함수 블록 다이어그램으로 명세된 PLC 프로그램에 대한 구조적 테스팅 기법

(A Structural Testing Strategy for PLC Programs Specified by Function Block Diagram)

지 은 경 † 전 승 재 †† 차 성 덕 †††
(Eunkyoung Jee) (Seungjae Jeon) (Sungdeok Cha)

요 약 프로그래머블 로직 컨트롤러(PLC: Programmable Logic Controller)가 안전성이 중요한 실시간 시스템 구현에 많이 사용되면서, PLC 프로그램에 대한 테스팅의 중요성이 날로 높아지고 있다. 본 논문에서는 PLC 프로그래밍 언어 중 하나인 함수 블록 다이어그램(FBD: Function Block Diagram)에 대한 구조적 테스팅 방안을 제안한다. FBD를 테스트하기 위해 먼저 타이머 함수 블록을 비롯한 각 함수 및 함수 블록에 대한 흐름그래프 템플릿을 정의하고, 템플릿을 기반으로 한 변환 알고리즘을 제안하며, 알고리즘을 따라 FBD로부터 변환된 흐름그래프에 기존의 제어 흐름 테스팅 커버리지와 데이타 흐름 테스팅 커버리지를 적용한다. 기존 FBD 테스팅은 테스트 케이스 생성시 FBD 내부 구조를 고려하지 않으며, FBD 프로그램으로부터 특정 중간단계 모델을 생성해 낼 수 있는 경우에만 적용될 수 있는 단점을 가진 반면, 본 논문에 제안된 방법은 FBD 내부 구조를 고려한 체계적 테스트 케이스 생성이 가능하며, 중간단계 모델의 형식에 관계없이 어떤 FBD에도 적용될 수 있다는 장점을 가진다. 특히 제안된 기법은 여러 실행주기에 걸쳐 테스트 되어야 하는 타이머 함수 블록을 포함한 FBD에 대한 철저한 테스팅을 가능하게 한다. 제안된 기법을 현재 원전계측제어시스템 개발사업단에서 개발 중인 디지털 원자로 보호계통 비교논리 프로세서 트립 논리에 적용하여 그 효과를 확인하였다.

키워드 : 프로그래머블 로직 컨트롤러, 함수 블록 다이어그램, 소프트웨어 테스팅, 구조적 테스팅

Abstract As Programmable Logic Controllers (PLCs) are frequently used to implement real-time safety critical software, testing of PLC software is getting more important. We propose a structural testing technique on Function Block Diagram (FBD) which is one of the PLC programming languages. In order to test FBD networks, we define templates for function blocks including timer function blocks and propose an algorithm based on the templates to transform a unit FBD into a flowgraph. We generate test cases by applying existing testing techniques to the generated flowgraph. While the existing FBD testing technique do not consider internal structure of FBD to generate test cases and can be applied only to FBD from which the specific intermediate model can be generated, this approach has advantages of systematic test case generation considering internal structure of FBD and applicability to any FBD without regard to its intermediate format. Especially, the proposed method enables FBD networks including timer function blocks to be tested thoroughly. To demonstrate the effectiveness of the proposed method, we use trip logic of bistable processor of digital nuclear power plant protection systems which is being developed in Korea.

Key words : Programmable Logic Controller, Function Block Diagram, Software Testing, Structural Testing

· 본 연구는 21세기 프론티어 연구개발사업의 일환으로 추진되고 있는 정보통신부의 유비쿼터스컴퓨팅및네트워크원천기반기술개발사업의 지원에 의한 것임

† 학생회원 : 한국과학기술원 전자전산학과
ekjee@dependable.kaist.ac.kr
†† 정 회 원 : 삼성전자 Visual Display 사업부
seungjae.jeon@samsung.com
††† 종신회원 : 고려대학교 정보통신대학 컴퓨터 · 통신공학부 교수
scha@korea.ac.kr

논문접수 : 2007년 4월 11일
심사완료 : 2008년 1월 25일

정보과학회논문지: 소프트웨어 및 응용 제35권 제3호(2008.3)

1. 서 론

프로그래머블 로직 컨트롤러(PLC: Programmable Logic Controller)는 제어 시스템 구현에 널리 쓰이는 산업용 컴퓨터이다[1]. PLC는 여러 분야에 사용되는데 이들 중 대부분은 오류 발생시 인명이나 재산에 심각한 피해를 초래할 수 있는 안전 필수 시스템들이다. PLC가 안전 필수 시스템 구현에 많이 사용되면서 PLC 프로그램 테스팅에 대한 중요성이 날로 높아지고 있다. 특히 원자력 발전소 계측제어 분야에서 기존의 RLL(Relay Ladder Logic) 기반 아날로그 시스템들을 PLC 기반 소프트웨어로 교체해 감에 따라, 원자력 계측제어 분야에서도 PLC 프로그램 테스팅이 중요한 이슈가 되고 있다.

PLC 프로그램은 국제 표준 IEC61131-3[2]에 정의된 5가지 언어들을 이용해 명세 된다. 본 논문은 PLC 프로그래밍 언어 중 가장 널리 사용되는 것 중 하나인 함수 블록 다이어그램(FBD: Function Block Diagram)으로 작성된 프로그램을 대상으로 한다.

FBD로 구현된 어플리케이션은 도구에 의해 자동으로 PLC 기계어 코드로 컴파일 된 후, PLC 위에서 실행된다. 기존 FBD 프로그램 테스팅 방법은 [3], [4]에서와 같이 FBD 프로그램을 블랙박스(black-box)로 보고 입력값을 넣어 결과의 정확성 여부를 확인하는 기능적 테스팅(functional testing)이었다. 그러나, 이들 방법은 FBD 내부 상세 구조를 고려하지 않기 때문에 전형적인 블랙박스 테스팅의 한계를 가지며, FBD를 직접 테스팅 한 것이 아니라 FBD로부터 PLC 실행코드로 변환하는 중간에 생성된 모델을 대상으로 테스팅을 수행한 것이기 때문에, FBD로부터 특정 중간 모델을 생성하지 않는 PLC 도구에서는 적용될 수 없는 단점을 가진다.

본 논문에서는 FBD 프로그램 내부 구조를 고려한 구조적 테스팅 기법을 제안한다. 이 방법은 단위 프로그램이 내부적으로 오류를 어떻게 다루는지 알 수 있고 테스트 케이스를 통해 프로그램 내 경로가 얼마큼 커버되었는지 알 수 있는 구조적 테스팅의 장점을 가진다. 또한 본 논문에서 제시하는 방법은 C코드나 페트리넷(Petri Net) 등 특정 중간 모델 생성 없이도 FBD를 테스팅 할 수 있는 방안이기 때문에, PLC 제조업체들이 FBD를 기계어 코드로 컴파일하는 과정에서 어떤 내부 포맷을 사용하는지에 관계없이 모든 FBD 프로그램에 적용될 수 있다는 장점을 가진다.

FBD 테스팅을 위해서 먼저 FBD 프로그램을 흐름그래프로 변환하는데, 이를 위해 변환의 근간이 되는 각 함수 블록에 대한 템플릿과 템플릿 기반 변환 알고리즘을 제안한다. FBD로부터 흐름그래프가 생성되면 기존의 제어 흐름 테스팅 커버리지와 데이타 흐름 테스팅 커버리지를 흐름그래프에 적용하여 테스트 케이스를 생성한다. 본 논문에서는 특히 테스팅 시 여러 주기에 걸친 상태 정보까지 고려해야 하는 타이머 함수 블록에 대한 흐름그래프 템플릿 생성 방법을 제시하며, 타이머 함수 블록을 포함하는 FBD 프로그램을 시간 요소를 고려하여 체계적으로 테스팅 할 수 있는 방안을 제시한다. 제안된 방법의 효과를 설명하기 위해, 현재 원전계측제어시스템 개발사업단(KNICS)[5]에서 개발중인 디지털 발전소 보호계통(Digital Plant Protection System)의 원자로 보호계통(Reactor Protection System)내에서 중요한 논리를 수행하는 비교논리 프로세서(Bistable Processor)의 트립 논리(Trip Logic) 예제를 사용한다.

본 논문은 다음과 같이 구성된다. 2장에서 FBD와 소프트웨어 테스팅을 간략히 소개하고 관련연구를 소개한다. 3장에서는 함수 및 타이머 함수 블록에 대한 흐름그래프 템플릿을 제안하며, 이를 기반으로 FBD프로그램을 흐름그래프로 변환하는 알고리즘을 제안한다. 4장에서는 변환된 FBD 프로그램에 제어 흐름 및 데이타 흐름 테스팅 커버리지를 적용하는 방법 및 적용결과를 설명하며, 마지막으로 5장에서 결론 및 향후 연구에 대해 언급한다.

2. 연구 배경

2.1 함수 블록 다이어그램(Function Block Diagram)

PLC[1]는 화학 공정 발전소, 원자력 발전소, 교통 제어 시스템 등 제어 시스템에 광범위하게 사용되고 있는 산업용 컴퓨터이다. PLC 프로그래밍 언어는 IEC 61131-3[2] 표준에 다섯 가지가 포함되어 있으며 각각은 Structured Text(ST), Function Block Diagram (FBD), Ladder Diagram(LD), Instruction List(IL), Sequential Function Chart(SFC) 이다. FBD는 PLC 프로그래밍 언어들 중에서 가장 널리 사용되는 것 중 하나로서, 표기 방법이 이해하기 쉽고 제어 블록들간의 데이타 흐름을 잘 표현하는 언어이다.

FBD 프로그램은 시스템 행위를 함수 블록들간 신호의 흐름으로 표현한다[6]. 입력 변수와 출력 변수 사이의 연결은 전기 회로와 같은 형태로 연결된 함수 블록들의 집합으로 표현된다. 함수 블록들은 수행하는 기능에 따라 몇 가지 그룹으로 구분될 수 있다. 그림 1은 주요 함수 블록 그룹과 각 그룹에 속한 대표적인 함수 블록의 예를 보여준다.

그림 2는 FBD 프로그램의 예를 보여준다. 그림 2는 출력변수인 *th_Prev_X_Trip*을 계산하는 단위(unit) FBD 프로그램으로서, 고정설정치 하강 트립 논리의 일부이다. 공정값이 지정된 트립설정치 이하로 내려간 상태로 지정된 시간 이상 경과된 경우, 또는 채널 오류

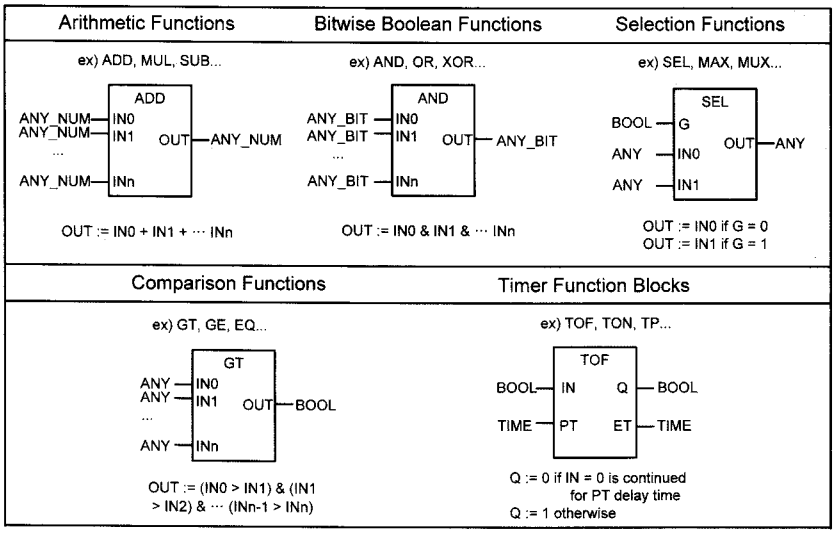


그림 1 FBD 함수 블록 그룹과 그룹 내 대표적인 블록들

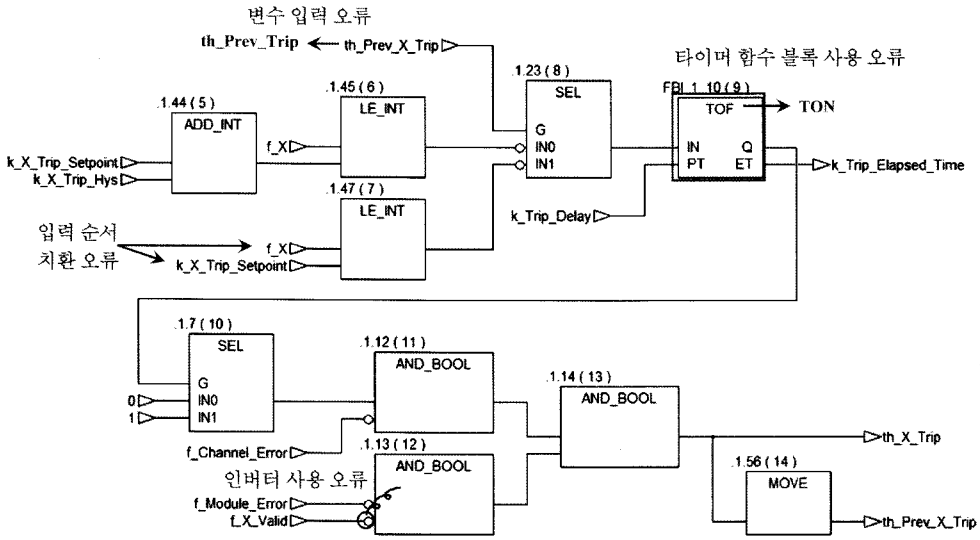


그림 2 삽입된 오류가 표시된 *th_Prev_X_Trip* 에 대한 FBD 단위 프로그램

(*f_Channel_Error*), 모듈 오류(*f_Module_Error*), 공정값 유효성 오류(*f_X_Valid*)가 발생한 경우에 *th_Prev_X_Trip* 변수를 트립 상태를 나타내는 0으로 설정하는 논리이며, 트립이 발생하면 이후 논리에서 원자로를 정지시키는 작업을 수행하게 된다.

FBD에 있는 모든 함수 블록들은 정해진 실행 순서를 가지는데, 이는 FBD의 주요한 특징 중 하나이다. FBD에 있는 각 함수 블록들은 매 스캔 주기(scan cycle)마다 정해진 실행 순서에 따라 순차적으로 수행된다. 함수 블록의 윗 부분 괄호 안에 표시된 숫자는 그 함수 블록의 실행 순서이다. 그림 2에서 1.44(5) ADD_INT 함수는 실행 순서가 (5)이며 이 단위 프로그램에서 가장 먼저 실행된다. 1.56(14) MOVE 함수는 가장 마지막으로 수행된다.

본 논문에 제시된 기법을 설명하기 위해 앞으로 그림 2의 FBD 예제를 사용한다. 제안된 테스팅 기법의 효과를 확인하기 위해 우리는 그림 2의 FBD에 의도적으로 FBD 프로그래밍에서 자주 발생하는 오류들을 삽입하였다. 1.47(7) LE_INT 함수의 두 입력을 치환하였으며, 1.23(8) SEL 함수의 입력 변수를 *th_Prev_X_Trip*에서 *th_Prev_Trip*으로 변경하였다. FBI_1.10(9) TOF 함수 블록 대신 TON 함수 블록을 사용하였으며, 1.13(12)

AND_BOOL 함수의 *f_X_Valid* 입력에 붙은 인버터(inverter)를 생략하였다.

FBD 네트워크는 함수(function)와 함수 블록(function block)을 포함한다. 함수는 내부 상태가 없는 반면, 함수 블록은 내부 상태를 가지며 입출력 변수뿐만 아니라 상태를 나타내는 내부 변수를 가진다[6]. 함수는 같은 입력에 대해 항상 같은 출력을 내지만, 함수 블록은 같은 입력에 대해 내부 상태에 따라 다른 출력을 낼 수 있다. 그림 1에서 Arithmetic, Bitwise Boolean, Selection, Comparison 그룹은 함수 그룹이고, Timer 그룹은 함수 블록 그룹이다.

2.2 소프트웨어 테스팅

소프트웨어 테스팅은 소프트웨어 내에 존재하는 오류를 찾아내기 위한 목적으로, 테스트 케이스를 가지고 소프트웨어를 실행하는 것이다[7]. 소프트웨어의 모든 가능한 행위를 테스트하는 것은 거의 불가능하기 때문에, 소프트웨어 테스팅의 필수 작업 중 하나는 테스트 될 테스트 케이스 집합을 결정하는 일이다.

테스트 케이스를 생성하는 방법은 기본적으로 두 가지 방식이 있다. 한 가지는 기능적 테스팅으로서, 프로그램을 입력 도메인에 있는 값을 출력 도메인에 있는 값에 매핑(mapping) 시키는 함수로 보는 관점에 기반한 것이다. 기능적 테스팅은 보통 블랙박스 테스팅(black-box testing)으로도 불린다. 다른 한가지 방법은 구조적 테스팅(structural testing)으로서, 블랙박스 테스팅과 대비하여 화이트박스 테스팅(white-box testing)이라고도 한다. 구조적 테스팅이 기능적 테스팅과 다른 근본적인 차이점은 블랙박스 내부의 구현 내용이 테스트 케이스를 생성하는데 사용된다는 점이다. 블랙박스 내부를 들여다 봄으로써 테스터는 함수가 실제로 어떻게 구현되어 있는지에 기반하여 테스트 케이스를 생성할 수 있다.

본 논문에서는 FBD에 대한 구조적 테스팅 방법을 제안한다. 구조적 테스팅 기법은 다시 두 부류로 나누어지는데, 소프트웨어 내부의 제어의 흐름에 초점을 둔 제어 흐름 테스팅(control flow testing)과, 각 변수가 정의되고 사용되는 것에 초점을 둔 데이타 흐름 테스팅(data flow testing)이 그것이다. 이 두 가지 테스팅 기법은 모두 필요하며, 상호 보완적으로 조합하여 사용할 수 있다[7]. 본 논문에서는 FBD 프로그램을 흐름그래프로 변환하여 기존의 제어 흐름 테스팅 기법과 데이타 흐름 테스팅 기법을 적용할 수 있는 방안을 제안한다.

2.3 관련 연구

PLC 프로그램에 대한 확인 및 검증을 수행한 관련 연구들은 [8], [9]에 잘 정리되어 있다. PLC 프로그램 중 FBD 언어로 구현된 프로그램을 대상으로 하며, 테스팅을 수행한 사례로는 [3], [4]가 있다. [3]에서는 자체 그래픽 편집기를 사용하여 명세 된 FBD 프로그램을 프로젝트 데이타베이스에 저장한 후, 저장된 FBD 정보로부터 ANSI C 코드를 자동으로 생성하고 이 ANSI C 코드를 대상으로 하여 테스팅을 수행하였으며, 테스팅 수행시 시뮬레이션 기반 확인(validation) 도구인 SIVAT을 활용하였다. [4]에서는 FBD를 하이 레벨 타임드 페트리넷(HLTPN: High Level Timed Petri Nets)으로 변환하고 이에 대해 시뮬레이션 기반 테스팅을 수행하였다. 명세 및 변환, 시뮬레이션 전 과정에 PLCTOOLS 통합 도구를 활용하였다.

[3]에서의 방법과 [4]에서 방법은 모두 자동화 도구를 이용해 테스팅을 지원한다는 것과 시뮬레이션 기능을 제공한다는 장점을 가진다. 그러나 [3]과 [4]에서 사용한 테스팅 방법은 프로그램을 블랙박스로 보고 테스터가 넣은 입력값에 대한 결과의 정확성 여부를 확인하는 기능적 테스팅이다. 기능적 테스팅에서는 FBD 프로그램 내부의 구조적 특징을 고려하지 않기 때문에, FBD 내부의 특정 부분은 집중적으로 테스트되고 다른 부분은 테스트 되지 못하는 문제가 발생하기 쉬우며, 테스팅의 목표설정, 테스팅이 얼마만큼 수행되었는지에 대한 평가를 구체적이고 정량적으로 하기 어렵다는 블랙박스 테스팅의 한계를 가진다. 또한 이들 연구에서는 FBD를 직접 테스팅 한 것이 아니라, [3]에서는 FBD를 PLC 실행코드로 변환하는 중간에 생성한 C코드를 대상으로, [4]에서는 FBD를 C코드 변환하는 중간 단계에 생성한 HLTPN 모델을 대상으로 테스팅을 수행한 것이기 때문에, 각각 해당하는 중간 형태 모델을 생성하지 않는 PLC 도구에서는 적용될 수 없다는 단점을 가진다. 본 논문에서는 이러한 단점들을 개선하기 위하여, FBD 내부 구조를 고려한 구조적 테스팅 기법을 제안하고자 하며, FBD로부터 PLC 실행코드를 생성할 때 중간코드 형식으로 어떤 형식을 쓰는지에 상관없이 일반적인 모든 FBD에 적용될 수 있는 방법을 제안하고자 한다.

3. FBD를 흐름그래프로 변환

3.1 FBD에서 흐름그래프로의 변환 알고리즘

본 논문은 FBD 단위(unit) 프로그램에 대한 테스팅을 다룬다. FBD에서 단위란 '주요 출력값(primary output)을 계산해 내는 의미 있는 함수 블록들의 집합'으로 정의된다[10]. FBD 프로그램에서 주요 출력은 메모리에 저장되었다가 외부 출력으로 나가거나 다른 단위 프로그램의 입력으로 사용된다. 그림 2의 FBD 프로그램은 *th_Prev_X_Trip*이라는 주요 출력값을 계산해 내는 함수 블록들의 집합으로서, 정의에 따라 하나의 단위 프로그램이다.

단위 FBD 프로그램으로부터 흐름그래프로의 변환은

FBD 테스팅에서 가장 기본적이고 중요한 과정이다. 알고리즘 1은 단위 FBD 프로그램으로부터 흐름그래프를 생성하는 알고리즘이다. 단위 FBD 프로그램으로부터 흐름그래프로 변환하는 과정에서 먼저 흐름그래프의 첫 번째 노드를 생성한다. 첫 번째 노드는 단위 FBD 내의 모든 변수들을 읽어 들이는 내용(content)을 가진다. 이렇게 하면 흐름그래프의 첫 번째 노드는 단위 프로그램 내에서 사용된 모든 변수들에 대한 정의(definition)노드가 된다. 첫 번째 노드를 생성한 이후에는 블록의 실행 순서에 따라 각 블록을 흐름그래프 세그먼트(segment)로 변환한다. 각 블록이 어떤 함수 또는 함수 블록이냐에 따라 흐름그래프 세그먼트로 변환되는 형식이 다르기 때문에 각 블록에 대해서 별도의 흐름그래프 세그먼트 생성 함수를 호출한다. 알고리즘 1의 **switch**(fb.type) 문 아래 GenFgSegForADD(fb, curNodeNum, endNodeNum)는 ADD 함수에 대해서, GenFgSegForAND(fb, curNodeNum, endNodeNum)는 AND 함수에 대해서 흐름그래프 세그먼트를 생성하는 함수이다. 알고리즘 1에는 지면 관계상 두 가지 함수에 대해서만 기록하였고, 실제 알고리즘은 IEC 표준 내의 모든 함수 및 함수 블록에 대해서 해당 흐름그래프 세그먼트 생성 함수를 호출한다.

흐름그래프 세그먼트에는 노드 및 에지가 포함되는데, 각 노드에는 그 노드가 가리키는 수행 내용(content)을 명시한다. 이 과정에서 함수 또는 함수 블록의 출력 변수가 지정되어 있지 않은 경우는 임시 출력 변수를 생성해 이름을 붙여준다. 임시 출력 변수의 이름은 *v2*, *v5*와 같이 '*v*[실행순서]' 형태로 정한다.

템플릿에 따라 흐름 그래프 세그먼트가 생성되면, 이를 전체 흐름그래프에 붙인다. 이 과정을 단위 FBD 프로그램 내의 마지막 실행 블록에 이르기까지 반복하여 모든 함수 블록이 변환되면 끝낸다.

알고리즘 1. FBD 단위 프로그램으로부터 흐름 그래프를 생성하는 알고리즘

```
procedure GenFlowgraphFromFBD (
    fbArray : FBArray;          { 입력: FB(함수블록) 배열 }
    startFbNum : integer;       { 입력: 첫번째 실행 FB 의 실행번호 }
    endFbNum : integer;         { 입력: 마지막 실행 FB 의 실행번호 }
    inputVarList : StringList;  { 입력: FBD 단위프로그램 입력 변수들 }
    flowgraph : Flowgraph       { 출력: 생성된 흐름 그래프 } )

var
    flowgraphSeg : Flowgraph;
    fb : FunctionBlock;
    node : Node;
    curNodeNum : integer;
    outVar, contentString : string;

begin { GenFlowgraphFromFBD }
    { 흐름 그래프의 첫번째 노드(노드 0)를 생성 - 첫번째 노드는 FBD 입력 변수들을
    모두 Read 하는 content 를 가진다. }
    curNodeNum := 0;
    node := CreateNode(curNodeNum);
    contentString := MakeContent(READ, inputVarList);
    node.content := contentString;
    InsertNode(flowGraph, node);
    { 첫 실행 FB 부터 마지막 실행 FB 까지 실행 순서에 따라 함수 또는 함수 블록을
    템플릿을 적용해 흐름그래프 세그먼트(segment)로 변환한다. }
    for curFbIndex := startFbNum to endFbNum do
        fb := fbArray[curFbIndex];
        switch(fb.type)
            case ADD: flowgraphSeg = GenFgSegForADD(fb, curNodeNum,
                      endNodeNum);
            case AND: flowgraphSeg = GenFgSegForAND(fb, curNodeNum,
                      endNodeNum);
            ...
            { IEC 표준 내 모든 함수 및 함수 블록에 대해 case 문을 가지고, 각
              함수 및 함수 블록에 대해 흐름 그래프 세그먼트를 생성하는 함수를
              호출한다. }
        end { switch }
        { GenFgSegForXXX 함수 호출을 통해 템플릿에 따라 생성된 흐름그래프
        세그먼트를 전체 흐름그래프에 붙인다. }
```

```
            InsertFlowgraphSeg(flowgraphSeg, flowgraph);
            curNodeNum = outNodeNum + 1;
        end { for curFbIndex }
        output flowgraph;
    end { GenFlowgraphFromFBDs }
```

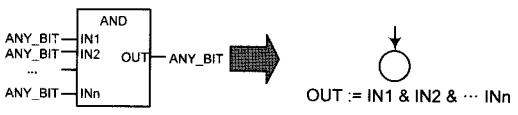
그림 6은 알고리즘 1에 따라 그림 2의 단위 FBD 프로그램을 변환한 흐름그래프이다. 그림 2의 단위 FBD 내 모든 변수들을 읽어 들이는 첫 번째 노드를 생성한 이후, 실행 순서에 따라 각 함수 및 함수 블록들을 해당하는 흐름그래프 세그먼트로 변환하였다. 생성된 흐름그래프 세그먼트는 전체 흐름그래프에 차례로 더해졌다.

3.2 함수에 대한 변환 템플릿

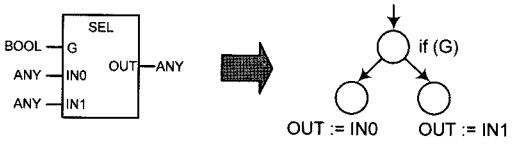
Arithmetic, Bitwise Boolean, Selection, Comparison 그룹에 속한 함수들은 모두 다음의 3 가지 중 하나로 변환된다. 그림 3(a)는 AND함수에 대한 변환 템플릿으로서 Arithmetic, Bitwise Boolean, Comparison 그룹에 속한 함수들은 모두 이와 같이 흐름그래프 상에서 하나의 노드로 변환된다. Selection 그룹에는 SEL, MAX, MIN, LIMIT, MUX 다섯 가지의 함수들이 있다. 그림 3(b)와 3(c)는 각각 SEL과 MUX에 대한 템플릿을 보여준다. Selection 그룹에 있는 다른 함수들 - MAX, MIN, LIMIT - 은 그림 3(a)의 AND와 같이 하나의 노드로 변환된다. 알고리즘 2는 MUX 함수에 대해서 템플릿에 따라 흐름그래프 세그먼트를 생성하는 알고리즘을 보여준다.

3.3 타이머 함수 블록에 대한 변환 템플릿

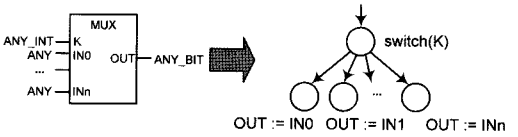
보통 함수들은 한 스캔 주기 만에 테스팅이 가능하지만, 타이머 함수 블록을 비롯하여 내부 상태를 가지는 함수 블록들은 여러 스캔 주기에 걸친 테스팅을 필요로 한다. 내부 상태를 가지는 함수 블록들에 대한 템플릿생성방법을 설명하기 위해 대표적으로 타이머 함수 블록 TOF를 선택하여 설명한다. IEC 61131-3[2]표준에 따라 타이머 그룹에는 TOF, TON, TP 함수 블록이 포함된다. 타이머 그룹에 속한 함수 블록을 흐름그래프로 바꿀 때는 각 타이머 함수 블록의 동작 특성을 고려하여 흐름그래프에 반영해야 한다.



(a) AND 함수에 대한 템플릿



(b) SEL 함수에 대한 템플릿



(c) MUX 함수에 대한 템플릿

그림 3 함수들에 대한 템플릿

그림 4(a)는 TOF(Off Delay) 함수 블록을 보여준다. TOF는 *IN*과 *PT* 두 개의 입력변수와 *Q*와 *ET* 두 개의 출력변수를 가진다. *IN*은 이진 입력 변수이며, *PT*는 지연시간을 지정하는 변수이다. *Q*는 이진 출력 변수이며, *ET*는 TOF 내부 타이머의 경과 시간을 나타낸다. TOF 함수 블록의 출력 *Q*는 입력 *IN*이 1에서 0으로 바뀐 이후 *PT* 시간만큼 0으로 지속되면 그 때 0이 된다. 그 이외의 경우 *Q*의 값은 1이다.

TOF, TON, TP와 같은 타이머 함수 블록의 행위는 IEC61131-3 표준에 타이밍 다이어그램으로 정의되어 있다. 그림 4(b)는 TOF의 행위 정의를 보여 주는 타이밍 다이어그램이다. 그림 4(b)의 다이어그램에서 가로축은 시간의 경과를 나타내며, 세로축은 입력 *IN*의 변화에 따른 *Q*와 *ET* 출력변수의 행위를 보여준다. TON과 TP의 행위도 TOF와 같이 타이밍 다이어그램으로 명시된다.

TOF를 흐름그래프 세그먼트로 변환하기 위해서 먼저 TOF의 행위를 조건(condition)과 동작(action) 테이블로 나타낸다. TOF는 함수 블록이기 때문에 내부 상태를 가진다. 내부 상태는 내부 변수 값의 조합으로 표현될 수 있다. 조건은 입력변수와 내부변수들로 구성되며, 동작은 출력변수와 내부변수에 대한 값 할당이다. TOF의 타이밍 다이어그램으로부터 TOF의 행위에 영향을 주는 두 가지 내부변수를 찾을 수 있었고, 각각을 *preIN*과 *inT*로 명명하였다. *preIN*은 입력변수 *IN*의 이전 스캔 주기 때의 값을 나타내는 변수이고, *inT*는 내부 타이머를 나타내는 변수이다.

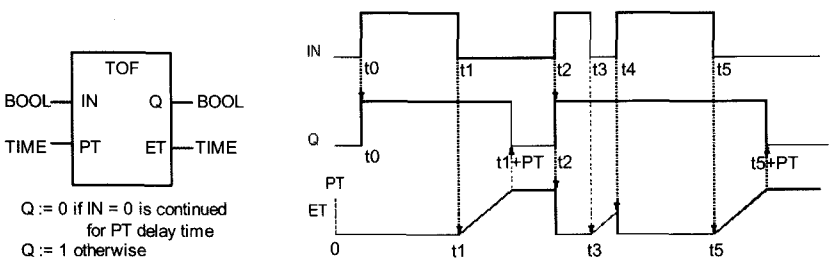
TOF의 모든 행위를 조건과 동작 표로 완전하게 나타내려면 관련된 변수 값의 모든 가능한 조합을 고려해야 한다. 조건에 영향을 주는 변수들은 *preIN*, *IN* 그리고 *inT*이다. *preIN*과 *IN*은 이진 변수이고, *inT* 변수는 무한한 값을 가질 수 있다. *inT*는 조건값에 영향을 주는 관점으로 보았을 때 [0,0], (0,*PT*), [*PT*,∞] 세 개의 동일 클래스(equivalence class)로 구분될 수 있다. 따라

알고리즘 2. MUX 함수에 대해 흐름 그래프 세그먼트를 생성하는 알고리즘

```
procedure GenFgSegForMUX (
        fb: FuntionBlock;          { 입력: 변환 대상 함수 블록 }
        startNodeNum : Integer;    { 입력: 시작 노드 번호 }
        endNodeNum : Integer;      { 출력: flowgraph segment 의 마지막 노드 번호 }
        flowgraphSeg: Flowgraph { 출력: MUX 함수에 대한 흐름그래프 세그먼트 } )

var
        flowgraphSeg : FlowgraphSegment;
        node : Node;
        outVar, contentString : String;
        childNodeList : NodeList;

begin { GenFgSegForMUX }
        { 조건 노드 생성 }
        curNodeNum := startNodeNum;
        node := CreateNode(curNodeNum);
        contentString := MakeContent(IF, fb.condVar);
        node.content := contentString;
        InsertNode(flowgraphSeg, node)
        { 함수 또는 함수 블록의 출력 변수가 지정되어 있지 않은 경우는 임시 출력
        변수를 생성해 이름 붙여준다. 임시 출력 변수의 이름은 v2, v5 와 같이
        'v[실행순서]' 형태로 정한다.}
        outVar := SetOutVariable (fb.executionNo, fb.outputVar);
        {조건에 따라 하위 노드 생성}
        for i=0 to fb.inputNum-1 do
                curNodeNum := curNodeNum + 1;
                node := CreateNode(curNodeNum);
                { 각 노드에 그 노드가 가리키는 수행 내용(content)을 명시한다. }
                contentString := MakeContent(ASSIGN, outVar, fb.in[i]);
                node.content := contentString;
                AddNode(childNodeList, node);
        end { for_i }
        InsertChildrenNodes(flowgraphSeg, childNodeList);
end { GenFgSegForMUX }
```



(a) (b)

그림 4 타이밍 다이어그램으로 표현된 TOF 함수 블록의 행위 정의

서, 세 개의 변수에 대해 12개의 값 조합이 생기게 된다.

표 1은 TOF 행위를 결정짓는데 관여하는 변수들로 만들어진 12가지의 조건 각각에 대한 TOF의 동작을 보여준다. 각 행은 *preIN*, *IN*, *inT* 에 대한 특정 조건이 만족될 때, *Q* 와 *inT* 동작이 어떠한지를 보여준다. 표 1의 맨 오른쪽에 있는 열은 각 케이스가 그림 4(b)의 타이밍 다이어그램에서 어디에 해당하는지를 보여준다. 표 1은 실제는 존재하지 않는 TOF의 행위도 포함하고 있다. b8, b9, b11 그리고 b12가 그 경우이다. 이 네 가지 경우는 결코 발생할 수가 없다. *preIN* 이 1일 때 *inT* 는 항상 0 이기 때문이다.

표 2는 표 1을 동일한 동작에 대한 조건들을 논리적으로 결합하여 축약한 표이다. 우리는 TOF의 출력변수 *Q*의 값과 내부타이머 *inT*의 행위가 같을 때 두 동작이

표 1 TOF의 행위를 묘사하는 조건/동작 표

Cases	조건(Condition)			동작(Action)		그림 4(b)에서 해당하는 부분
	preIN	*IN*	*inT*	*Q*	*inT*	
b1	0	0	0	0	remains stopped	[0,t0)
b2	0	0	*0<inT<PT*	1	continues increasing	(t1,t1+*PT*, (t3,t4), (t5,t5+*PT*)
b3	0	0	*inT>= PT*	0	stops and remains	[t1+*PT*,t2), [t5+*PT*,-)
b4	0	1	0	1	stops and is reset	t0
b5	0	1	*0< inT<PT*	1	stops and is reset	t4
b6	0	1	*inT>=PT*	1	stops and is reset	t2
b7	1	0	0	1	is reset and starts	t1,t3, t5
b8	1	0	*0< inT<PT*	-	존재하지 않는 경우	-
b9	1	0	*inT>=PT*	-	존재하지 않는 경우	-
b10	1	1	0	1	remains stopped	(t0,t1),(t2,t3), (t4,t5)
b11	1	1	*0< inT<PT*	-	존재하지 않는 경우	-
b12	1	1	*inT>=PT*	-	존재하지 않는 경우	-

표 2 TOF의 행위를 묘사하는 축약된 조건/동작 표

Cases	조건(Condition)			동작(Action)		표 1에서 해당하는 케이스	그림 5에서 해당 노드
	preIN	*IN*	*inT*	*Q*	*inT*		
r1	0	0	0	0	remains stopped	b1	10, 12
r2	0	0	*0<inT<PT*	1	continues increasing	b2	1, 3
r3	0	0	*inT>= PT*	0	stops and remains	b3	2, 5
r4	0	1	-	1	stops and is reset	b4, b5, b6	4, 7
r5	1	0	0	1	is reset and starts	b7	6, 9
r6	1	1	0	1	remains stopped	b10	8, 11
r7	1	-	*0< inT*	-	존재하지 않는 경우	b8, b9, b11, b12	-

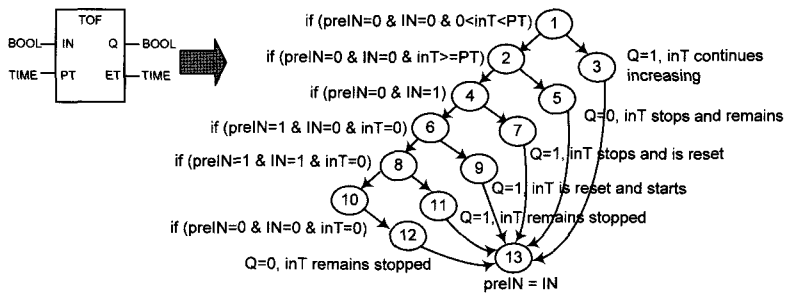


그림 5 TOF함수 블록에 대한 흐름그래프 템플릿

같다고 정의하였으며, 내부타이머 *inT*의 동작을 5가지 경우 - 'remains stopped', 'continues increasing', 'stops and remains', 'stops and is reset', 'is reset and starts' - 로 분류하였다. 표 1에서 b4, b5, b6의 경우 *Q*의 값과 *inT*의 행위가 같으므로 이들은 같은 동작이며, 이 세 경우에 대한 조건은 논리적으로 결합되어 '*preIN* = 0 and *IN* = 1'이 된다. 이는 표 2에서 r4 경우에 해당한다. 동일한 동작에 대한 조건들을 논리적으로 결합함으로, TOF의 행위는 표 2에 나타난 것처럼 7가지 경우로 표현될 수 있다. 이 결과를 가지고 TOF 함수 블록을 흐름그래프로 변환하는 템플릿을 생성한다. 그림 5는 TOF에 대한 템플릿이다. 표 2의 맨 오른쪽 열은 각 경우가 그림 5의 흐름그래프 세그먼트에서 어느 노드에 해당하는지를 나타낸다.

타이머 그룹의 TON과 TP에 대한 템플릿도 비슷한 방법으로 생성될 수 있다. 조건/동작 표를 사용하여 그 함수 블록의 행위를 완전하게 기술한 후 그것을 흐름그래프 템플릿으로 나타낸다. 이 방법은 타이머 함수 블록들뿐만 아니라 다른 함수 블록들에도 동일하게 적용될 수 있다.

3.4 변환 알고리즘에 대한 분석

흐름그래프로의 변환을 위해 FBD 프로그램 내 블록

들은 한 번씩만 방문되며, 각 함수 및 함수 블록에 대한 템플릿은 고정된 개수의 노드 및 에지를 가지기 때문에 본 논문에서 제안한 알고리즘은 O(n)의 시간 복잡도를 가진다.

알고리즘 1의 for 문에서 볼 수 있는 것처럼 FBD 프로그램에서 흐름그래프로 변환할 때의 변환 순서는 이미 정해져 있는 FBD 프로그램 내 블록들의 실행순서를 따른다. FBD 프로그램 내 블록들의 실행순서와 동일한 순서로 흐름그래프 세그먼트가 만들어지고 순차적으로 붙여져 나가기 때문에, 각각 블록을 행위를 정확하게 흐름그래프 세그먼트로 변환한다면, 대상 FBD 프로그램을 수행한 것과 흐름그래프를 따라 수행한 결과가 같다는 것이 보장된다. 따라서, 대상 FBD 프로그램에서 흐름그래프로 오류 없이 변환됨을 확인하기 위해서는 각 함수 및 함수 블록에 대한 흐름그래프 템플릿이 그 행위를 정확히 반영하고 있는지 확인하면 된다.

각 블록에 대한 흐름그래프 템플릿은 IEC61131-3 표준에서 기술하고 있는 각 블록의 기능 및 행위에 대한 정의를 충실히 반영하여 정의하였고, 블록과 그에 대한 흐름그래프 템플릿을 비교해 보았을 때 양쪽 행위의 같음이 직관적으로 확인 가능하다. 출력에 영향을 미치는 입력 및 내부변수 조합에 따라 가지(branch)를 나누고 각 조건 가지마다 출력 변수에 해당값을 할당(assign)하는 노드를 붙이는 원칙을 사용하였는데, 이러한 원칙에 기반하되, 흐름그래프의 복잡도가 불필요하게 커지는 것을 막기 위해 FBD 프로그래밍에서 오류 발생가능성이 거의 없는 부분은 여러 개의 가지로 나누지 않고 하나의 노드로 표현하였다. 예를 들어, SEL 함수의 경우는 입력 G의 조건에 따라서 두 개의 가지로 나뉘어지는 구조로 변환되지만, AND 함수의 경우는 하나의 노드로 변환된다. 이러한 결정은 FBD 프로그래밍에서 자주 발생하는 오류 케이스에 대한 정보와 FBD 프로그래밍에 숙련된 도메인 전문가들의 의견에 기반하여 결정하였다.

함수 블록 변환의 경우는, 입력변수 값뿐만 아니라 내부 상태까지 고려되어야 하는 대상 함수 블록의 복잡한 행위를 완전하고 정확하게 반영하기 위해, 출력값에 영향을 주는 모든 변수들의 가능한 모든 조합을 커버하는 조건/동작 표를 만들고 그 표를 기반으로 템플릿을 정의하였으므로 누락되는 경우가 없음을 확인할 수 있다.

본 논문에서 제안한 변환 템플릿 및 알고리즘의 정확성에 대해서, 현재로서는 FBD 프로그래밍 전문가 및 테스팅 전문가가 검토를 통해 정확성 및 실용성을 확인한 것으로 증명을 대신하였으나, 향후 보다 체계적인 증명으로 뒷받침할 예정이다.

4. FBD에 대한 단위 테스팅

4.1 타이머 함수 블록 테스팅

단위 FBD 프로그램을 흐름그래프로 변환한 후, 적절한 테스트 커버리지 기준(test coverage criteria)을 선정하고 테스트 케이스를 생성한다. 그림 1의 다섯 가지 함수 블록 그룹들 중에서 타이머 이외 그룹에 속한 것들은 입력 값이 같으면 항상 같은 출력을 내는 내부 상태가 없는 함수들이다. FBD가 함수들로만 구성되었을 경우, 입력 값들이 결정되면 한 스캔 주기 안에 출력값이 결정되게 되므로 한 스캔 주기 만의 테스팅으로도 충분하다. 그러나, FBD가 타이머 함수 블록들을 포함하게 되면 입력 값이 같더라도 내부상태에 따라 출력 값이 달라지므로, 여러 주기에 걸쳐 테스팅 하는 것이 필요하다.

타이머 함수 블록들을 포함한 FBD를 충분히 테스트하기 위해서는 입력 변수들과 타이머 함수 블록들의 내부 상태 조합을 가능한 많이 커버할 수 있는 테스트 케이스를 생성해야 한다. 위에서 제안한 템플릿에 기반하여 생성된 흐름그래프에 All-Edge 커버리지 기준을 적용하면 타이머 함수 블록들의 모든 구별되는 내부 상태를 최소 한 번 이상 커버하는 테스트 케이스를 생성할 수 있다. All-Edge 커버리지 보다 더 강력한 커버리지 기준을 적용하면 보다 정교한 테스트 케이스 집합을 얻을 수 있다.

함수들로만 이루어진 FBD 테스팅에 비해 타이머 함수 블록을 포함한 FBD 테스팅의 가장 구별되는 특징은 각 테스트 케이스에 대해 전제 조건(precondition)을 명시해야 한다는 것이다. 전제 조건은 타이머 함수 블록들의 내부 변수들의 값 조합이다. 테스트 케이스에 전제 조건이 붙으면, 그 전제 조건이 만족하는 상태까지 이르기 위한 테스트 스텁(test stub)이 필요하다.

4.2 제어 흐름 테스팅(Control Flow Testing)

제어 흐름을 테스팅 하기 위한 커버리지는 All-Nodes, All-Edges, All-Paths 등이 있다. 표 3은 그림 6의 흐름그래프에 대해 All-Edges 테스트 커버리지 기준을 만족시키는 테스트 케이스들이다. All-Edges 테스트 커버리지 기준을 만족하려면 흐름그래프 상의 모든 에지(edge)가 최소 한 번 이상 실행되어야 한다.

표 3에서 *Prev_Trip*, *Prev_X_Trip*, *X*, *Ch_Err*, *Md_Err*, *Valid*는 각각 *th_Prev_Trip*, *th_Prev_X_Trip*, *f_X*, *f_Channel_Error*, *f_Module_Error*, *f_X_Valid* 변수를 나타내며 이 여섯 개의 열은 단위 프로그램에 대한 입력 변수들이다. 오른쪽의 마지막 두 열 중 Actual은 이들 입력을 가지고 프로그램을 실행시켰을 때의 결과 *th_Prev_X_Trip* 변수의 값이며, Expected는 오류 없이 정상적으로 실행되었을 경우 *th_Prev_X_Trip* 변수가 가져야 할 기대값이다. 실제 실행 결과값이 기대값

표 3 All-Edges 테스트 커버리지 기준을 만족시키는 테스트 케이스 집합

테스트 케이스	Precondition		Inputs						*th_Prev_X_Trip*	
	pre_v8	*inT9*	*Prev_Trip*	*Prev_X_Trip*	*X*	*Ch_Err*	*Md_Err*	*Valid*	Actual	Expected
CT1	0	0	1	0	100	0	0	1	0	1
CT2	0	0	0	0	100	0	0	1	0	1
CT3	1	50	0	0	100	0	0	1	0	1
CT4	1	100	0	0	100	0	0	0	0	1
CT5	1	100	1	1	80	0	0	1	0	1

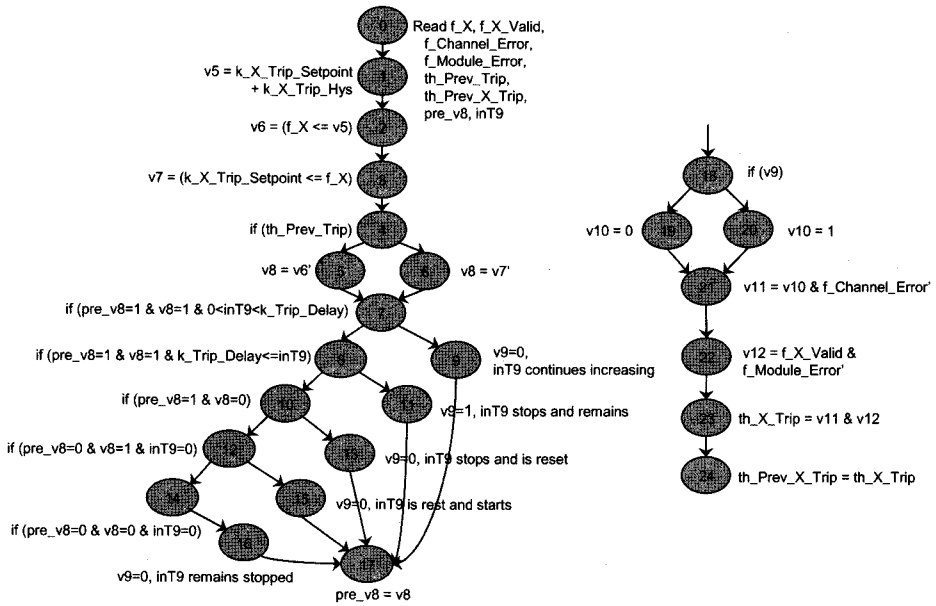


그림 6 오류를 포함한 *th_Prev_X_Trip* 단위 FBD 프로그램에 대한 흐름그래프

과 다른 경우 오류가 있음을 알 수 있다.

테스트 케이스 CT1은 변수 *v8*의 이전 주기값, *pre_v8*이 0이고 내부타이머 *inT9*가 0인 전제조건이 만족된 상태에서, *th_Prev_Trip, th_Prev_X_Trip, f_X, f_Channel_Error, f_Module_Error, f_X_Valid* 입력변수의 값이 각각 1, 0, 100, 0, 0, 1일 때, 테스팅 수행 결과로 나온 *th_Prev_X_Trip*의 결과값은 0이며, 원래 나와야 할 기대값은 1이라는 것을 의미한다. 실제 수행 결과값이 기대값과 다르기 때문에 이 경우 프로그램상에 오류가 존재함을 알 수 있고, 이 테스트 케이스에 대한 분석을 통해 오류를 발견할 수 있다. 예제로 사용된 단위 프로그램은 타이머 함수 블록을 포함하고 있기 때문에 테스트 케이스 각각에 전제조건이 붙어 있다.

4.3 데이타 흐름 테스팅(Data flow testing)

데이타 흐름 테스팅은 변수가 정의되고 사용된 지점에 초점을 맞춘 구조적 테스팅의 한 형태이다[7]. 흐름그래프에 데이타 흐름 테스팅 기법을 적용하기 위해서는, 사용된 모든 변수들에 대한 정의(definition) 노드와 사용(usage) 노드를 확인한 후 각 변수에 대한 du-path들을 확인하는 것이 필요하다. 그리고 나서, 흐름그래프에 대해 All-Defs, All-Uses와 같은 데이타 흐름 테스트 커버리지 기준을 적용한다. FBD 프로그램에 대해 데이타 흐름 테스팅 수행할 때는 FBD의 특성이 올바르게 반영되도록 다음과 같은 사항들을 고려하여야 한다.

첫째로, 기존 FBD에는 없지만 흐름그래프로 변환하는 과정에서 생성된 임시 변수들도 입출력 변수들과 같게 다루어야 한다. FBD를 변환하여 생성한 흐름그래프에는 기존 FBD상에 명시된 입출력 변수들뿐만 아니라, 기존 FBD 상에는 나타나지 않지만 흐름그래프로 변환하는 과정에서 생성된 *v[number]* 형태의 임시 출력 변수들 및 타이머 함수 블록에 관계되어 생성된 내부 변수들이 존재한다. FBD 데이타 흐름 테스팅에서는 이들 변수를 모두 프로그램 내 변수로 고려하여, 각 변수에 대한 정의 노드, 사용 노드, du-path 정보를 추출해야 의미 있는 테스트 케이스를 얻을 수 있다. 예를 들어, 그림 6의 흐름그래프에서 *f_X* 나 *f_Channel_Error* 와

같은 기존 입출력 변수에 대해서뿐만 아니라, *v5*부터 *v12*까지의 임시 출력 변수들, *pre_v8*이나 *inT9*과 같은 타이머 함수 블록 관련 내부 변수들에 대해서도 du-path 정보를 추출하여 데이타 흐름 테스팅을 수행하여야 한다.

둘째로, 흐름그래프의 첫 번째 노드는 단위 FBD 프로그램 내에 사용 중인 모든 변수에 대한 정의 노드로 삼는다. 단위 FBD 프로그램은 전체 FBD 프로그램의 일부이기 때문에, 현재 테스팅 수행 중인 단위 FBD 프로그램 내에서 사용된 변수의 정의가 현재 테스트하는 단위 FBD 밖에 있을 수 있다. 이 경우, 테스팅을 통해 '정의되지 않은 변수가 사용된' 데이타 오류가 발견될 수 있는데, 이러한 오류는 실제로 유효한 오류는 아니다. 이러한 불필요한 오류를 막기 위해서, 단위 FBD를 변환한 흐름그래프의 첫 번째 노드는 흐름그래프에 사용된 모든 변수를 읽어 들이는 정의 노드로 만들었다. 임시 출력 변수들은 예외인데, 임시 출력 변수들은 흐름그래프 상에 항상 정의 노드를 가지기 때문에 첫 번째 노드에서 반드시 정의해 줄 필요는 없다. 그림 6의 흐름그래프에서 첫 번째 노드 0을 보면 임시 출력 변수들을 제외한 모든 변수들이 정의되어 있는 것을 볼 수 있다.

셋째로, 타이머 함수 블록이 포함된 경우는 테스트 케이스의 실행 순서가 중요하다. 표 4는 그림 6의 흐름그래프에 존재하는 모든 변수들에 대한 du-path 정보를 바탕으로 생성한, All-Uses 테스트 커버리지 기준을 만족하는 테스트 케이스 집합이다. *pre_v8*나 *inT9*와 같은 내부 변수의 경우 정의 노드와 사용 노드간의 du-path가 한 스캔 주기 내의 테스팅으로 커버되지 않은 경우가 발생한다. 하나의 테스트 케이스를 수행하면 FBD의 내부 상태, 즉 내부 변수의 값이 바뀌게 되는데, 이렇게 바뀐 상태에서 다음 주기 때 특정 입력을 넣어야 커버되는 du-path가 있는 경우, 연속적인 두 주기 이상의 테스팅이 필요하게 된다. 이 때 이들 테스트 케이스는 서로 의존적인 관계가 되어 수행 순서가 중요해 진다. 표 4에서 DT1과 DT2는 순서대로 실행되어야 한다. DT3와 DT4, DT4와 DT5, DT6와 DT7, DT9와 DT10 간에도 순서가 유지되어야 한다.

4.4 사례 연구

우리는 본 논문에 제안한 방법을 현재 원전계측제어 시스템 개발사업단에서 개발중인 디지털 원자로 보호 시스템의 비교논리 프로세서 트립 논리 일부에 적용하였다. 제안된 방법의 효과를 확인하기 위해서 그림 2의 *th_Prev_X_Trip* FBD 단위 프로그램에 네 가지 오류를 삽입하였고, 이 오류들이 테스팅을 통해 발견되는지를 확인하였다. 이 오류들은 모두 FBD 프로그래밍을 할 때 자주 발생되는 것들이다. FBD 프로그래밍 시 발생할 수 있는 다양한 오류들에 대한 설명과 분류는 [11]을 참조한다. 심겨진 오류들은, 제어 흐름 테스팅에서 All-Edges 커버리지를 만족하도록 만든 표 3의 테스트 케이스들에 의해서 모두 발견될 수 있었으며, 데이타 흐름 테스팅에서 All-Uses 커버리지를 만족하도록 만든 표 4의 테스트 케이스들을 통해서도 모두 발견될 수 있었다.

- **오류 케이스 1 (타이머 함수 블록 사용 오류)**: 타이머 함수 블록 사용시 기능이 비슷한 TON과 TOF간에 혼동하여 잘못 사용하는 경우가 종종 발생한다. TOF를 써야 정확한 부분에 TON을 쓴 오류를 삽입하였는데, 이 오류는 제어 흐름 테스팅에서 표 3의 CT2와 CT3테스트 케이스에 의해서, 데이타 흐름 테스팅에서 표 4의 DT3, DT4, DT6, DT7테스트 케이스에 의해서 발견되었다.
- **오류 케이스 2 (입력 순서 치환 오류)**: FBD 프로그래밍을 하다 보면 종종 입력 변수들의 순서를 뒤바꿔어 기술하는 실수가 발생한다. AND_BOOL같은 함수는 입력들의 순서가 바뀌어도 관계가 없지만, SEL, MUX, GE와 같은 함수에서는 입력들의 순서가 바뀔

표 4 All-Uses 테스트 커버리지 기준을 만족시키는 테스트 케이스 집합

테스트 케이스	Precondition		Inputs						*th_Prev_X_Trip*	
	pre_v8	*inT9*	*Prev_Trip*	*Prev_X_Trip*	*X*	*Ch_Err*	*Md_Err*	*Valid*	Actual	Expected
DT1	0	0	0	0	91	0	0	1	0	0
DT2	0	0	1	0	100	0	0	1	0	1
DT3	0	0	0	0	100	0	0	1	0	1
DT4	1	50	0	0	80	0	0	1	0	1
DT5	0	0	0	0	80	1	1	1	0	0
DT6	1	50	0	0	100	0	0	1	0	1
DT7	1	100	0	0	80	0	0	1	0	1
DT8	1	100	0	0	100	0	0	0	0	1
DT9	1	100	0	0	100	0	0	1	1	1
DT10	0	100	1	1	100	0	0	1	0	1

경우 의미가 반대가 되거나 연산이 반대로 일어나 중대한 오류를 발생시킬 수 있다. 그림 2의 FBD에서 실행번호 (7)번 LE_INT함수의 두 입력 *f_X*와 *k_X_Trip_Setpoint* 의 순서를 바꾸어 쓴 오류를 삽입하여 테스팅을 수행한 결과, 이 오류는 제어 흐름 테스팅의 CT5, 데이타 흐름 테스팅의 DT10 테스트 케이스에 의해 발견되었다.

- **오류 케이스 3 (인버터 사용 오류)**: 작은 원으로 표시되는 인버터는 불필요한 곳에 더해지거나, 필요한 곳에 생략되는 일이 자주 발생한다. 우리는 그림 2의 실행순서 (12)번 AND_BOOL 함수의 IN1입력인 *f_X_Valid*에 붙어 있어야 할 인버터를 생략하였다. 이 오류는 제어 흐름 테스팅의 CT4, 데이타 흐름 테스팅의 DT8 테스트 케이스에 의해서 발견되었다.
- **오류 케이스 4 (변수 입력 오류)**: 입력 또는 출력 변수 이름을 잘못 쓰게 되면, 함수 블록에 잘못된 값이 할당되거나 엉뚱한 변수에 출력값이 저장되게 된다. 변수의 개수가 많은 어플리케이션의 경우 특히 비슷한 이름의 변수를 잘못 쓰는 오류를 범하기 쉽다. 그림 2의 FBD에서 실행순서 (8) 번 SEL 함수 블록의 *th_Prev_X_Trip* 입력을 *th_Prev_Trip*으로 잘못 입력한 오류를 삽입하였는데, 이 오류는 제어 흐름 테스팅의 CT1, 데이타 흐름 테스팅의 DT2 테스트 케이스에 의해서 발견되었다.

5. 결 론

본 논문에서는 PLC 프로그래밍 언어 중 하나인 FBD 프로그램에 대한 구조적 테스팅 방안을 제안하였다. 단위 FBD 프로그램을 테스팅 하기 위해서는 먼저 FBD를 흐름그래프로 변환하는데, 이를 위해 각 함수 및 함수 블록을 흐름그래프로 변환하는 템플릿과 템플릿 기반 변환 알고리즘을 제안하였다. 내부 상태를 가지는 타이머 함수 블록에 대해 변환 템플릿을 생성하는 방법을 제안하였으며, 기존의 구조적 테스팅 기법들을 적용할 때 타이머 함수 블록의 특징을 어떻게 반영해야 하는지 보였다.

FBD 프로그램을 흐름그래프로 변환하고 나면, 흐름그래프를 대상으로 기존에 존재하는 소프트웨어 테스팅 기법들을 효과적으로 적용할 수 있다. 본 논문에서는 FBD 프로그램으로부터 변환된 흐름그래프에 제어 흐름 테스팅 기법과 데이타 흐름 테스팅 기법을 모두 적용하였으며, 기존의 테스팅 방법들을 FBD 프로그램에 적용할 때 고려할 사항들을 명시하였다.

제안된 기법의 효과를 설명하기 위해서, 현재 원전계측제어시스템 개발사업단에서 개발중인 디지털 원자로 보호 시스템의 비교논리 프로세서 트립 논리 예제를 사용하였다. FBD 프로그래밍에서 자주 발생하는 오류들을 FBD 프로그램에 삽입하였고, 제안된 기법들을 적용해서 얻은 테스트 케이스들에 의해 이 오류들이 모두 발견될 수 있음을 확인하였다.

기존에는 FBD 프로그램으로부터 별도의 중간 모델 또는 코드를 생성해서 그에 대해 기능적 테스팅을 수행하는 방법만 있었고, FBD 내부 구조를 고려한 체계적인 구조적 테스팅 기법은 없었다. 본 논문에서 제안한 방법은 FBD를 흐름그래프로 변환함을 통해 FBD 내부 구조를 고려한 구조적 테스팅을 가능하게 했다. 또한, 기존 테스팅 방법들이 특정 중간 모델 또는 코드를 생성하지 않는 FBD 프로그램에는 적용될 수 없었던 것에 비해, 제안된 방법은 중간 모델 형식에 관계없이 일반적으로 모든 FBD에 적용될 수 있다는 장점을 가진다.

향후 연구에서는, FBD 단위 프로그램들이 모두 테스팅 된 상태에서, 단위들 간의 인터페이스나 상호작용들을 테스팅하는 FBD 결합 테스팅 방안에 대한 연구가 필요하며, FBD 테스팅을 지원해 줄 수 있는 효과적인 도구 개발에 대한 연구가 필요하다.

참 고 문 헌

[1] A. Mader, "A Classification of PLC Models and Applications," *Proc. WODES 2000: 5th Workshop on Discrete Event Systems*, Gent, Belgium, Aug. 21-23, 2000.

[2] IEC, International Standard for Programmable Controllers: Programming Languages (Part 3), 1993.

[3] S. Richter and J. Wittig, "Verification and validation process for safety I&C systems," *Nuclear Plant Journal*, pp.36-40, May-June 2003.

[4] L. Baresi, M. Mauri, A. Monti, and M. Pezzè, "PLCTOOLS: Design, Formal Validation, and Code Generation for Programmable Controllers," *Formal methods in PLC programming Special Session at IEEE Conference on Systems, Man and Cybernetics (SMC'2000)*, Nashville, USA, Oct. 8-11, 2000.

[5] KNICS, Korea Nuclear Instrumentation and Control System Research and Development Center, http://www.knics.re.kr

[6] R. Lewis. "Programming industrial control systems using IEC 1131-3 Revised Edition (IEE Control Engineering Series)," *The Institute of Electrical Engineers*, 1998.

[7] Paul C. Jorgensen, "Software testing: a craftsman's approach," *CRC Press*, 1995.

[8] M. Bani Younis and G. Frey, Formalization of Existing PLC Programs: A Survey, *Proceedings of CESA 2003*, Lille, France, July, 2003.

[9] G. Frey and L. Litz, Formal Methods in PLC Programming, *Proceedings of the IEEE Conference on Systems, Man and Cybernetics (SMC'2000)*,

Nashville, USA, Oct. 8-11, 2000.
[10] J. Yoo, S. Park, H. Bang, T. Kim and S. Cha, "Direct Control Flow Testing on Function Block Diagrams," *The 6th International Topical Meeting on Nuclear Reactor Thermal Hydraulics, Operations and Safety (NUTHOS-6)*, Nara, JAPAN, Oct. 4-8, 2004.
[11] Y. Oh, J. Yoo, S. Cha and H. Son, "Software Safety Analysis of Function Block Diagrams using Fault Trees," *Reliability Engineering and System Safety*, Vol.88, No.3, pp. 215-228, 2005.

지 은 경

1999년 KAIST 전자전산학과 전산학전공 학사. 2001년 KAIST 전자전산학과 전산학전공 석사. 2001년~2002년 몽골 울란바타르대학 전임강사. 2003년~2004년 (주)이마린로직스 소프트웨어 엔지니어. 2004년~현재 KAIST 전자전산학과 전산학전공 박사과정. 관심분야는 소프트웨어공학, 정형검증, 소프트웨어 테스팅, 소프트웨어 안전성

전 승 재

2005년 KAIST 전자전산학과 전산학전공 학사. 2007년 KAIST 전자전산학과 전산학전공 석사. 2007년~현재 삼성전자 디지털영상연구소 소프트웨어 엔지니어 관심분야는 소프트웨어공학, 정형검증, 소프트웨어 개발 프로세스

차 성 덕

1983년 University of California, Irvine 전산학 학사. 1986년 University of California, Irvine 전산학 석사. 1991년 University of California, Irvine 전산학 박사. 1990년~1991년 Hughes Aircraft Company, Ground Systems Group 연구원. 1991년~1994년 The Aerospace Corporation 연구원 1994년~2008년 KAIST 전자전산학과 전산학전공 교수. 2008년~현재 고려대학교 정보통신대학 컴퓨터·통신공학부 교수. 관심분야는 소프트웨어공학, 정형기법, 정보보호, 침입탐지