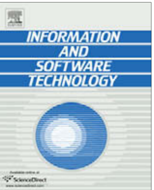
Information and Software Technology 51 (2009) 1131–1139

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof



A data flow-based structural testing technique for FBD programs

Eunkyoung Jee [a], Junbeom Yoo [b], Sungdeok Cha [c,*], Doohwan Bae [a]

[a] Div. of Computer Science, EECS Department, KAIST, 335 Gwahangno, Yuseong-gu, Daejeon 305-701, Republic of Korea
[b] Div. of Computer Science and Engineering, Konkuk University, 1 Hwayang-dong, Gwangjin-gu, Seoul 143-701, Republic of Korea
[c] Dept. of Computer Science and Engineering, Korea University, Anam-dong, Seongbuk-Gu, Seoul 136-701, Republic of Korea

ARTICLE INFO

Article history:
Received 4 August 2008
Received in revised form 22 January 2009
Accepted 24 January 2009
Available online 10 March 2009

Keywords:
Software testing
Structural testing
Test coverage criteria
Programmable logic controller
Function block diagram

ABSTRACT

With increased use of programmable logic controllers (PLCs) in implementing critical systems, quality assurance became an important issue. Regulation requires structural testing be performed for safety-critical systems by identifying coverage criteria to be satisfied and accomplishment measured. Classical coverage criteria, based on control flow graphs, are inadequate when applied to a data flow language function block diagram (FBD) which is a PLC programming language widely used in industry. We propose three structural coverage criteria for FBD programs, analyze relationship among them, and demonstrate their effectiveness using a real-world reactor protection system. Using test cases that had been manually prepared by FBD testing professionals, our technique found many aspects of the FBD logic that were not tested sufficiently. Domain experts, having found the approach highly intuitive, found the technique effective.

© 2009 Elsevier B.V. All rights reserved.

1. Introduction

PLCs are widely used to implement safety-critical systems such as nuclear power plant control software. Among the five standard PLC programming languages defined by the International Electrotechnical Commission (IEC) [1], FBD is a commonly used implementation language. The Korea Nuclear Instrumentation and Control System R&D Center (KNICS) project [2], whose goal is to develop a comprehensive suite of digital reactor protection system, is an example. For such systems be approved for operation, developer must demonstrate compliance to strict quality requirements including unit testing and coverage measurement. NRC (Nuclear Regulatory Commission) [3] mandates that software unit testing for safety-critical systems as follows:

> ...The two aspects of test coverage that are particularly important for the unit testing of safety system software are *coverage of requirements* and *coverage of the internal structure of the code*. ... For safety system software, the unit test coverage criteria to be employed should be identified and justified. ... [USNRC Regulation Guide 1.171] [4]

"Coverage of requirements" is usually demonstrated by domain experts conducting functional testing in which test cases are manually derived from requirements expressed in natural language. When testing FBD software, simulation-based functional testing is often performed on intermediate C code or other model generated from FBD programs [5,6].

Structural testing techniques and test coverage criteria suitable to FBD programs have not been well-established. Although some PLC vendors (e.g., AREVA NP) provide intermediate C code generation facility, generated code is often used only for simulation purpose and not for structural testing. Even if C code is subject to structural testing using well-known coverage criteria, there are several limitations when applied directly to FBD code. First, as C code generation scheme is not standardized, coverage measure for a given FBD might be different from one vendor to another. More importantly, C code does not map directly to the original FBD programs, and coverage analysis result would be difficult to understand to FBD testers.

As a member of the KNICS project, due to lack of structural testing technique and coverage criteria readily applicable on FBD, we tried to apply existing techniques and coverage criteria as much as practical by transforming an FBD program into equivalent control flow graph (CFG) [7,8]. Unfortunately, CFG does not accurately reflect data flow-centric characteristics of FBD. It was difficult to accurately express data flow relations in control flow graphs and formally define a relationship between parts of the control flow graph and the corresponding FBD blocks. Our experience made it clear that conventional structural testing techniques and coverage criteria, originally developed with procedural programming languages like C and JAVA in mind, do not work well on FBD programs.

This paper proposes a new structural testing technique and three different coverage criteria in which characteristics unique

* Corresponding author. Tel.: +82 2 3290 4844; fax: +82 42 350 8488.
E-mail addresses: scha@korea.ac.kr, sungdeok.cha@gmail.com (S. Cha).

0950-5849/$ - see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2009.01.003

to FBDs are fully reflected. We interpret an FBD program as a directed data flow graph and define the notion of data flow path (*d*-path) on the FBD structure. Each *d*-path is a finite sequence of edges along which input data "flow" to the output. We also define *d*-path condition (DPC) for each *d*-path on the basis of function conditions and function block conditions. We propose three test coverage criteria for FBD programs using *d*-path conditions so that adequacy of the test cases against the selected criterion can be quantitatively measured. Analysis of uncovered *d*-paths can also assist in automatically generating additional test cases necessary to meet the required coverage goal.

A case study, whose details are explained in Section 4, was conducted in close collaboration with nuclear engineers and FBD test engineers using a preliminary version of the trip (shutdown) logic of bistable processor (BP) of reactor protection systems (RPS). Intermediate C code needs not be generated, and results apply directly to FBD programs. It must be emphasized that case study was not conducted on an artificially generated and toy-sized problem as is often the case with many research papers. We used two representative trip modules of the BP subsystem without simplification and did not arbitrarily choose test cases used in the case study. Test cases had been manually generated by FBD testing professional working on the KNICS project, and it took nearly 3 man-months to generate test cases for whole BP system. Coverage analysis revealed only 90% and 68% achievement with respect to the basic coverage criteria we defined. FBD testing professionals found the technique intuitive and easy to understand.

The remainder of the 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 formal definitions of basic elements and test coverage criteria for FBD programs. Section 4 reports results of a KNICS BP case study and evaluates the proposed approach. We conclude the paper at Section 5.

2. Related work

2.1. Function block diagram

The main characteristic of the PLC programs is indefinite and cyclic execution [9]. Program reads input, computes new internal states, and updates output in each scan cycle. Such behavior makes PLC suitable for interactions with continuous environments. FBD, one of the standard PLC programming languages, is widely used because of its graphical notations and ease of developing applications with a high degree of data flow among the components.

FBD is a data flow language. It is based on viewing a system in terms of the flow of signals between processing elements [10]. A collection of blocks is wired together like a circuit diagram as shown in Fig. 1. Each block, either a function or function block, implements a primitive operation (e.g., ADD, SEL (Selection), or TON (Timer On)), and edges represent data flow. 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 Fig. 1, the TON block is a function block, and all other blocks (e.g., ADD_INT, LE_INT, and SEL) are functions. Number associated with each block denotes its execution order in the sequence.

Fig. 1 is an example FBD network used throughout the paper. It calculates the output *th_X_Logic_Trip* and is a part of the fixed-set-point-falling trip logic of a BP for RPS. (The software requirement specification document for BP is about 200 pages long, and its FBD implementation consists of more than 2000 function blocks and variables.) The output variable *th_X_Logic_Trip* is set to *true* if *f_X* value falls below the trip set-point (*k_X_Trip_Setpoint*) for longer than the specified delay (*k_Trip_Delay*). Trip signal *true* would safely shut down a nuclear reactor.

2.2. FBD unit testing

In this paper, we focus on unit testing of FBD programs. A unit FBD program consists of function blocks necessary to compute a primary output (e.g. *th_X_Logic_Trip* in Fig. 1) [11]. The primary output, stored in the PLC memory, becomes an external output or used internally as input to other FBD units.

Functional testing techniques have been used to test FBD units. The approach of [5] used ANSI C code generated from FBD program and developed a simulation-based validation tool named *SIVAT*. In [6], FBD program was transformed into High Level Timed Petri Nets (HLTPN) model, and simulation-based testing was performed on the HLTPN model. They developed an integrated tool environment named *PLCTOOLS* to support the entire development process including specification, transformation, and simulation. Unfortunately, such approaches do not support analysis on internal structure and data flow-centric aspects of FBD programs.

Although structural testing on FBD programs is mandated, there have been no well-established guidelines on structural testing for FBD programs. Existing structural test coverage criteria [12–14] are based on CFG and procedural languages. A structured CFG enforces single entry and single exit node principle. Each node represents a basic block which consists of the maximal sequence of instructions without jumps. FBD, like other data flow languages such as Lustre and LabView, is based on data flow graph (DFG) [15] model. In DFG, however, each node and edge represents an operation and a data flow, respectively. Consequently, a DFG usually has multiple entries and exits, and control flow is not shown explicitly. When CFG-based coverage criteria are applied to DFG, any test case, by definition, would cover all the blocks and edges included in the DFG.

Attempting to adapt conventional structural testing techniques is an obvious choice when performing structural testing on FBD programs. In [7,8], we transformed an FBD program into a semantically equivalent control flow graph using templates for each function or function block, while taking the execution order into consideration. Conventional test coverage criteria (e.g., all-edges or all-uses) were then applied and test cases generated. Unfortunately, domain experts experienced difficulty in interpreting the

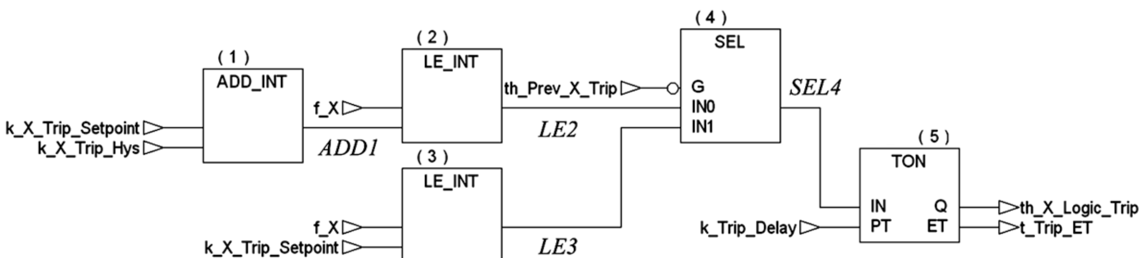


Fig. 1. A small FBD program for calculating *th_X_Logic_Trip*.

results. Even when some of the uncovered CFG nodes, edges, or paths were found, results did not clearly identify the FBD blocks or data flow that had have not been adequately tested. Most importantly, CFGs and DFGs are based on different computational model, and logical errors often made in implementation are unlikely to be the same. For example, all-branches coverage criterion on CFG is based on the assumption that incorrect branches are the most frequent errors. It is unclear what certain (e.g., say 95%) branch coverage means when applied to DFG. That is why a set of new coverage criteria which accurately reflect data flow-centric aspect of DFG is needed to effectively test FBD programs. Likewise, it is awkward to apply conventional "data flow testing" coverage criteria (e.g., all-defs, all-uses, etc.) as is to DFGs for the same reason.

2.3. Lustre approach

Research on test coverage criteria for data flow languages is not new. For example, [16] defined structural test coverage criteria for Lustre which is a synchronous data-flow declarative language and often used to specify safety-critical systems. A Lustre program is treated as a directed graph called an operator network, and the activation condition specifies when 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.

While activation condition concept is useful, [16] has weakness in that operators are limited to simple temporal operators and that it is unable to cope with complex function block conditions. Formal definition of activation condition for function blocks with multiple outputs is necessary for the approach to become applicable to large and complex real-world projects. In addition to customizing the activation condition concept to properly reflect characteristics unique to FBD, we also extended the previous approach by supporting multiple outputs as well as non-Boolean edges.

3. Structural test coverage criteria for FBD programs

3.1. Basic definition: d-path and d-path condition

We define structural test coverage criteria for FBD programs based on a formal definition of function block (FB) and *d*-path. An FBD program *F*, a directed graph which consists of multiple inputs and outputs, is defined as a tuple, $F = \langle FBs, V, E\rangle$, where *FBs* is a set of function or function block instances, *V* is a set of variable, and *E* is a set of edges. An edge connects one block to another block or a variable. The FBD program shown in Fig. 1 consists of five blocks and 13 edges, and there are seven entries and two exits. As internal edges are not explicitly named, we assign unique names (e.g., *ADD*1 or *LE*2) to all internal edges.

When a function or function block has *n* inputs and *m* outputs, its outputs are $e_{OUTi} = OOO_{OUTi}(e_{IN1}, e_{IN2}, \ldots, e_{INn})$ where *e* means an edge, OOO is the name of the function or function block, and $1 \leqslant i \leqslant m$. Fig. 2 shows an arbitrary function. If the OOO in Fig. 2 is *ADD*, the output of the *ADD* function is defined as $e_{OUT} = ADD(e_{IN1}, e_{IN2}, \ldots, e_{INn})$.

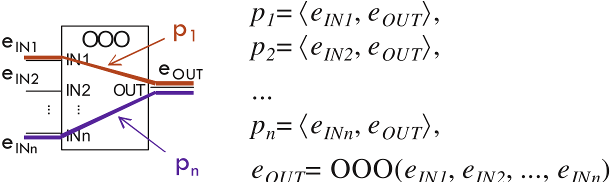


Fig. 2. An arbitrary function and its formal representation.

d-path, which we define in this paper and attach d- prefix to distinguish it from the general path in the control flow graph, is defined as follows:

Definition 1. (**d-path**) A *d*-path is a finite sequence $\langle e_1, e_2, \ldots, e_n\rangle$ of edges, where $\forall i \in [1, n-1], e_i, e_{i+1} \in E$ and e_{i+1} is a successor of e_i. ⌟

The length of a *d*-path is same as the number of edges included in it. A unit *d*-path is a *d*-path with length 2 in the form $\langle e_i, e_o\rangle$. Semantics of a function or function block is defined by a set of unit *d*-paths from an input edge to an output edge. In Fig. 2, p_1, p_2, p_n, etc. represent unit *d*-paths. A *d*-path is guaranteed to be finite because FBD programs have no internal feedback loops.

Let *DP* denote the set of all *d*-paths from input edges to output edges. Let DP_n denote the set of all *d*-paths of length *n*. If the maximum length of the *d*-paths in an FBD program is n, DP of the FBD program is represented by $DP = \bigcup_{i=1\ldots n} DP_i$. If there are several *d*-paths of the same length, we uniquely identify each by attaching another suffix (e.g., p_{51} and p_{52} are two *d*-paths of length 5). For example, FBD program shown in Fig. 1 has seven *d*-paths defined for the output *th_X_Logic_Trip* and the maximum *d*-path length is 5. Therefore, $DP = DP_5 \cup DP_4 \cup DP_3 \cup DP_2 \cup DP_1$, and illustrative examples include:

$$DP_5 = \{p_{51}, p_{52}\} - - - \text{of length } 5$$
$$p_{51} = \langle k_X_Trip_Setpoint, ADD1, LE2, SEL4, th_X_Logic_Trip\rangle$$
$$p_{52} = \langle k_X_Trip_Hys, ADD1, LE2, SEL4, th_X_Logic_Trip\rangle$$
$$DP_2 = \{p_{21}\} - - - \text{of length } 2$$
$$p_{21} = \langle k_Trip_Delay, th_X_Logic_Trip\rangle$$

d-path condition (DPC) of a *d*-path *p*, similar to the activation condition first proposed in [16], is the condition along the *d*-path of an FBD program under which input value plays a role in computing the output. We use the *d*- prefix to distinguish it from the traditional *path condition* defined on a control flow graph. The *d*-path condition is defined by a function $DPC : DP \to EX$ where *DP* and *EX* refer to the set of all *d*-paths and a set of logical expressions composing of variables, respectively.

Definition 2. (**d-path condition**) The *d*-path condition of a *d*-path of length n, $DPC(p_n)$, is defined recursively as follows:

$$DPC(p_n) = \begin{cases} true & \text{if } n = 1 \\ DPC(p_{n-1}) \wedge FC(\langle e_{n-1}, e_n\rangle) & \text{if } n \geqslant 2 \text{ and } \langle e_{n-1}, e_n\rangle \\ & \quad \text{is connected by a function} \\ DPC(p_{n-1}) \wedge FBC(\langle e_{n-1}, e_n\rangle) & \text{if } n \geqslant 2 \text{ and } \langle e_{n-1}, e_n\rangle \\ & \quad \text{is connected by a function block} \end{cases}$$

where function condition, $FC(\langle e_{n-1}, e_n\rangle)$, is defined for each function, and function block condition, $FBC(\langle e_{n-1}, e_n\rangle)$, is defined for each function block. ⌟

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 exists *n* FCs for each *d*-path from an input to the output. Fig. 3 shows the definition of FCs for representative functions. There are three different types of FCs. Type 4, similar definition for function blocks, will be discussed in the last.

Type 1: All inputs always play a role in determining the output. Best illustrated by the *ADD* function, FC is *true* for all the unit paths. Functions belonging to type 1 include all the functions defined in the arithmetic, converter, and numerical groups as well as some of the logic group.

Types		Representative Examples	FCs or FBCs	Others
Function	Type1	ADD e_{IN1} IN1 e_{IN2} IN2 OUT e_{OUT}	• FC(<e_{IN1}, e_{OUT}>) = FC(<e_{IN2}, e_{OUT}>) = true	[Arithmetic] DIV, MOD, MOVE, MUL, SUB, EXPT [Logic] NOT, ROL, ROR, SHL, SHR [Converter] BOOL_TO, INT_TO, ... [Numerical] ABS, COS, ...
	Type2	AND e_{IN1} IN1 e_{IN2} IN2 OUT e_{OUT}	• FC(<e_{IN1}, e_{OUT}>) = $\neg e_{IN1} \vee e_{IN2}$ • FC(<e_{IN2}, e_{OUT}>) = $\neg e_{IN2} \vee e_{IN1}$	[Logic] OR, XOR [Selection] LIMIT, MAX, MIN, MUX
		SEL e_G G e_{IN0} IN0 e_{IN1} IN1 OUT e_{OUT}	• FC(<e_G, e_{OUT}>) = true • FC(<e_{IN0}, e_{OUT}>) = $\neg e_G$ • FC(<e_{IN1}, e_{OUT}>) = e_G	
	Type3	LE e_{IN1} IN1 e_{IN2} IN2 e_{IN3} IN3 OUT e_{OUT}	When there are 2 inputs, • FC(<e_{IN1}, e_{OUT}>) = FC(<e_{IN2}, e_{OUT}>) = true When there are 3 inputs, • FC(<e_{IN1}, e_{OUT}>) = $\neg(e_{IN1} \le e_{IN2}) \vee (e_{IN2} \le e_{IN3})$ • FC(<e_{IN2}, e_{OUT}>) = true • FC(<e_{IN3}, e_{OUT}>) = $\neg(e_{IN2} \le e_{IN3}) \vee (e_{IN1} \le e_{IN2})$	[Comparison] EQ, GE, GT, LT, NE
Function Block	Type4	TOF e_{IN} IN e_{PT} PT Q e_Q ET e_{ET}	• FBC(<e_{IN}, e_Q>) = $e_{IN} \vee (ie_{preIN} = 0 \wedge (ie_{inT} = 0 \vee ie_{inT} \ge e_{PT}))\}$ • FBC(<e_{PT}, e_Q>) = $(ie_{inT} > 0)$	[Timer] TOF, TON, TP [Bistable] RS, SR [Counter] CTD, CTU, CTUD [Edge Detection] F_TRIG, R_TRIG

Fig. 3. Definition of representative function conditions and function block conditions.

Type 2: Input value appears on output edge unchanged only in certain condition. *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. *AND* block is another example. While one might also argue that *AND* belongs to the type 1, we classify it as type 2 because it characterizes behavior of *AND* block more precisely. If e_{IN1} is *true*, same value 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 constraint. Formal definition on FC for the *AND* block with two inputs *IN*1 and *IN*2 is:

$$\text{if } p_1 = \langle e_{IN1}, e_{OUT} \rangle \wedge p_2 = \langle e_{IN2}, e_{OUT} \rangle \wedge e_{OUT} = AND(e_{IN1}, e_{IN2}),$$
$$FC(p_1) = \textit{if } e_{IN1} \textit{ then } e_{IN2} \textit{ else true} = \neg e_{IN1} \vee e_{IN2}$$
$$FC(p_2) = \neg e_{IN2} \vee e_{IN1}$$

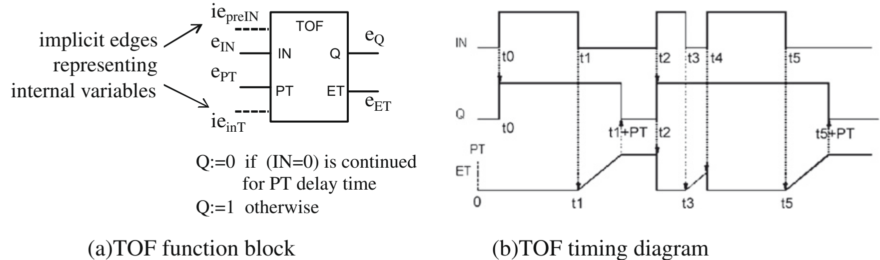
Type 3: Some or all input values are used in the output computation under specific condition. Unlike type 2 functions, output of type 3 functions is not necessarily same as one of the inputs. Rather, some or all inputs are used in determining the output, or output data type is different from that of input data. Functions of comparison group belong to type 3. For example, *LE* function with two integer inputs generates a Boolean output. When the function *LE* has two inputs, all FCs of the *LE* are *true*. Inputs always influence the output. However, when the function *LE* has 3 inputs, FCs are defined as follows:

if $p_1 = \langle e_{IN1}, e_{OUT} \rangle \wedge p_2 = \langle e_{IN2}, e_{OUT} \rangle \wedge p_3 = \langle e_{IN3}, e_{OUT} \rangle \wedge e_{OUT} = LE(e_{IN1}, e_{IN2}, e_{IN3})$,

$$FC(p_1) = \textit{if } (e_{IN1} \leqslant e_{IN2}) \textit{ then } (e_{IN2} \leqslant e_{IN3}) \textit{ else true} = \neg(e_{IN1} \leqslant e_{IN2}) \vee (e_{IN2} \leqslant e_{IN3})$$
$$FC(p_2) = true$$
$$FC(p_3) = \neg(e_{IN2} \leqslant e_{IN3}) \vee ((e_{IN1} \leqslant e_{IN2})$$

Type 4: "Internal" variables as well as inputs must be analyzed to determine the output. $FBC(\langle e_i, e_o \rangle)$ is same as $FB(\langle e_i, e_o \rangle)$ except e_i and e_o is connected by a single "function block". Whereas FC definitions are relatively simple, FBC is more complex due to internal variables as well as input variables. In the proposed technique, internal variables are modeled as function blocks with implicit edges. In this section, we use TOF (Off Delay), shown in Fig. 4(a), as an illustrative example. TOF has two input variables, *IN* and *PT*, and two output variables, *Q* and *ET*. *IN* is a Boolean input, and *PT* indicates delay time. Similarly, *Q* is a Boolean output, and *ET* captures the elapsed time of the internal timer. Semantics of TOF function block is such that it generates the *Q* output *false* when *IN* input remains *false* during the delay time specified by variable *PT* ever since *IN* value changed from *true* to *false*. Otherwise, the output *Q* is *true*. The behavioral definition of timer such as TOF is described by timing diagrams as shown in Fig. 4(b). It shows how outputs *Q* and *ET* vary in response to different *IN* values. As time, labeled 't', passes from left to right, Boolean variables *IN* and *Q* change between *false* and *true*.

To formally define TOF semantics, we use condition and action table as shown in Table 1. The condition is specified in terms of input and internal variables. Likewise, action is an assignment made to output and internal variables. In representing all possible combinations of relevant variables, *preIN* and *inT* denote the value of *IN*



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

Fig. 4. TOF function block and its behavioral definition (a) TOF function block (b) TOF timing diagram.

Table 1
Condition and action table describing behavior of TOF.

Cases	Condition			Action	
	ie_{preIN}	e_{IN}	ie_{inT}	e_Q	ie_{inT}
1	0	0	0	0	Remains stopped
2	0	0	$0 < ie_{inT} < e_{PT}$	1	Continues increasing
3	0	0	$\geqslant e_{PT}$	0	Stops and remains
4	0	1	–	1	Stops and is reset
5	1	0	0	1	Is reset and starts
6	1	1	0	1	Remains stopped
7	1	–	$0 < ie_{inT}$	–	Nonexistent case

stored in the previous scan cycle and internal timer, respectively. *ie* represents an implicit edge as opposed to an explicit edge.

FBCs for the output Q of TOF are evaluated as follows:

if $p_1 = \langle e_{IN}, e_Q \rangle \ \wedge \ p_2 = \langle e_{PT}, e_Q \rangle \ \wedge \ e_Q = TOF_Q(e_{IN}, e_{PT})$,

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

For the $FBC(p_1)$, when e_{IN} is *true*, it flows into the output e_Q without any constraints. If e_{IN} is *false*, output e_Q is also *false* only if $(\neg ie_{preIN} \wedge (ie_{inT} = 0 \vee (ie_{inT} \geqslant e_{PT})))$.

3.3. D-path condition computation

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 FB or FBC is expanded. For example, there are two functions and one function block in the *d*-path $p_{41} = \langle f_X, LE2, SEL4, th_X_Logic_Trip \rangle$ whose DPC is calculated as follows:

Expressions shown in (1) through (5) visually highlight which element is replaced as DPC computation proceeds backward. When backward symbolic computation is completed, DPC should contain expressions containing only the input and internal variables because all the expressions corresponding to the intermediate edges would be replaced. The expression of (5) is transformed into the expression with only input and internal variables by substituting intermediate edge names with expressions from (6)–(9).

$$SEL4 = \neg th_Prev_X_Trip\ ?\ LE3 : LE2 \tag{6}$$
$$LE3 = f_X \leqslant k_X_Trip_Setpoint \tag{7}$$
$$LE2 = f_X \leqslant ADD1 \tag{8}$$
$$ADD1 = k_X_Trip_Setpoint + k_X_Trip_Hys \tag{9}$$

3.4. FBD test coverage criteria

Building on the definition of DPC, we now define three different coverage criteria for FBD programs. They are basic coverage (BC), input condition coverage (ICC), and complex condition coverage (CCC). In addition to formal definition and analysis, we also discuss how these criteria can be used in test case generation so that a suite of test cases may satisfy the chosen criterion.

Definition 3. (**Basic Coverage**) A set of test cases T satisfies the basic coverage criterion if and only if $\forall p \in DP \exists t \in T | DPC(p)|_t = true$. ⌟

Basic coverage (BC) focus 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. As noted earlier, a test case t is "meaningful" if the input of the *d*-path p have 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

$$DPC(p_{41})$$
$$= DPC(\langle f_X, LE2, \underset{\sim}{SEL4, th_X_Logic_Trip} \rangle) \tag{1}$$
$$= DPC(\langle f_X, \underline{\underline{LE2, SEL4}} \rangle) \wedge \underset{\sim}{FBC(\langle SEL4, th_X_Logic_Trip \rangle)} \tag{2}$$
$$= DPC(\langle \underline{f_X, LE2} \rangle \wedge \underline{\underline{FC(\langle LE2, SEL4 \rangle)}} \wedge FBC(\langle SEL4, th_X_Logic_Trip \rangle)) \tag{3}$$
$$= \underline{FC(\langle f_X, LE2 \rangle)} \wedge FC(\langle LE2, SEL4 \rangle) \wedge \underset{\sim}{FBC(\langle SEL4, h_X_Logic_Trip \rangle)} \tag{4}$$
$$= true \wedge th_Prev_X_Trip \wedge \underset{\sim}{(SEL4 \vee (preSEL4 = 0 \wedge (inT5 = 0 \vee inT5 \geq k_Trip_Delay)))} \tag{5}$$

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.

Fig. 5, a partial logic for calculating *th_X_Trip* taken from the KNICS project, clearly illustrates how selected test cases may or may not satisfy certain coverage criteria. As shown in Fig. 6(a), there are 7 DPCs of lengths 3 or 5. If we are to assume that the constant values *k_X_Min* and *k_X_Max* are 2 and 98, respectively, TS1 = {(2, 0, 0, 0)} for the inputs $(f_X, f_Module_Error, f_Channel_Error, th_X_Logic_Trip)$ would achieve 100% BC. In the tables of Fig. 6, we marked 'O' to represent that the test requirement on the column is satisfied at least once by the test cases of the test set on the row.

While basic coverage is straightforward in concept, it is often ineffective in detecting logical errors that FBD program might have. Suppose that the (7) AND_BOOL function had been incorrectly used instead of (7) OR_BOOL block. In such situation, the test set TS1 = {(2, 0, 0, 0)} would still satisfy the basic coverage 100%, and it would be unable to detect such error. What's needed is another coverage in which all the variation of values of Boolean input edge is analyzed.

Definition 4. (**Input Condition Coverage**) A set of test cases 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 ICC, it is no longer sufficient to choose an arbitrary value on all the input edges whose values would influence the outcome (e.g., $DPC(p_{31})$). Fig. 6(b) illustrates that one must now choose a set of test cases such that input values include both *true* and *false* for 3 Boolean inputs (e.g., $DPC(p_{31}) \wedge ME$ as well as $DPC(p_{31}) \wedge \neg ME$). There are 10 test requirements to be satisfied, and TS1 achieves only 70% (or 7 out of 10) coverage with respect to the ICC. TS2, a super set of TS1 including another test case (2,

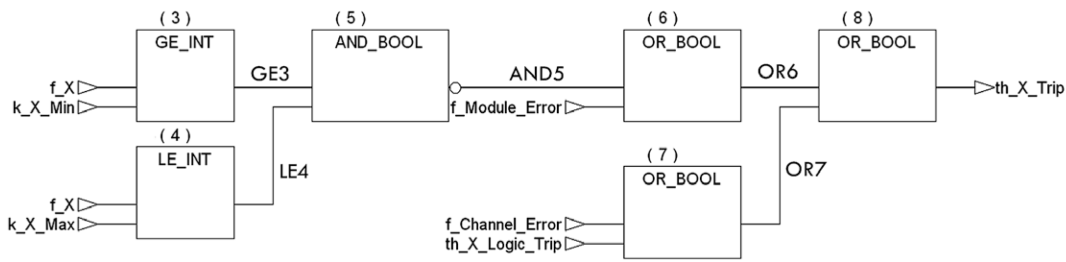


Fig. 5. A simplified FBD program for calculating *th_X_Trip*.

(a) Coverage Result for **Basic Coverage Criterion**

Test Set	DPC(p_{31})	DPC(p_{32})	DPC(p_{33})	DPC(p_{51})	DPC(p_{52})	DPC(p_{53})	DPC(p_{54})	#Test Reqs (Satisfied/ Total)	Coverage (%)
TS1	O	O	O	O	O	O	O	7/7	100%

(b) Coverage Result for **Input Condition Coverage Criterion**

Test Set	DPC(p_{31}) ∧ ME	DPC(p_{31}) ∧ ¬ME	DPC(p_{32}) ∧ CE	DPC(p_{32}) ∧ ¬CE	DPC(p_{33}) ∧ LT	DPC(p_{33}) ∧ ¬LT	DPC (p_{51})	DPC (p_{52})	DPC (p_{53})	DPC (p_{54})	#Test Reqs (Satisfied/ Total)	Coverage (%)
TS1	X	O	X	O	X	O	O	O	O	O	7/10	70%
TS2	O	O	O	O	O	O	O	O	O	O	10/10	100%

(c) Coverage Result for **Complex Condition Coverage Criterion**

Test Set	DPC(p_{31}) ∧ ME	DPC(p_{31}) ∧ ¬ME	DPC(p_{31}) ∧ OR6	DPC(p_{31}) ∧ ¬OR6	DPC(p_{31}) ∧ TR	DPC(p_{31}) ∧ ¬TR	• • •	DPC(p_{54}) ∧ LE4	DPC(p_{54}) ∧ ¬LE4	DPC(p_{54}) ∧ AND5	DPC(p_{54}) ∧ ¬AND5	DPC(p_{54}) ∧ OR6	DPC(p_{54}) ∧ ¬OR6	DPC(p_{54}) ∧ TR	DPC(p_{54}) ∧ ¬TR	#Test Reqs (Satisfied/ Total)	Coverage (%)
TS1	X	O	X	O	X	O		O	X	X	O	X	O	X	O	25/50	50%
TS2	O	O	O	O	O	O		O	X	X	O	X	O	X	O	34/50	68%
TS3	O	O	O	O	O	O		O	O	O	O	O	O	O	O	50/50	100%

For the input vector (f_X, f_Module_Error, f_Channel_Error, f_X_Logic_Trip),
TS1 = { (2, 0, 0, 0) }
TS2 = { (2, 0, 0, 0), (2, 1, 1, 1) }
TS3 = { (2, 0, 0, 0), (2, 1, 1, 1), (0, 0, 1, 1), (99, 0, 1, 1) }

ME: f_Module_Error
CE: f_Channel_Error
LT: th_X_Logic_Trip
TR: th_X_Trip

Fig. 6. Coverage assessment result for the FBD program in Fig. 5.

1, 1, 1), is needed to satisfy 100%. However, TS2 would still fail to detect the error assumed earlier because TS1 and TS2 examine only the (0,0) and (1,1) combinations of *f_Channel_Error* and *th_X_Logic_Trip* variables.

Definition 5. (**Complex Condition Coverage**) A set of test cases 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 $1 \le i \le n$. ⌟

The CCC criterion requires that every Boolean edge's variation in the d-path be tested at least once under the satisfied DPC. For the same FBD programs, as illustrated Fig. 6(c), one must satisfy 50 different test requirements although we chose to explicitly specify only 14 of them due to space limitations. The first six columns show test requirements relevant to $p_{31} = \langle f_Module_Error, OR6, th_X_Trip \rangle$ which has three Boolean edges.

Test set TS3, containing two more test cases, must be included to achieve 100% of the CCC. It is important to note that Fig. 6 shows how different coverage criteria demands more thorough analysis of the FBD logic for a given $DPC(p_{31})$. Whereas there is only one test requirement for BC, the corresponding numbers increase to two and six for ICC and CCC, respectively.

Various analysis, including subsumption relations, can be performed on three different criteria. In software testing research literature, a test coverage criterion A is said to subsume another criterion B iff, for every program P, every test set satisfying A with respect to P also satisfies B with respect to P [13]. Therefore, ICC criterion apparently subsumes BC criterion. If $in(p)$ is not a Boolean type, test requirements for ICC and BC criteria become the same. For a Boolean input type, every test set satisfying $in(p) \wedge DPC(p)$ and $\neg in(p) \wedge DPC(p)$ at least once also satisfies $DPC(p)$ for every d-path p; thus, every test set satisfying ICC criterion also satisfies BC criterion. Similarly, CCC criterion subsumes both ICC and BC criteria. Every test set satisfying $e_i \wedge DPC(p)$ and $\neg e_i \wedge DPC(p)$ at least once, for every d-path p and $1 \leqslant i \leqslant n$, also satisfies $in(p) \wedge DPC(p)$ and $\neg in(p) \wedge DPC(p)$ at least once, because $in(p)$ is a specific case of e_i where i is 1.

4. Case study and evaluation

4.1. KNICS bistable processor trip logic

We applied the proposed technique on two of the 18 trip logics, FIX_RISING and MANUAL_RATE, in the Bistable Processor (BP) design from the KNICS project. Table 2 shows size information and coverage assessment result. MANUAL_RATE module is more complex than FIX_RISING module. According to the unit test result document [17], there were 8 and 19 test cases for each, respectively. If one were to divide 18 trip logics into four groups in terms of complexity, 1 indicating the simplest and 4 the most complex modules, our examples belong to the groups 1 and 3. Therefore, two modules are representative enough of the BP design in terms of size and complexity. However, we made no simplification on the FBD design, and we used test cases prepared by FBD testing professionals in entirety in evaluating adequacy of test cases. It took two skilled FBD engineers working full-time for about 6 weeks each to document FBD testing plan and generate test cases for the whole BP system.

The most striking result of the case study is that test cases derived by domain experts achieved only 90% and 68% of the BC for the two submodules, respectively, although the definition is relatively simple. In fact, when informed on coverage measures, they were surprised that their test cases failed to investigate FBD programs in adequate depth.

Fig. 7 visually demonstrates detailed coverage analysis result applied on the FIX_RISING trip logic. It consists of 26 functions and more than 60 edges, and there are 30 d-paths for a primary output whose length vary from 2 to 9. As it is unnecessary for readers to review the detailed FBD design to appreciate significance of the result, we omit the full FBD design. Eight different test cases, each with 3 inputs, were subject to coverage analysis with respect to BC, ICC, and CCC. Other constant inputs are omitted to keep the table size manageable. Columns at the right part of the tables represent test requirements which grow from 30 for BC to 35 and 240 for ICC and CCC, respectively. Test requirements have different forms (e.g., $DPC(p_{xx}), in(p_{xx}) \wedge DPC(p_{xx}), \neg e_i \wedge DPC(p_{xx})$, etc.) according to selected coverage criteria.

Shaded cells represent the test requirements satisfied by each of 8 test cases. For example, executing test case T1 with respect to BC covers 10 DPCs. As expected, each test case usually satisfies several DPCs. However, there are 3 DPCs that were never covered, and the BC coverage measure is 90% (or 27 out of 30). DPCs that had not been covered are highlighted with thick lines. When the same design and test cases are evaluated using ICC and CCC, coverage measure dropped to about 83% (or 29 out of 35) and 66% (or 158 out of 240), respectively. This case study, convincingly demonstrated that the proposed idea is highly effective in revealing which logical aspects of FBD design remain untested, assessing quality of test cases, and monitoring progress towards meeting the mandated quality goals.

Proposed coverage criteria can also be used generate missing test cases to satisfy uncovered test requirements. We used a SMT solver, Yices [18], to solve uncovered test requirements and found two more test cases T9 = (26805, F, 0) and T10 = (26805, T, 0) for inputs PV, TL, and TC, respectively. Inclusion of two test cases would increase coverage measure to 97% for BC (from 90%), 97% for ICC (from 83%), and 83% for CCC (from 66%), respectively.

4.2. Coverage criteria evaluation

For any test coverage criteria be effective, the following properties must hold: (1) Test coverage criteria definition must be intuitive and properly reflect the computational model; (2) One must be able to formally reason on subsumption relationship among different criteria; (3) One must be able to accurately specify the quality goals, measure current quality of software, and monitor progress in quality assurance activity; and (4) For a given criteria, one must be able to generate test cases necessary to achieve the mandated coverage.

In this paper, we proposed structural test criteria suitable for FBD programs which properly reflect data flow-centric aspect of the computational model. Conventional coverage criteria, based on control flow graph, are clearly inadequate to FBD because any test case would easily achieve 100% of statement or branch coverage. Such test is highly unlikely to detect logical errors in FBD programs. BC, ICC, and CCC criteria are defined based on the formal

Table 2
Submodule information and coverage assessment result.

Submodule name	# Blocks	# Inputs	# d-Paths	# Test cases	BC (%)	ICC (%)	CCC (%)
FIX_RISING	26	10	30	8	90(27/30)	83(29/35)	66(158/240)
MANUAL_RATE	53	23	305	19	68(207/305)	61(229/375)	46(1794/3869)

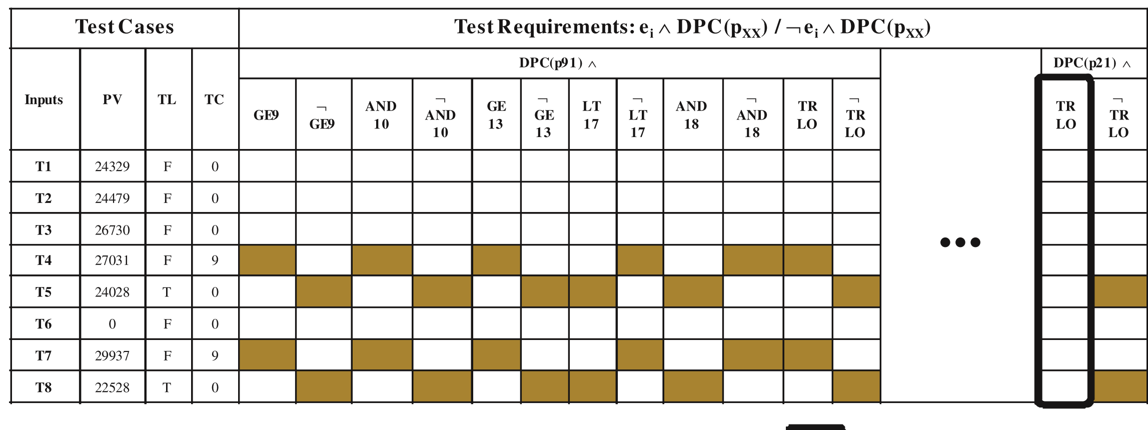
a * Basic Coverage: 90% (27 of 30 test requirements are satisfied at least once)

Test Cases				Test Requirements: DPC(p_{XX})																													
Inputs	PV	TL	TC	p91	p92	p81	p82	p83	p84	p85	p71	p72	p73	p74	p75	p76	p61	p62	p63	p64	p65	p66	p67	p51	p52	p53	p41	p42	p43	p44	p31	p32	p21
T1	24329	F	0																														
T2	24479	F	0																														
T3	26730	F	0																														
T4	27031	F	9																														
T5	24028	T	0																														
T6	0	F	0																														
T7	29937	F	9																														
T8	22528	T	0																														

b * Input Condition Coverage: 83% (29 of 35 test requirements are satisfied at least once)

Test Cases				Test Requirements: in(p_{XX}) ∧ DPC(p_{XX}) / ¬in(p_{XX}) ∧ DPC(p_{XX}) / DPC(p_{XX})																						
Inputs	PV	TL	TC	p91	p92	p81	p82	p83 (T)	p83 (F)	p84	p85	p71 (T)	p71 (F)	p72	p73	p74	p75	p76	p61	p62	p63	p64	•••	p31 (T)	p31 (F)	p32	p21
T1	24329	F	0																								
T2	24479	F	0																								
T3	26730	F	0																								
T4	27031	F	9																								
T5	24028	T	0																								
T6	0	F	0																								
T7	29937	F	9																								
T8	22528	T	0																								

c * Complex Condition Coverage: 66% (158 of 240 test requirements are satisfied at least once)



Test Cases				Test Requirements: e_i ∧ DPC(p_{XX}) / ¬e_i ∧ DPC(p_{XX})														
				DPC(p91) ∧													DPC(p21) ∧	
Inputs	PV	TL	TC	GE9	¬GE9	AND 10	¬AND 10	GE 13	¬GE 13	LT 17	¬LT 17	AND 18	¬AND 18	TR LO	¬TR LO	•••	TR LO	¬TR LO
T1	24329	F	0															
T2	24479	F	0															
T3	26730	F	0															
T4	27031	F	9															
T5	24028	T	0															
T6	0	F	0															
T7	29937	F	9															
T8	22528	T	0															

* PV: PV_OUT, TL: TRIP_LOGIC, TC: TRIP_CNT, TRLO: TRIP_LOGIC_out

Unsatisfied test requirements

Fig. 7. Coverage assessment result for the FIX_RISING submodule.

definition of FBD and *d*-path. We also defined FCs and FBCs for functions and function blocks in IEC61131-1 standard. Our approach, customizing the activation condition concept to FBD, allows focused analysis of data flows from input to output edges. Case study performed on the BP design of the KNICS project, with no simplification introduced, clearly demonstrated that the approach is effective. Our approach revealed many aspects of the FBD design that had been neglected by FBD testing professionals. Proposed coverage criteria, ranging from BC to CCC, provide various levels of rigor in FBD unit testing. In addition to proving subsumption relationship, we also identified conceptual counterpart when compared to classical structural testing coverage criteria. For example, ICC corresponds to the branch coverage in that programs are tested on both true and false branches. Likewise, *d*-path condition is a way of properly reflecting data flow-centric aspect of the FBD computation model which is similar to the data flow coverage criteria such as all-defs. Therefore, proposed structural coverage criteria on FBD programs allow developers and quality assurance personnel communicate precisely and effectively in terms of quality goals and current accomplishments.

Possibility of automated test case generation is another important contribution of our research, and we are developing a coverage measurement tool with capability to generate test cases corresponding to uncovered test requirements. To demonstrate

feasibility, we used a satisfiability module theories (SMT) solver Yices [18] to derive such test cases. If an FBD program includes only functions, test case generation can be automated using SMT solver. In the case of function blocks, such as timer function blocks, proper sequencing of test cases is also important. Such task is a part of planned research.

One must realize that it may be impossible to achieve 100% coverage for any coverage criterion because some *d*-paths may turn out to be infeasible. For example, no test case may cover $DPC(p_{42})$ or $TRIP_LOGIC_out \wedge DPC(p_{21})$. Although infeasible test requirements do not necessarily imply that the FBD program is incorrect, such possibility is high if an SMT solver returns 'unsatisfiable' as result.

D-paths in FBD programs are always finite because FBD programs do not allow internal loops. On the other hand, cyclic and infinite execution, an essential characteristic of the PLC programs, can be considered "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. When measuring the adequacy of test set for FBD programs with function blocks, we should keep track of internal variables as well as input and output variables in the sequence of test cases. When generating test cases satisfying test requirements including internal variables, a test case is not adequate and sequence of test cases is needed.

5. Conclusion

In this paper, we proposed structural test criteria suitable for FBD language which is often used to implement safety-critical systems. While regulation authority requires that safety be rigorously demonstrated through various activities including unit testing, little is known how to rigorously perform structural testing and measure coverage criteria on FBD programs. Existing structural test coverage criteria, based on control flow graph, cannot be directly applied to data flow languages such as FBD. To fill such gap, we focused on how data flows from input to output edges can be analyzed, and three different criteria (BC, ICC, and CCC) have been defined. Proposed structural coverage criteria are found to be intuitive to FBD programmers, and the results of a case study convincingly demonstrated the effectiveness of the proposed technique. When applied on a FBD design obtained from the KNICS project, our technique found which logical aspects of FBD design were untested by test cases prepared by FBD testing professional for over extended period of time. Tool development to automate application of structural testing is currently in progress.

Acknowledgement

This work was partially supported by the MKE (Ministry of Knowledge Economy), Korea, under the ITRC (Information Technology Research Center) support program supervised by the IITA (Institute of Information Technology Advancement) (IITA-2009-(C1090-0902-0032)), and also partially supported by a Korea University Grant and the Korean Research Foundation Grant funded by the Korean Government (KRF-2008-331-D00524).

References

[1] International Standard for Programmable Controllers: Programming Languages Part 3, IEC, 1993.
[2] KNICS (Korea Nuclear Instrumentation and Control System Research and Development Center), <http://www.knics.re.kr/english/eindex.html>.
[3] USNRC (United States Nuclear Regulatory Commission), <http://www.nrc.gov>.
[4] USNRC, Software Unit Testing for Digital Computer Software Used in Safety Systems of Nuclear Power Plants, Regulatory Guide 1.171, September 1997.
[5] S. Richter, J. Wittig, Verification and validation process for safety I&C systems, Nuclear Plant Journal (May–June) (2003) 36–40.
[6] L. Baresi, M. Mauri, A. Monti, M. Pezze, PLCTools: design, formal validation, and code generation for programmable controllers, in: Proceedings of the IEEE International Conference on System, Man, and Cybernetics, Nashville (USA), October 8–11, 2000, pp. 2437–2442.
[7] E. Jee, J. Yoo, S. Cha, Control and data flow testing on function block diagrams, in: Proceedings of the 24th International Conference on Computer Safety, Reliability and Security (SAFECOMP), LNCS 3688, Fredrikstad, Norway, September 28–30, 2005, pp. 67–80.
[8] E. Jee, S. Jeon, H. Bang, S. Cha, J. Yoo, G. Park, K. Kwon, Testing of timer function blocks in FBD, in: Proceedings of the 13th Asia Pacific Software Engineering Conference (APSEC), Bangalore, India, December 6–8, 2006, pp. 243–250.
[9] A. Mader, A classification of PLC models and applications, in: Proceedings of the 5th International Workshop on Discrete Event Systems (WODES 2000), Ghent (Belgium), 2000.
[10] R. Lewis, Programming industrial control systems using IEC 1131-3, IEE Control Engineering Series, Revised ed., The Institute of Electrical Engineers, 1998.
[11] J. Yoo, S. Park, H. Bang, T. Kim, S. Cha, Direct control flow testing on function block diagrams, in: Proceedings of the 6th International Topical Meeting on Nuclear Reactor Thermal Hydraulics, Operations and Safety (NUTHOS-6), October 2004.
[12] H. Zhu, P. Hall, J. May, Software unit test coverage and adequacy, ACM Computing Surveys 29 (1997) 366–427.
[13] M. Pezze, M. Young, Software Testing and Analysis: Process, Principles, and Techniques, Wiley & Sons, 2008.
[14] S. Rapps, E.J. Weyuker, Selecting software test data using data flow information, IEEE Transactions on Software Engineering, SE-11 (4) (1985) 367–375.
[15] A.L. Davis, R.M. Keller, Data Flow Program Graphs, Computer 15 (2) (1982) 26–41.
[16] A. Lakehal, I. Parissis, Structural test coverage criteria for lustre programs, in: Proceedings of the 10th International Workshop on Formal Methods for Industrial Critical Systems (FMICS), Lisboa, Portugal, September 2005.
[17] KNICS-RPS-STR141 (Rev.00) – Software Test Result for the Bistable Processor of the Reactor Protection System, Korea Atomic Energy Research Institute, 2006.
[18] B. Dutertre, L. de Moura, The Yices SMT solver, <http://yices.csl.sri.com/tool-paper.pdf>.