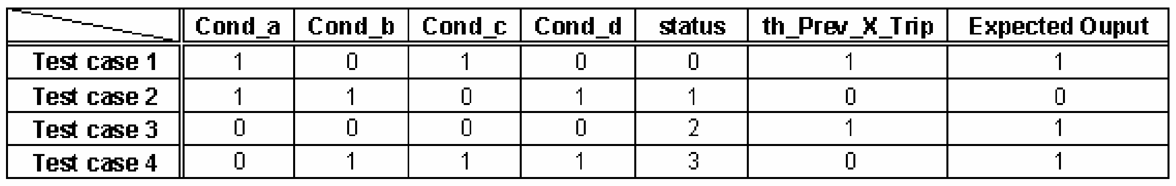
4.2 Test Coverage Criteria and Test Case Generation

After transforming the unit of FBD program to the control flow graph, we choose an adequate test coverage criterion and generate test cases satisfying the criterion. A test coverage criterion is the extent to which a set of test cases covers a program and is used to determine whether a program has been sufficiently tested. It specifies the minimal set of program entities that should be exercised by the test cases. Given a set of test cases, we can determine whether or not a given test coverage criterion is satisfied by executing the program on those cases and examining how much extent of the program is covered by those cases. There are three representative control flow test coverage criteria. In this paper, we chose all-edges test coverage criterion in the case study.

All-nodes test coverage criterion requires that each node in the control flow graph should be executed by some test cases. *All-edges test coverage criterion* requires that each edge in the control flow graph traversed during some program executions. This form of testing is also called branch testing because each branch output is exercised under this criterion. The all-edges criterion subsumes the all-nodes criterion because if all edges in the flow graph are exercised by the test cases, then it is guaranteed that all-nodes are also exercised by the test cases. *All-paths test coverage criterion* requires that every complete path in the program should be tested. Complete path means a path from the entry node to the exit node of the flow graph. This criterion subsumes the all-edges test criterion. The all-paths test criterion is very stringent, but generally impractical.

Table 1 below shows the test cases, which satisfy all-edges test coverage criterion. 6 columns are the input variables of *th_X_Pretrip* FBD unit, and the final one is the expected output. These 4 test cases cover all edges in the control flow graph shown in Figure 7.

Table 1. Test cases satisfying all-edges test coverage criterion



	Cond_a	Cond_b	Cond_c	Cond_d	status	th_Prev_X_Trip	Expected Ouput
Test case 1	1	0	1	0	0	1	1
Test case 2	1	1	0	1	1	0	0
Test case 3	0	0	0	0	2	1	1
Test case 4	0	1	1	1	3	0	1

4.3 Case Study

This subsection introduces some examples to demonstrate how the proposed FBD unit testing approach can detect faults in FBD programs. We made the FBD program for *th_X_Pretrip* in Figure 6 to have some errors. These sown errors are frequently occurred in practical FBD programming and more detailed categorization and description can be found in (Oh, 2004). All of sown errors could be detected by the test cases, identified by all-edges test coverage criteria.

Error case 1 (Switched inputs): FBD programmers often confuse the order of input variables in the function block where their orders are very important. SEL, MUX, or GE_INT function blocks belong to these function blocks. We switched the two inputs of (14) SEL function block in Figure 6, *th_Prev_X_Trip* and 0. Error case 1 can be detected by test case 1. While the expected output of test case 1 is 1, the actual output is 0.

Error case 2 (Misplaced inverters): Omission or misplacement of function block happens frequently when programming FBD. In particular, inverter block is prone to be omitted where it is necessary or misplaced where it is not necessary. We inserted an inverter in IN0 input of (20) SEL function block. Error case 2 can be detected by test case 2.

Error case 3 (Incorrect inputs): Incorrect input/output variables or value addresses the cases that a wrong variable or value is given as an input to a function block. If the wrong initial value is assigned to a variable, it is also considered as an input fault. We changed *th_Prev_X_Trip*, IN0 input variable of (22) SEL function block, into *th_Prev_Trip*. Error case 3 can be detected by test case 3.

Error case 4 (Incorrect inputs): As another example of incorrect inputs, we changed the IN1 input of (24) SEL function block from 1 into 0. Swapping input value from zero to one or from one to zero happens frequently. Error case 4 can be detected by test case 4.

5. CONCLUSIONS

In this paper, we propose a testing technique that applies existing software testing techniques to the FBD programs without generating intermediate code. The previous PLC-based software testing generates an intermediate code such as C, which is equivalent to the original FBD, and targets an intermediate code. Although the behavior of FBD is similar to a procedure or function of software, there is no systematic way to apply software testing techniques to FBD. The proposed technique does not generate intermediate code and directly applies software testing techniques to FBD.

We defined the concept of unit and integration testing in terms of FBDs. We then transform the FBD program into a control flow graph, and apply the existing control flow testing coverage criteria to the transformed graph. Test cases are generated to satisfy a given test coverage criterion. In order to demonstrate the effectiveness of the proposed method, we introduce an example of trip logic of DPPS RPS BP, which is currently developed by KNICS. We used all-edges coverage criterion, and also seed frequently occurring errors in the example FBD, which include switched input, misplaced inverter and incorrect input/output variable. They all are detected by test cases generated by the proposed approach.

We are currently focusing on the data flow testing (Frankl, 1988), which aims at exercising definition-use associations of variables in the program. The characteristics of FBD make the data flow testing more adequate for FBD unit testing because FBD intuitively expresses data flow as signal flow using graphical notation like a circuit diagram. FBD testing technique should include an approach for integration testing as well as unit testing. Integration testing assumes that each unit of the system is separately tested and targets the interface and interaction between units. In the end, it is necessary to make test coverage criteria specialized for FBD because FBD has its own characteristics. The particular characteristics of FBD such as time and history have to be taken into the consideration.

ACKNOWLEDGEMENT

This research was partially supported by the Advanced Information Technology Research Center(AITrc), Software Process Improvement Center(SPIC), and the Internet Intrusion Response Technology Research Center(IIRTRC).

REFERENCES

1. US NRC, Digital Instrumentation and Control Systems in Nuclear Power Plants: safety and reliability issues, National Academy Press, 1997.
2. IEC, International Standard for Programmable Controllers: Programming Languages(Part 3), 1993.
3. KNCIS, Korea Nuclear Instrumentation and Control System Research and Development Center, http://www.knics.re.kr/english/eindex.html.

4. A. Mader, A Classification of PLC Models and Applications, In Proc. *WODES 2000: 5th Workshop on Discrete Event Systems*, August 21-23, Gent, Belgium, 2000.
5. J. Yoo, T. Kim, S. Cha, J-S. Lee, H.S. Son, A Formal Software Requirements Specification Method for Digital Nuclear Plants Protection Systems, *Journal of Systems and Software*, in press, 2003.
6. J. Cho, J. Yoo, S. Cha, NuEditor – A Tool Suite for Specification and Verification of NuSCR, In proc. *Second ACIS International Conference on Software Engineering Research, Management and Applications (SERA2004)*, pp298-304, LA, USA, May 5-7, 2004.
7. Paul C. Jorgensen, Software testing: a craftsman's approach, CRC Press, 1995.
8. P. G. Frankl, E. J. Weyuker, An applicable family of data flow testing criteria, *IEEE Trans. Software Engineering*, **14(10)**, pp1483–1498, Oct. 1988.
9. Y. Oh, J. Yoo, S. Cha, H.S. Son, Software Safety Analysis of Function Block Diagrams using Fault Trees, *Reliability Engineering and System Safety*, in press, 2004.