Automated Test Coverage Measurement for Reactor Protection System Software Implemented in Function Block Diagram

Eunkyoung Jee[1], Suin Kim[2], Sungdeok Cha[3], and Insup Lee[1]

[1] University of Pennsylvania, 3330 Walnut Street, Philadelphia, PA 19104, USA
eunkjee@seas.upenn.edu, lee@cis.upenn.edu
[2] KAIST, 335 Gwahangno Yuseong-gu, Daejeon, Republic of Korea
suin.kim@gmail.com
[3] Korea University, Anam-dong Seongbuk-gu, Seoul, Republic of Korea
scha@korea.ac.kr

Abstract. We present *FBDTestMeasurer*, an automated test coverage measurement tool for function block diagram (FBD) programs which are increasingly used in implementing safety critical systems such as nuclear reactor protection systems. We have defined new structural test coverage criteria for FBD programs in which dataflow-centric characteristics of FBD programs were well reflected. Given an FBD program and a set of test cases, *FBDTestMeasurer* produces test coverage score and uncovered test requirements with respect to the selected coverage criteria. Visual representation of uncovered data paths enables testers to easily identify which parts of the program need to be tested further. We found many aspects of the FBD logic that were not tested sufficiently when conducting a case study using test cases prepared by domain experts for reactor protection system software. Domain experts found this technique and tool highly intuitive and useful to measure the adequacy of FBD testing and generate additional test cases.

Keywords: test coverage measurement, test automation, function block diagram, programmable logic controller.

1 Introduction

As programmable logic controllers (PLCs) are widely used to implement safety-critical systems such as nuclear reactor protection systems, testing of PLC programs is getting more important. Among the five standard PLC programming languages defined by the International Electrotechnical Commission (IEC) [1], function block diagram (FBD) is a commonly used implementation language. The Korea Nuclear Instrumentation and Control System R&D Center (KNICS) project, whose goal is to develop a comprehensive suite of digital reactor protection systems, is an example in which PLC programs implementing safety critical systems were implemented in FBD. For such safety critical systems to be approved for operation, developers must demonstrate compliance to strict quality requirements including unit testing and test result evaluations [2,3].

E. Schoitsch (Ed.): SAFECOMP 2010, LNCS 6351, pp. 223–236, 2010.

Current FBD testing relies on mostly functional testing in which test cases are manually derived from natural language requirements. Although functional testing and structural testing are complementary each other and both are required to be applied to safety critical software [3], there have been little research and practices on structural testing for FBD programs.

Another difficulty of current FBD testing is lack of test evaluation techniques. Regulation authorities such as U.S.NRC require that test results be documented and evaluated to ensure that test requirements have been satisfied [2]. Although test results for FBD programs implementing safety critical software need to be evaluated thoroughly, there have been no other methods directly applicable to FBD programs except manually reviewing and analyzing test documents for assuring test quality. Domain experts have felt that manual reviews only were not adequate to assure test quality. More systematic and quantitative ways to evaluate the adequacy of the test cases have been strongly required.

In order to enable the structural testing for FBD programs, we, software engineers, have defined structural test coverage criteria suitable to FBD programs in which the unique characteristics of the FBD language were fully reflected [4]. An FBD program is interpreted as a directed data flow graph and three test coverage criteria have been defined using the notion of the data flow path (d-path) and the d-path condition (DPC) for each d-path.

To work out a solution to lack of systematic test evaluation methods, we present an automated test coverage measurement tool, *FBDTestMeasurer*, which measures the coverage of a set of test cases on the FBD program with respect to the test coverage criteria proposed in [4]. Given a unit FBD program, a set of test cases, and selected test coverage criteria, *FBDTestMeasurer* generates test requirements with respect to the selected test coverage criteria and measures the coverage of the test cases automatically. It provides coverage score and unsatisfied test requirements as a result. Uncovered d-paths can be visually presented on the graphical view of the FBD program.

The proposed technique has following contributions: 1) automated quantitative and systematic test evaluation for FBD programs gives concrete basis of quality assurance, 2) visual representation of uncovered d-paths on the FBD program helps testers analyze the uncovered test requirements intuitively, and 3) unsatisfied test requirements provided by *FBDTestMeasurer* reveal inadequately tested parts and help testers generate additional test cases.

We conducted a case study using representative trip (shutdown) modules of the Bistable Processor (BP) of the Reactor Protection Systems (RPS) in the KNICS project. The test cases had been manually generated by the FBD testing experts working in the KNICS project. It took nearly 3 man-months to generate the test cases for the whole BP system. We could find many insufficiently tested aspects of the FBD program by the set of test cases.

The remainder of this paper is organized as follows: Section 2 provides the background for the study including a literature survey of the most relevant research. Section 3 explains the test coverage criteria for FBD programs. Section 4 presents the automated test coverage measurement techniques for FBD

programs and the related issues. Section 5 demonstrates the results of the KNICS BP case study. We conclude the paper in Section 6.

2 Related Work

PLC programs are executed in a permanent loop. In each iteration of the loop, called a scan cycle, the PLC program reads inputs, computes a new internal state and outputs, and updates outputs. This cyclic behavior makes PLCs suitable for control tasks and interaction with continuous environments [5].

FBD, one of the standard PLC programming languages, is widely used because of its graphical notations and suitability for developing applications with a high degree of data flow among the components. FBD is a data flow language based on viewing a system in terms of the flow of signals between processing elements [6]. A collection of blocks is wired together like a circuit diagram as shown in Figure 1. An example FBD network of Figure 1 is a part of the fixed-set-point-falling trip logic of the BP for the RPS. The output variable *th_X_Logic_Trip* is set to *true* if the *f_X* value falls below the trip set-point (*k_X_Trip_Setpoint*) for longer than the specified delay (*k_Trip_Delay*). The trip signal *true* would safely shut down a nuclear reactor. Blocks of FBD programs are categorized into functions and function blocks. A function does not have internal states and its output is determined solely by current inputs. In contrast, a function block maintains internal states and produces outputs. In Figure 1, the TON block is a function block, and all other blocks (e.g., ADD_INT, LE_INT, and SEL) are functions.

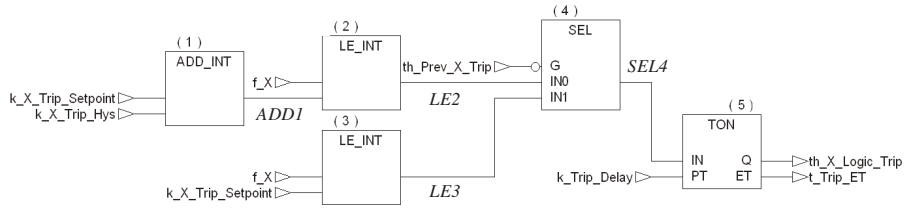


Fig. 1. A small FBD program for calculating *th_X_Logic_Trip*

We focus on unit testing for FBD programs. A unit FBD consists of blocks necessary to compute a primary output (e.g., *th_X_Logic_Trip* in Figure 1) according to the unit definition on FBD programs [7]. There have been functional testing research and practices to test FBD units. In [8], authors have developed a simulation-based validation tool named *SIVAT* which uses ANSI C code generated from FBD programs internally. In [9], an FBD program is transformed into a High Level Timed Petri Nets (HLTPN) model, and the simulation-based testing is performed on the HLTPN model. An integrated tool environment named *PLCTOOLS* has been developed to support the entire development process including specification, transformation, and simulation. Unfortunately, these approaches support

only functional testing. Neither internal structure nor the dataflow-centric aspects of FBD programs were analyzed in their testing approaches.

As a member of the KNICS project, due to lack of structural testing techniques and coverage criteria readily applicable to FBD programs, we tried to apply conventional test coverage criteria to FBD programs by transforming FBD programs into equivalent control flow graphs (CFGs) [10,11]. Although this approach contributed to make structural testing for FBD programs possible, it has limitations in that CFGs does not accurately reflect the data flow-centric characteristics of FBD. Our experience made it clear that conventional structural testing techniques and coverage criteria, originally developed for procedural programming languages, do not work well on FBD programs. We have developed new test coverage criteria for FBD programs by focusing on the data flow aspects of the FBD language [4]. Test coverage criteria can be really useful when they are integrated with automated tool supports. In this paper, we propose a test coverage measurement procedure and an automated tool based on the test coverage criteria defined in our previous work.

There have been a lot of research and tools for code coverage [12,13,14,15,16]. These approaches and our approach have common basic principles of test coverage measurement. However, these tools target the procedural languages such as C, C++, Java, Cobol, Perl, PHP, Ada, etc., not the data flow languages such as FBD and Lustre. They use the test coverage criteria defined on control flow graphs (e.g., statement coverage, decision coverage, etc.) while we use the different test coverage criteria defined on data flow graphs (e.g., basic coverage, input condition coverage, etc.).

Research of test coverage criteria for data flow languages is not new. A. Lakehal et al. [17] have defined the structural test coverage criteria for Lustre, a synchronous data-flow declarative language, based on the activation condition which specifies when the data flow from an input edge to an output edge may occur. Depending on the path length and the values taken along the edges, multiple coverage criteria were defined. They developed *Lustructu* [18], a tool for the automated coverage assessment of Lustre programs. While the concept of activation condition was useful, the approach presented in [17] was unable to cope with complex function block conditions of FBD programs because the target operators of [17] were limited to simple temporal operators. We developed a systematic way to deal with nontrivial function block conditions by identifying the internal variables and involving them in the function block conditions. In addition to customizing the activation condition concept to properly reflect the characteristics unique to FBD, we also extended their approach by supporting multiple outputs as well as non-Boolean edges.

3 Test Coverage Criteria for FBD Programs

3.1 D-Path and D-Path Condition

We have defined the structural test coverage criteria for FBD programs [4]. An FBD program is considered as a directed graph with multiple inputs and outputs.

The FBD program shown in Figure 1 consists of five blocks and 13 edges. A *d-path* is a finite sequence $\langle e_1, e_2, ..., e_n \rangle$ of edges in the directed graph of an FBD program. A unit d-path is a d-path with the length 2 in the form of $\langle e_i, e_o \rangle$. A d-path is guaranteed to be finite because FBD programs have no internal feedback loops. The *d-path condition* (DPC) of a d-path is the condition along the d-path under which the input value plays a role in computing the output. We use the *d-* prefix to distinguish the *d-path* and the *d-path condition* from the traditional *path* and the *path condition* defined in control flow graphs. The d-path condition of a d-path p, $DPC(p)$, is defined by conjunction of function condition, $FC(\langle e_i, e_{i+1} \rangle)$, for each function and function block condition, $FBC(\langle e_j, e_{j+1} \rangle)$, for each function block along the d-path.

3.2 Function Condition (FC) and Function Block Condition (FBC)

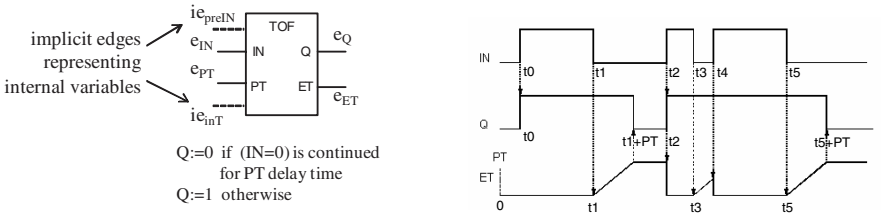
$FC(\langle e_i, e_o \rangle)$, is the condition under which the value at the output edge e_o is influenced by the value at the input edge e_i through a single function. If a function has n inputs, there exist n FCs for each d-path from an input to the output. There are three types of FCs.

For the functions belonging to type 1, all inputs always play a role in determining the output. Best illustrated by the ADD function, FCs for all the unit d-paths are *true*. In the type 2 functions, an input value appears unchanged on the output edge in a certain condition. The SEL function is an obvious example in that either e_{IN0} or e_{IN1} flows into the output unchanged depending on the value of e_G. The AND block is another example. If e_{IN1} is *true*, the value *ture* flows into the output only if the other input e_{IN2} is also *true*. If e_{IN1} is $false$, the output is also $false$ without any further constraints. Formal definitions of FCs for the AND block with two inputs $IN1$ and $IN2$ are:
if $p_1 = \langle e_{IN1}, e_{OUT} \rangle \wedge p_2 = \langle e_{IN2}, e_{OUT} \rangle \wedge e_{OUT} = AND(e_{IN1}, e_{IN2})$,

$$\begin{aligned} FC(p_1) &= if\ e_{IN1}\ then\ e_{IN2}\ else\ true \\ &= \neg e_{IN1} \vee e_{IN2} \\ FC(p_2) &= \neg e_{IN2} \vee e_{IN1} \end{aligned}$$

The type 3 functions have characteristics such that some or all input values are used in determining the output computation under specific conditions. Unlike the type 2 functions, the output of the type 3 function is not necessarily same as one of the inputs.

We categorized all FBCs into type 4. $FBC(\langle e_i, e_o \rangle)$ is same as $FC(\langle e_i, e_o \rangle)$ except e_i and e_o are connected by a single "function block". Whereas FC definitions are relatively simple, FBC definitions are more complex due to the internal variables which are modeled as the implicit edges to the function block in this approach. For example, the semantics of TOF (Timer Off Delay) function block, shown in Figure 2(a), is such that it generates the Q output $false$ when the IN input remains $false$ during the delay time specified by the variable PT ever since the IN value turned to $false$ from *true*. Otherwise, the output Q is *true*.



(a) TOF function block (b) TOF timing diagram

Fig. 2. TOF function block and its behavioral definition

The behavioral definition of timer blocks such as the TOF is described by timing diagrams as shown in Figure 2(b). It shows how outputs Q and ET vary in response to the different IN values as time passes.

When we formally defined the TOF semantics by representing all the possible inputs and output combinations of relevant variables using a condition/action table, two internal variables were identified. *preIN* and *inT* denote the value of the *IN* stored in the previous scan cycle and the internal timer, respectively.

FBCs for the output Q of TOF are defined as follows:
if $p_1 = \langle e_{IN}, e_Q \rangle \ \wedge \ p_2 = \langle e_{PT}, e_Q \rangle \wedge \ e_Q = TOF\_Q(e_{IN}, e_{PT})$,

$$\begin{aligned} FBC(p_1) &= if\ e_{IN}\ then\ true\ else\ (\neg ie_{preIN} \wedge (ie_{inT} = 0 \vee (ie_{inT} \geq e_{PT}))) \\ &= e_{IN} \vee (ie_{preIN} = 0 \wedge (ie_{inT} = 0 \vee ie_{inT} \geq e_{PT})) \\ FBC(p_2) &= (ie_{inT} > 0) \end{aligned}$$

For the $FBC(p_1)$, when the e_{IN} is *true*, it flows into the output e_Q without any constraints. If the e_{IN} is $false$, the output e_Q is also $false$ only if $(\neg ie_{preIN} \wedge (ie_{inT} = 0 \vee (ie_{inT} \geq e_{PT})))$. The *ie* represents an implicit edge as opposed to an explicit edge. We defined all FCs and FBCs for the functions and function blocks in the IEC standard[1]. Detailed definitions can be found in [19].

3.3 FBD Test Coverage Criteria

Three different test coverage criteria for FBD programs are defined based on the definition of DPCs. The process of deriving d-path condition (DPC) is similar to the one used in backward symbolic execution. Starting from the output edge of the given d-path, each FC or FBC is expanded. For example, when there are two functions and one function block in the d-path $p_{4\_1} = \langle f\_X,\ LE2,\ SEL4,\ th\_X\_Logic\_Trip \rangle$ in Figure 1, $DPC(p_{4\_1})$ is conjunction of two FCs and one FBC as follows:

$$\begin{aligned} &DPC(p_{4\_1}) \\ &\quad = DPC(\langle f\_X, LE2, SEL4, th\_X\_Logic\_Trip \rangle) \\ &\quad = FC(\langle f\_X, LE2 \rangle) \wedge FC(\langle LE2, SEL4 \rangle) \wedge FBC(\langle SEL4, th\_X\_Logic\_Trip \rangle) \end{aligned} \tag{1}$$

When the backward symbolic computation is completed, the DPC should contain only input and internal variables because all the expressions corresponding to the intermediate edges would be replaced. For example, the expression of (1) is transformed into the expression with only input and internal variables by substituting the FCs and the FBC with the corresponding expressions from (2) to (4) and then substituting the intermediate edge names with the expressions from (5) to (8).

$$FC(\langle f\_X, LE2\rangle) = true \tag{2}$$
$$FC(\langle LE2, SEL4\rangle) = th\_Prev\_X\_Trip \tag{3}$$
$$FBC(\langle SEL4, th\_X\_Logic\_Trip\rangle) = SEL4 \vee (preSEL4 = 0 \wedge (inT5 = 0 \vee inT5 \geq k\_Trip\_Delay))) \tag{4}$$

$$SEL4 = \neg th\_Prev\_X\_Trip \ ?\ LE3 : LE2 \tag{5}$$
$$LE3 = f\_X \leq k\_X\_Trip\_Setpoint \tag{6}$$
$$LE2 = f\_X \leq ADD1 \tag{7}$$
$$ADD1 = k\_X\_Trip\_Setpoint + k\_X\_Trip\_Hys \tag{8}$$

Building on the definition of DPC, the basic coverage, the input condition coverage, and the complex condition coverage have been defined for FBD programs. Let DP denote the set of all d-paths from input edges to output edges.

Definition 1. *A set of test data T satisfies the* basic coverage criterion *if and only if* $\forall p \in DP\ \exists t \in T\ |DPC(p)|_t = true$.

The basic coverage (BC) focuses on covering every d-path in the FBD program under test at least once. Test requirements for BC are DPCs for all d-paths of the target program. A test case t is meaningful if the input of the d-path p has influence in determining the output of p. Such condition is captured by $|DPC(p)|_t = true$ in the above definition. Otherwise (e.g., $|DPC(p)|_t = false$), the test case t is unable to make the input of the p flow down the given d-path and survive all the way to the output. Such test case is surely ineffective in testing the correctness of the d-path, and it fails to contribute towards meeting the coverage requirement.

While the basic coverage is straightforward in concept, it is often ineffective in detecting logical errors that FBD programs might have. Another stronger coverage is needed.

Definition 2. *A set of test data T satisfies the* input condition coverage criterion *if and only if,* $\forall p \in DP,\ \exists t \in T\ |in(p) \wedge DPC(p)|_t = true$ *and* $\exists t' \in T\ |\neg in(p) \wedge DPC(p)|_{t'} = true$ *where $in(p)$ is a Boolean input edge of the d-path p.*

To satisfy the input condition coverage (ICC) criterion, it is no longer sufficient to choose an arbitrary value for the input edge whose value would influence the outcome. One must now choose a set of test data such that input values include

both *true* and $false$ for Boolean inputs (e.g., $DPC(p_{3\_1}) \wedge th\_Prev\_X\_Trip$ as well as $DPC(p_{3\_1}) \wedge \neg th\_Prev\_X\_Trip$ for $p_{3\_1} = \langle th\_Prev\_X\_Trip, SEL4, th\_X\_Logic\_Trip \rangle$).

Definition 3. *A set of test data T satisfies the* complex condition coverage criterion *if and only if, $\forall p \in DP$, $\exists t \in T \; |e_i \wedge DPC(p)|_t = true$ and $\exists t' \in T \; |\neg e_i \wedge DPC(p)|_{t'} = true$ where e_i is a Boolean edge in the d-path p of length n and $1 \leq i \leq n$.*

The complex condition coverage (CCC) criterion which is stronger than the ICC requires that every Boolean edge's variation in the d-path be tested at least once with the satisfied DPC. Every test set satisfying the ICC criterion also satisfies the BC criterion. Similarly, the CCC criterion subsumes both the ICC and the BC criteria.

4 Automated Test Coverage Measurement for FBD Programs

4.1 FBDTestMeasurer

Test coverage measurement is a general method to evaluate test adequacy. We developed *FBDTestMeasurer* to measure coverage of a set of test cases with respect to the structural test coverage criteria for FBD programs automatically. Figure 3 shows the architecture of *FBDTestMeasurer*.

Parsing. *FBDTestMeasurer* receives a unit FBD program in the standard XML format and extracts d-paths for the selected outputs. A unit FBD program may have many outputs, but there are usually one or a few primary outputs on which analysis should focus. *FBDTestMeasurer* allows users to choose output variables which they want to analyze.

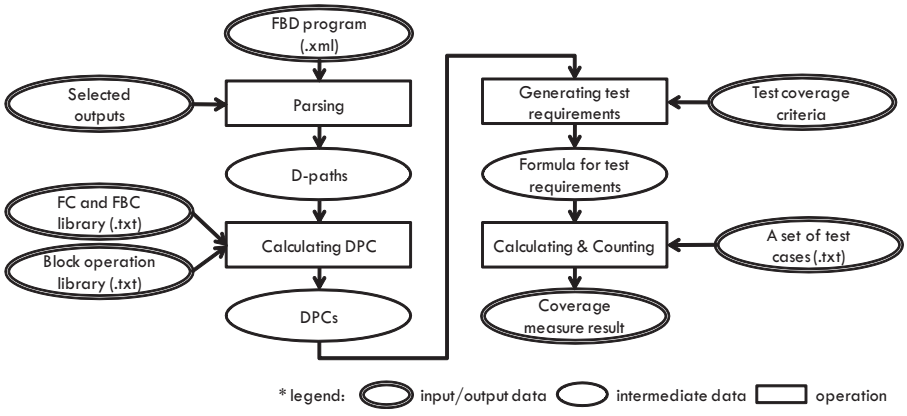


Fig. 3. *FBDTestMeasurer* architecture

Calculating DPC. DPC calculation requires an FC and FBC library and a block operation library as well as d-path information. DPC is the conjunction of FCs and FBCs. When calculating DPC, *FBDTestMeasurer* searches the necessary FC or FBC templates from the FC and FBC library which includes FCs and FBCs for all functions and function blocks in the IEC standard[1].

For example, expressions (2), (3), and (4) are calculated by extracting corresponding FC and FBC templates from the FC and FBC library. The block operation library is necessary to replace intermediate edge names in the DPC with corresponding operational descriptions. *FBDTestMeasurer* searches for the corresponding block's operational description from the block operation library in order to make the DPC contain only input and internal variables. For example, $SEL4$, an intermediate edge name, included in the expression (4) is replaced by $\neg th\_Prev\_X\_Trip \ ? \ LE3 : LE2$ shown in expression (5) after extracting the operational description template for the SEL from the block operation library. $LE3$ and $LE2$ are replaced by expression (6) and (7), respectively.

We decided to keep FC/FBC information and block operation information in separate library files for flexible capability to cope with new blocks. FBD programs can have various kinds of blocks and many PLC case tools allow users to make user-defined blocks. When new blocks are used in the FBD program, DPC calculation still works well if users simply inserts the FCs or FBCs and the operational descriptions of the new blocks into the library files.

Generating test requirements. *FBDTestMeasurer* enables users to select test coverage criteria which they want to specify. One or more test coverage criteria can be selected. According to the selected test coverage criteria, *FBDTestMeasurer* generates test requirements. All test requirements are represented by logical formula connected by conjunction.

Calculating and Counting. *FBDTestMeasurer* receives a set of test cases. We made a textual file template for specifying test cases of FBD programs. If assigning input values of a test case to a test requirement makes the test requirement *true*, the test requirement is covered by the test case. *FBDTestMeasurer* counts test requirements covered at least once by the test cases. After counting covered test requirements, *FBDTestMeasurer* provides test coverage score, i.e., percentage of the number of covered test requirements divided by the number of all test requirements, and uncovered test requirements.

Figure 4 shows a screen shot of *FBDTestMeasurer* which consists of several parts: input files open, d-path finder, user's selection, graphical view of the target FBD program, and result console. Given a unit FBD program and a set of test cases by opening files, *FBDTestMeasurer* presents a graphical view of the target program. When a user selects test coverage criteria and output variables in the left window of the tool, *FBDTestMeasurer* shows the coverage measurement result in the output console and produce a log file.

We implemented a D-Path Finder feature which visually highlights a d-path with the number which the user specifies. Specially, this function is highly effective to reveal which parts of the program were not covered. When the

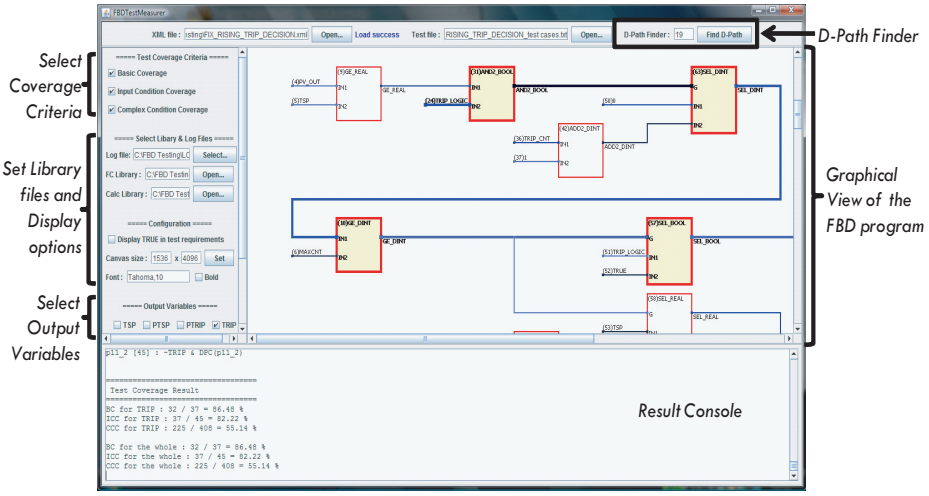


Fig. 4. Screenshot of *FBDTestMeasurer*

FBDTestMeasurer has generated uncovered test requirements, a user can identify uncovered parts in the target program clearly by inserting the number of the uncovered d-path into the D-Path Finder.

4.2 Test Coverage Measurement Issues

Internal Variables. FBD consists of functions and function blocks. If an FBD program under test consists of functions only, test requirement formulas only contain input variables. However, when the target FBD program includes function blocks, test requirement formulas contain internal variables as well as input variables. For example, $DPC(p_{3\_1})$ for $p_{3\_1} = \langle\ th\_Prev\_X\_Trip,\ SEL4,\ th\_X\_Logic\_Trip\ \rangle$ in Figure 1 as follows:

$$DPC(p_{3\_1}) = (SEL4 \vee\ \ (preSEL4 = 0 \wedge (inT5 = 0 \vee inT5 \geq k\_Trip\_Delay))) \tag{9}$$

In the DPC equation (9), $preSEL4$ and $inT5$ are internal variables denoting the value $SEL4$ stored in the previous scan cycle and the internal timer of TON, respectively. When we measure test coverage of a set of test cases for the FBD program including function blocks, we should track the values of internal variables as well as input variables.

There are two input variables, *f_X* and *th_Prev_X_Trip*, in the FBD program shown in Figure 1 assuming that *k_X_Trip_Setpoint*, *k_X_Trip_Hys*, and *k_Trip_Delay* are constants whose values are 95, 1, and 100, respectively. We assume that the scan time is 50ms. Table 1 shows three test cases for the FBD program in Figure 1. The test case description includes two internal variables as well as two input variables because the FBD program contains a function block

Table 1. A set of test cases for the FBD program shown in Figure 1

Test cases	Inputs				Expected output		
	Internal vars (Precondition)		Input vars		Output var	Internal vars (Post condition)	
	preSEL4	inT5	f_X	th_Prev_X_Trip	th_X_Logic_Trip	preSEL4	inT5
T1	false	0	90	false	false	true	50
T2	true	50	87	false	false	true	100
T3	true	100	85	false	true	true	100

and two internal variables involved in the internal state of the FBD program. Internal variables on inputs are considered as the precondition and internal variables on outputs are considered as the postcondition of the test cases. Every internal variable's value should be traced since they are used in the DPC computation. *FBDTestMeasurer* can deal with FBD programs including function blocks as well as functions by keeping track of all internal variables.

Loop. D-paths in FBD programs are always finite because FBD programs do not allow internal loops. On the other hand, the cyclic and infinite execution, an essential characteristic of the PLC programs, can be considered an "external loop". We assumed that a test case is executed on a scan cycle. Testing of FBD programs containing only functions (e.g., no internal states) is straightforward. Each test case is independent from others, and the ordering of test cases is irrelevant. However, if an FBD program contains function blocks, the sequence of test cases becomes important due to internal states.

Infeasible Test Requirements. It may be impossible for a set of test cases to achieve 100% coverage for any coverage criterion because some test requirements may turn out to be infeasible. Even though infeasible test requirements do not necessarily imply FBD programming errors, such possibility is high. Analyzing causes of the infeasible test requirements can give valuable information to find programming errors or improve the logical structure of the FBD program.

5 Case Study

We applied the proposed technique to two submodules of the 18 trip logics, _5_TRIP and _D_TRIP_LOGIC, in the BP design from the KNICS project. The BP performs a core logic to determine the trip status which makes nuclear reactor stop. The BP is a safety critical system required to be tested thoroughly by government regulation authority. The BP has 190 pages of software design specification and the whole BP consists of over one thousand function blocks and about one thousand variables. The unit test report [20] for the BP consists of 139 pages and has more than 300 test cases. Testers have executed the set of test cases on the BP PLC using a signal generator.

Table 2. Submodule information and coverage assessment result

sub-module	blocks	inputs	test cases	output variable	d-paths	BC	ICC	CCC
_5_TRIP	33	15	11	TRIP	37	86% (32/37)	82% (37/45)	55% (225/408)
				PTRIP	37	86% (32/37)	82% (37/45)	55% (225/408)
_D_TRIP _LOGIC	52	23	19	TRIP _LOGIC	305	69% (209/305)	62% (232/375)	48% (1843/3870)
				PTRIP _LOGIC	1259	32% (408/1259)	28% (426/1546)	20% (3545/17540)

In the KNICS project, once a testing team finished unit testing, a separate V&V team examined and validated the testing result. The testing team experienced a problem of assuring whether they performed adequate tests and the V&V team felt difficulties in measuring the adequacy of the executed tests because there had been no readily applicable test coverage criteria and automated testing tools for FBD programs.

Table 2 shows the size information and the coverage assessment result. _5_TRIP submodule is simple and _D_TRIP_LOGIC submodule is rather complex. We chose two modules representative enough of the BP design in terms of size and complexity. According to the unit test result document [20], there were 11 and 19 test cases for each, respectively. We made no simplification on the FBD design, and we used test cases prepared by FBD testing professionals in entirety for evaluating the adequacy of the test cases. It took about 6 weeks for two skilled FBD engineers to document the FBD testing plan and to generate test cases for the whole BP system.

_5_TRIP submodule consists of 33 functions and more than 80 edges, and there are 37 d-paths for the output TRIP whose length varies from 2 to 11. Eleven different test cases, each with 9 inputs, were subject to coverage analysis with respect to BC, ICC, and CCC. The other 6 inputs of total 15 inputs are constant inputs. Test requirements for the output TRIP grew from 37 for BC to 45 and 408 for ICC and CCC, respectively. We found that five DPCs for the output TRIP were never covered, and the BC coverage measure was 86% (or 32 out of 37). When the same design and test cases were evaluated using ICC and CCC, coverage measure dropped to about 82% (or 37 out of 45) and 55% (or 225 out of 408), respectively. Coverage measurement result for the output PTRIP was same as for TRIP. Coverage achievement for the _D_TRIP_LOGIC submodule was much lower than for the _5_TRIP submodule. BC, ICC, and CCC of the test set for the output PTRIP_LOGIC was only 32%, 28%, and 20%, respectively.

Test cases derived by domain experts achieved only 86%, 86%, 69%, and 32% of the BC for the outputs of two submodules, respectively, although the definition is relatively simple. In fact, when informed on coverage measures, domain experts were surprised that their test cases failed to investigate FBD programs in adequate depth.

Visual highlighting of d-path, one of functions supported by *FBDTestMeasurer*, was helpful to detect which d-paths were not adequately tested and which d-paths were involved in making infeasible test requirements.

This case study convincingly demonstrated that the proposed idea is highly effective in revealing which logical aspects of the FBD design remain untested, assessing quality of test cases, and monitoring progress towards meeting the mandated quality goals.

6 Conclusion

We presented *FBDTestMeasurer*, an automated test coverage measurement tool for FBD programs. We have defined new structural test coverage criteria suitable for FBD programs in our previous research. Given an FBD program and a set of test cases, *FBDTestMeasurer* generates test requirements with respect to the chosen structural test coverage criteria and performs coverage assessment of the set of test cases. *FBDTestMeasurer* provides testers with the unsatisfied test requirements and also supports visual representations of the uncovered d-paths. These features help testers to find inadequately tested parts of the FBD program and to generate additional test cases efficiently. The result of the KNICS case study convincingly demonstrated the effectiveness of the proposed techniques. Our experiment revealed which logical aspects of the FBD design were not sufficiently tested by the test cases prepared by the FBD testing professionals. The domain experts found the techniques and the tool highly useful to demonstrate the adequacy of the FBD testing quantitatively and to improve it. We are currently developing automated test case generation techniques for FBD programs.

Acknowledgments. This research was supported in part by NSF CNS-0720518, NSF CNS-0721541, and NSF CNS-0720703.

References

1. IEC: International Standard for Programmable Controllers: Programming Languages Part 3
2. USNRC: Software Test Documentation for Digital Computer Software Used in Safety Systems of Nuclear Power Plants, Regulatory Guide 1.170 (September 1997)
3. USNRC: Software Unit Testing for Digital Computer Software Used in Safety Systems of Nuclear Power Plants, Regulatory Guide 1.171 (September 1997)
4. Jee, E., Yoo, J., Cha, S., Bae, D.: A data flow-based structural testing technique for FBD programs. Information and Software Technology 51(7), 1131–1139 (2009)
5. Mader, A.: A classification of PLC models and applications. In: Proceedings of the 5th International Workshop on Discrete Event Systems (2000)
6. Lewis, R.: Programming industrial control systems using IEC 1131-3, IEE Control Engineering Series, Revised ed. The Institute of Electrical Engineers (1998)
7. Yoo, J., Park, S., Bang, H., Kim, T., Cha, S.: Direct control flow testing on function block diagrams. In: Proceedings of the 6th International Topical Meeting on Nuclear Reactor Thermal Hydraulics, Operations and Safety (October 2004)

8. Richter, S., Wittig, J.: Verification and validation process for safety I&C systems. Nuclear Plant Journal, 36–40 (May-June 2003)
9. Baresi, L., Mauri, M., Monti, A., Pezze, M.: Formal validation, and code generation for programmable controllers. In: Proceedings of the IEEE International Conference on System, Man, and Cybernetics, pp. 2437–2442 (October 2000)
10. Jee, E., Yoo, J., Cha, S.: Control and data flow testing on function block diagrams. In: Proceedings of the 24th International Conference on Computer Safety, Reliability and Security, pp. 67–80 (September 2005)
11. Jee, E., Jeon, S., Bang, H., Cha, S., Yoo, J., Park, G., Kwon, K.: Testing of timer function blocks in FBD. In: Proceedings of the 13th Asia Pacific Software Engineering Conference, pp. 243–250 (December 2006)
12. Yang, Q., Li, J.J., Weiss, D.M.: A survey of coverage-based testing tools. The Computer Journal 52(5), 589–597 (2009)
13. Parasoft: Insure++, `http://www.parasoft.com/jsp/products/insure.jsp`
14. IBM: Rational Test RealTime, `http://www-01.ibm.com/software/awdtools/test/realtime/`
15. Aivosto: VB Watch, `http://www.aivosto.com/vbwatch.html`
16. VectorSoftware: VectorCAST/Cover, `http://www.vectorcast.com/software-testing-products/embedded-code-coverage.php`
17. Lakehal, A., Parissis, I.: Structural test coverage criteria for Lustre programs. In: Proceedings of the 10th International Workshop on Formal Methods for Industrial Critical Systems (September 2005)
18. Lakehal, A., Parissis, I.: Lustructu: A tool for the automatic coverage assessment of Lustre programs. In: Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering, pp. 301–310 (November 2005)
19. Jee, E.: A Data Flow-Based Structural Testing Technique for FBD Programs. PhD thesis, KAIST (2009)
20. Korea Atomic Energy Research Institute: KNICS-RPS-STR141 (Rev.00) - Software Test Result for the Bistable Processor of the Reactor Protection System (2006)