Function Block Diagrams에 대한 제어 및 데이터 흐름 테스팅*

지은경, 유준범, 박수현, 차성덕

한국과학기술원 전자전산학과 전산학전공
대전 유성구 구성동 373-1 한국과학기술원
{ekjee, jbyoo, suhyun, cha}@dependable.kaist.ac.kr

요약 : 본 논문에서는 FBD(Function Block Diagram) 프로그램으로부터 중간코드를 생성하는 과정을 거치지 않고 직접 FBD 프로그램을 테스팅하는 방안을 제안한다. 기존에 PLC 기반 소프트웨어에 대한 테스팅은 FBD 프로그램으로부터 테스팅 목적으로 C 와 같은 중간 코드를 생성하고 이를 대상으로 테스팅을 수행하였다. 본 논문에서는 중간코드를 생성하지 않고 직접 FBD 를 테스팅함으로 테스팅 비용이 절감되는 장점을 가진 기법을 제안한다. 이를 위해 먼저 FBD 프로그램의 관점에서 단위 및 결합테스팅의 개념을 정의하고, FBD 프로그램을 흐름 그래프로 변환하는 알고리즘을 제안한다. FBD 로부터 흐름 그래프를 생성하고 나면, 흐름 그래프를 대상으로 기존의 제어 흐름 테스팅 커버리지와 데이터 흐름 테스팅 커버리지를 적용하여 테스트 케이스들을 생성한다. 제안된 기법의 효과를 설명하기 위해서, 현재 KNICS[3]에서 개발중인 디지털 발전소 원자로 보호계통 중 BP(Bistable Processor) 시스템 일부의 트립 논리 예제를 사용한다.

핵심어: 테스팅, Function Block Diagram, 제어 흐름 테스팅, 데이터 흐름 테스팅, Programmable Logic Controller

1. 서론

소프트웨어 테스팅은 소프트웨어 내에 존재하는 오류를 찾아내기 위한 목적으로, 테스트 케이스 입력을 가지고 소프트웨어를 실행하는 것으로서[1], 소프트웨어 개발시 소프트웨어 품질 보증을 위한 필수적인 프로세스로 여겨진다. 인공위성 제어 시스템, 원자력 발전소 제어 시스템과 같이 고장이 발생할 경우, 인명이나 재산에 큰 손실을 일으킬 수 있는 안전성이 중요한 시스템에서는 테스팅의 역할이 더욱 중요해진다.

원자력 발전소 제어 시스템 분야에서 기존의 RLL(Relay Ladder Logic)을 기반으로 한 아날로그 시스템을 소프트웨어에 의해 제어되는 디지털 시스템으로 교체해 감에 따라, 원자력 발전소 디지털 제어 소프트웨어에 대한 테스팅이 중요해지고 있다. 이 제어 소프트웨어는 안전성이 중요한 실시간 소프트웨어 구현을 위해 널리 사용되는 산업용 컴퓨터인 PLC(Programmable Logic Controller) [2] 위에 구현되는데, 구현 언어로는 LD(Ladder Diagram)나 FBD(Function Block Diagram) 와 같은 PLC 프로그래밍 언어가 사용된다. 안전성이 중요한 시스템 제어에 많이 사용되는 PLC 프로그램을 테스팅하기 위해서는 이들 구현 언어의 특성을 이해하고 정확하면서도 효율적인 테스팅을 수행해야 한 한다. 본 논문은 PLC 프로그래밍 언어 중 가장 널리 사용되는 것 중 하나인 FBD 를 대상으로 한다.

FBD 와 같은 PLC 프로그래밍 언어로 구현된 PLC 소프트웨어는 도구에 의해 자동적으로 PLC 기계어 코드로 컴파일된 후 실행된다. 최종 산출물인 PLC 기계어 코드에 대해 테스팅을 적용하는 것은 너무 복잡해서 거의 수행하기가 어렵고, 설계 단계에 해당하는 FBD 프로그램을 테스팅하는 방안은 없었기 때문에, 기존의 PLC 프로그램 테스팅은 FBD 로부터 PLC 기계어 코드로 컴파일 하는 과정에서 변환된 C 프로그램을 대상으로 이루어졌다. 이 방법은 FBD 테스팅을 어느 정도 수행할 수는 있지만, FBD 테스팅을 위해서 별도로 C 프로그램과 같은 중간 코드를 추출해야 하는 부담과 추가 비용의 문제를 가진다.

본 논문에서는 PLC 프로그래밍 언어로 널리 쓰이는 FBD 로 작성된 프로그램에 대해서 중간 코드 생성 없이 직접 테스팅하는 방안을 제안한다. FBD 프로그램으로부터 PLC 기계어 코드로의 변환 과정은 지난 몇십년간 PLC 개발업체들과 사용자들에 의해 꾸준히 검증되어 왔기 때문에 오류가 없다고 가정하고, 본 논문의 범위에 포함하지 않는다.

FBD 테스팅을 위해 본 논문에서는 먼저 FBD 프로그램 기반 소프트웨어 테스팅의 단위를 정의한다. 여러 함수 블록들(function blocks)의 네트워크로 구성되어 있는 FBD 프로그램의 특성을 고려하여 함수 블록 네트워크의 관점에서 테스팅의 단위 및 모듈을 정의한다. 본 논문에서는 FBD 에 대한 단위 테스팅 프로세스를 중점적으로 다루는데,

* This work was supported by the Korea Science and Engineering Foundation(KOSEF) through the Advanced Information Technology Research Center(AITrc). This work was supported by the Information Technology Research Center(ITRC).

이를 위해서 FBD 프로그램을 흐름 그래프(flow graph)의 형태로 변환하는 알고리즘을 제안한다. FBD 로부터 흐름 그래프가 생성되면, 기존의 제어 흐름 테스팅 커버리지와 데이터 흐름 테스팅 커버리지를 흐름 그래프에 적용한다. 본 논문에서 제안된 방법은, FBD 기반 직접 테스팅을 수행하여, FBD 로부터 별도의 중간 코드를 생성할 필요가 없기 때문에 테스팅 비용이 절감되는 장점을 가진다. 제안된 방법의 효과를 설명하기 위해, 현재 KNICS[3]에서 개발중인 디지털 발전소 보호계통(Digital Plant Protection System)의 원자로 보호계통(Reactor Protection System)중 중요한 논리를 수행하는 BP(Bistable Processor : 비교논리 프로세서)의 트립 논리 예제를 사용한다.

본 논문은 다음과 같이 구성된다. 2 장에서 FBD 와 소프트웨어 테스팅을 간략하게 소개하며, 3 장에서는 FBD 테스팅의 단위를 정의한다. FBD 프로그램을 흐름 그래프로 변환하는 알고리즘을 4 장에 제안하고, 5 장에서는 변환된 FBD 프로그램에 제어 흐름 및 데이터 흐름 테스팅 커버리지를 적용한다. 마지막으로 6 장에서 결론 및 향후 연구에 대해 언급한다.

2. 연구 배경

2.1 Function Block Diagram

PLC(Programmable Logic Controller)[2]는 화학 공정 발전소, 원자력 발전소, 교통 제어 시스템 등 광범위한 제어 시스템에 사용되고 있는 산업용 컴퓨터이다. PLC 는 프로세서와 메인 메모리, 공통 버스로 연결된 입력 모듈과 출력 모듈을 포함하는 통합시스템이다.

PLC 프로그래밍 언어는 IEC 61131-3 [4] 표준에 다섯 가지가 포함되어 있는데, Structured Text(ST), Function Block Diagram(FBD), Ladder Diagram(LD), Instruction List(IL), Sequential Function Chart(SFC) 이다. FBD 는 표기 방법이 이해하기 쉽고 제어 블록들간의 데이터 흐름을 잘 표현하는 장점으로 인해 PLC 프로그래밍 언어들 중에서 가장 널리 사용되는 것 중 하나이다.

FBD 프로그램은 시스템 행위를 함수 블록들간 신호의 흐름으로 표현한다. 입력 변수와 출력 변수 사이의 함수는 전기 회로도와 같은 형태로 연결된 함수 블록들의 집합으로 표현된다. 직사각형 모양으로 표시되는 함수 블록은 왼쪽은 입력변수와 오른쪽은 출력변수와 연결된다. 함수 블록들은 수행하는 기능에 따라 몇 가지 그룹으로 구분된다.

그림 1 은 몇 가지 함수 블록 그룹과 각 그룹에 속한 함수 블록의 예제를 보여준다. KNICS[3]에서 개발중인 원자로 보호 시스템(RPS : Reactor Protection System)은 그림 1 에 있는 5 개 그룹에 속한 함수 블록들로 프로그램 된다.

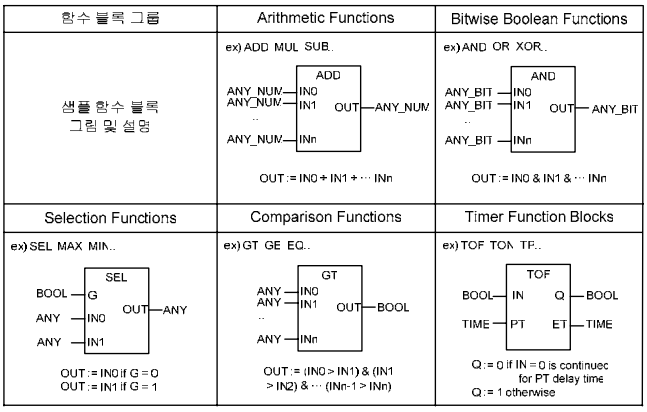


그림 1. FBD 함수 블록들의 대표적인 예

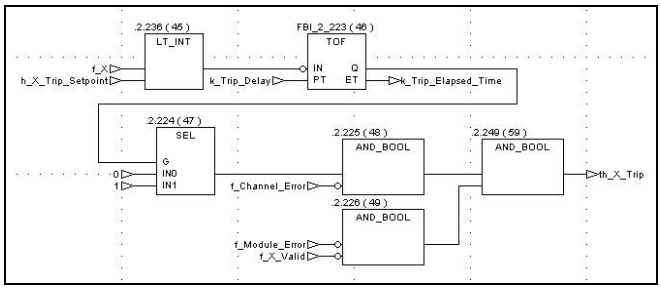


그림 2. FBD 프로그램 예

그림 2 는 함수 블록 네트워크의 예를 보여준다. 최종 출력값 *th_X_Trip* 은 여러 개의 함수 블록 실행의 조합으로 생성된다. 제일 왼쪽 위의 LT_INT 함수는 입력값으로 *f_X* 와 *h_X_Trip_Setpoint* 를 받아들여서 *f_X* 가 *h_X_Trip_Setpoint* 보다 작으면 1 을, 아니면 0 을 내보낸다. 이 결과값은 반전되어 TOF 함수 블록의 입력으로 주어지는데, 이전 함수 블록의 반전된 결과값과 지연 시간을 나타내는 *k_Trip_Delay* 를 입력으로 TOF 함수가 수행되고, TOF 의 출력 Q 는 다음 함수 블록 SEL 의 입력으로 들어간다. SEL 함수는 G 입력으로 들어온 값이 0 이면 IN0 입력을 선택하고, 1 이면 IN1 입력을 선택해서 출력으로 내보낸다. 마지막으로, AND_BOOL 함수를 통해 SEL 함수의 출력값과 *f_Channel_Error*, *f_Module_Error*, *f_X_Valid* 의 값을 각각 반전 시킨 값들을 모두 AND 한 것이 최종적으로 *th_X_Trip* 의 값이 된다.

2.2 소프트웨어 테스팅

소프트웨어 테스팅은 소프트웨어 내에 존재하는 오류를 찾아내기 위한 목적으로, 테스트 케이스를 가지고 소프트웨어를 실행하는 것이다[1]. 소프트웨어 테스팅의 수행결과로 테스터는 소프트웨어 내에 존재하는 오류를 발견하거나 또는

대상 소프트웨어가 테스트 케이스가 나타내는 부분에 대해서는 정확히 동작한다는 것을 확인할 수 있다.

시스템이 처할 수 있는 모든 경우에 대해서 테스팅을 수행하는 것은 불가능하기 때문에 테스팅에서는 일반적으로 대표적인 테스트 케이스들을 선별해서 테스팅을 수행한다. 각 테스트 케이스에 대해서는 입력 집합과 예상 출력 결과가 명시되어야 하며, 테스팅은 테스트 케이스에 명시된 입력을 가지고 소프트웨어를 실행시켜서 나온 결과가 예상 출력 결과와 일치하는지를 비교함으로 진행된다. 가능한 많은 테스트 케이스를 가지고 실행해 보면 좋겠지만 시간과 비용의 문제가 따르기 때문에, 적은 개수로도 소프트웨어 내에 존재하는 다양한 오류들을 가능한 많이 찾아낼 수 있는 테스트 케이스를 설계하는 것이 중요하다. 어떤 테스트 케이스를 가지고 테스팅 하느냐가 테스팅의 효율성을 결정한다고 할 수 있다.

테스트 케이스를 만드는 방법은 기본적으로 두 가지 방식이 있다. 한 가지는 기능적 테스팅(functional testing)으로서, 프로그램을 입력 도메인에 있는 값을 출력 도메인에 있는 값에 매핑(mapping) 시키는 함수로 보는 관점에 기반한 것이다. 시스템을 내부가 보이지 않는 블랙박스로 보고 입력과 출력으로 블랙박스의 기능을 이해하는 방식으로, 블랙박스 테스팅(Black Box Testing)이라고도 한다. 다른 한가지 방법은 구조적 테스팅 (structural testing)으로서, 테스터가 테스트 케이스를 만들 때, 소프트웨어 내부에서 기능이 실제로 어떻게 구현되어 있는가에 기반해서 만드는 방식이다. 소프트웨어의 내부를 보고 필요한 정보들을 사용하기 때문에, 블랙박스 테스팅과 대비하여 화이트 박스 테스팅(White Box Testing)이라고도 한다. 기능적 테스팅은 소프트웨어의 명세만을 기반으로 테스트 케이스들을 만드는 반면, 구조적 테스팅은 소프트웨어 명세뿐만 아니라 실제 구현 코드에 기반해서 테스트 케이스들을 만든다는 점에서 차이가 있다.

본 논문에서는 구조적 테스팅 방식을 적용한다. 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 데이터 흐름 테스팅(data flow testing)이 그것이다. 이 두 가지 테스팅 기법은 모두 필요하며, 상호보완적으로 조합하여 사용할 수 있다[1]. 본 논문에서는 FBD 프로그램을 흐름 그래프로 변환해서 기존의 제어 흐름 테스팅 기법과 데이터 흐름 테스팅 기법을 적용할 수 있는 방안을 제안한다. 제어 흐름 테스팅에서는 All-Nodes, All-Edges, All-Paths 테스트 커버리지들을 적용하며, 데이터 흐름 테스팅에서는 All-Defs, All-Uses, All-DU-Paths 커버리지들을 적용한다.

3. FBD 테스팅의 단위 구분

FBD 는 함수 블록들의 네트워크로 표현된다. 기존의 절차적 프로그래밍 언어로 작성된 프로그램 테스팅에서 단위(unit)와 모듈(module)을 정의했던 것을 그대로 FBD 프로그램에 적용할 수 없기 때문에, FBD 프로그램에서 테스팅의 단위와 모듈이 어디까지인지 명확히 정의하는 것이 필요하다.

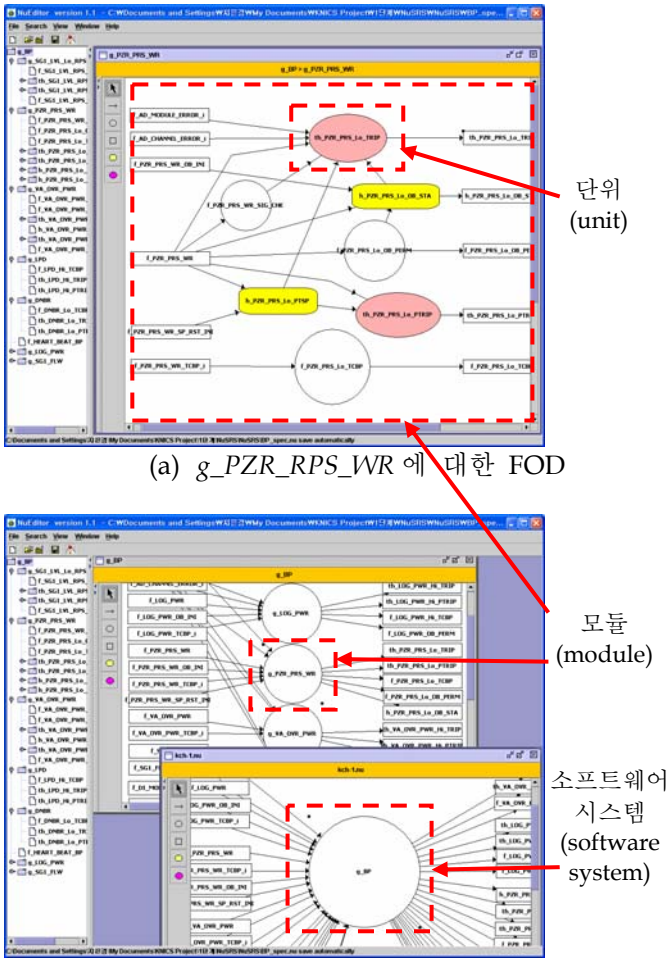
FBD 테스팅에서 단위를 함수 블록 하나로 정의한다면, 앞서 하나의 함수 블록은 항상 정확하게 동작한다고 가정했기 때문에, 단위 테스팅은 필요가 없어진다. 반면, 하나의 단위를 여러 함수 블록들로 정의한다면 한 단위 내에서 변수들간의 상호작용에 대한 것을 고려해야 되기 때문에 결합 테스팅(integration testing)의 문제들을 다루게 된다. FBD 프로그램에서 단위 테스팅을 수행하는 영역에 대한 적합한 구분이 필요하다.

우리는 FBD 프로그램의 단위를 '주요 출력값을 계산해 내는 의미 있는 함수 블록 집합'으로 정의한다. FBD 프로그램에서 주요 출력(primary output)은 메모리에 저장되었다가 외부 출력이나 다른 단위 프로그램의 입력으로 사용된다. FBD 에서 출력으로 표시된 것들 중에서, 단순히 프로그래밍 편의를 위해서 일시적으로 쓰인 출력 변수는 주요 출력에 포함되지 않는다. 그림 2 의 FBD 프로그램은 *th_X_Trip* 이라는 주요 출력값을 계산해 내는 함수 블록 집합으로서, 하나의 단위 프로그램으로 여길 수 있다.

FBD 프로그램의 모듈은 '의미 있는 기능을 수행하는 단위들의 집합'으로 정의한다. KNICS RPS BP 에서 각 트립 논리 블록은 하나의 모듈로 정의될 수 있으며, 각 모듈은 여러 개의 단위들로 구성된다.

KNICS 에서는 소프트웨어 요구명세를 기술하기 위해서 NuSCR[5] 정형 명세 언어를 사용한다. NuSCR 은 안전성과 정확성이 중시되는 원자력 분야에 맞게 특화된 정형명세 언어로서, FOD(Function Overview Diagram), FSM(Finite State Machine), TTS(Timed Transition System), SDT(Structured Decision Table) 표기를 통합적으로 사용해서 시스템을 명세한다. NuSCR 로 기술된 소프트웨어 명세를 보면 FBD 프로그램에서의 단위, 모듈을 구분하는 지침을 얻을 수 있다.

그림 3(a)는 KNICS RPS BP 의 일부분인 가압기 압력 트립 모듈 *g_PZR_PRS_WR* 을, NuSCR 명세 지원 도구인 NuEditor[6]를 사용해서 표현한 것이다. 그림 3(a)는 FOD(Function Overview Diagram)이다. FOD 는 데이터 흐름 다이어그램(data-flow diagram)과 비슷한 형태의 표기를 사용하여, 여러 다양한 노드들간의 관계를 표현하는 다이어그램으로서, 상위



(a) *g_PZR_RPS_WR* 에 대한 FOD

(b) *g_BP* 에 대한 FOD

그림 3. KNICS RPS BP 에 대한 FOD 들

FOD 에 표현된 각 노드에 다시 하위 FOD 를 그리는 방식으로, 복잡한 시스템을 계층적 구조로 체계적으로 표현할 수 있다.

NuSCR 로 기술된 시스템 명세는 설계단계에서 FBD 프로그램으로 구현되는데, 앞서 정의한 FBD 의 단위, 모듈 정의를 따르면, 그림 3(a)에 표현된 FOD 의 각 노드는 FBD 에서 각각 개별적인 '단위(unit)'가 된다. 또한, 그림 3(a)의 전체 FOD 는 FBD 에서 하나의 '모듈'에 해당한다. 즉, 6 개의 입력을 받아들여 5 개의 출력을 내는, 7 개의 단위로 구성된 하나의 모듈이 되는 것이다. 그림 3(a)의 *g_PZR_PRS_WR* 에 대한 FOD 의 상위 FOD 는 그림 3(b)에서 윗부분에 표현된 *g_BP* 에 대한 FOD 이다. 그림 3(b)의 아래쪽을 보면 *g_BP* 라는 하나의 노드와 여러 개의 외부 입력, 출력들을 가지는 FOD 를 볼 수 있는데, 이는 그림 3(b)의 윗부분에 표현된 FOD 의 상위 FOD 로서 이 시스템의 최상위 FOD 이다.

FBD 프로그램에서 소프트웨어 시스템은 전체 모듈들의 집합으로 정의할 수 있다. 소프트웨어 시스템은 시스템의 외부로부터 입력을 받아들이고 시스템의 외부로 출력을 내보낸다. 그림 3(b) 아래쪽에 표시된 최상위 노드인 *g_BP* 는 FBD 에서 '소프트웨어 시스템' 정의에 해당한다. 왼쪽에 직사각형 내부에 표시되어 나열된 것들은 외부 입력 변수들이고, 오른쪽에 나열된 것들은 외부 출력 변수들이다.

이제까지 FBD 기반 테스팅을 위해서, FBD 프로그램에서의 단위, 모듈, 소프트웨어 시스템에 대한 정의를 하였다. FBD 프로그램에서 단위, 모듈 등의 구분은 시스템을 NuSCR 명세 언어로 표기했을 때 더욱 분명하게 구분되어 나타나며, 자연어 명세로 표현되었을 때에도 그 구분은 쉽게 이루어질 수 있다.

4. FBD 로부터 흐름 그래프(flow graph)의 생성

FBD 프로그램 테스팅을 위해서 본 논문에서는 FBD 단위 프로그램을 흐름 그래프로 변환하는 알고리즘을 제안한다. 흐름 그래프를 대상으로 제어 흐름 테스팅 또는 데이터 흐름 테스팅을 수행하는 기법들[7,8,9]은 기존에 제안되었기 때문에, FBD 프로그램으로부터 흐름 그래프를 생성할 수 있으면 기존의 테스팅 기법들을 적용하여 테스팅을 수행할 수 있다. 따라서, FBD 프로그램을 흐름그래프로 변환하는 것은 FBD 테스팅을 위한 가장 기본적이고 중요한 과정이라 할 수 있다.

예제로 사용할 그림 4 는 *th_X_Pretrip* 변수를 계산하는 FBD 단위 프로그램이다. 이 단위 FBD 는 그림 3 에서 보여진 *g_PZR_PRS_WR* 모듈의 일부로서, 예비 트립 설정치 값을 받아들여서 예비 트립 값을 결정하는 기능을 한다.

FBD 프로그램을 흐름 그래프로 변환하기 위해서는 먼저 FBD 프로그램의 실행 특성을 이해해야 한다. FBD 에 있는 모든 함수 블록들은 모두 정해진 실행 순서를 가지는데, 이는 FBD 의 주요한 특징 중 하나이다. FBD 프로그램의 함수 블록들은 매번 스캔 주기 마다 실행 순서에 따라 순차적으로 실행된다. 그림 4 에서 각 함수 블록 위에 쓰여진 괄호에 적힌 숫자는 그 함수 블록의 실행 순서를 나타낸다. 예를 들어, 그림 4 에서는 실행순서 번호가 가장 작은 숫자인 (11)번 AND_BOOL 함수 블록이 가장 먼저 실행되고, (24)번 MOVE 함수 블록이 가장 마지막에 실행된다. 출력인 *th_X_Pretrip* 값은 마지막 함수 블록이 실행되기 전에 출력된다. 변환된 흐름 그래프는 이렇게 FBD 가 순차적 실행 순서를 가진다는 특성을 충분히 반영해야 한다.

그림 5 는 변환된 흐름 그래프이다. 흐름 그래프는 노드와 화살표로 그려진다. 각 노드의 안쪽에 표시된 숫자는 노드의 이름이다. FBD 프로그램에 대한 흐름

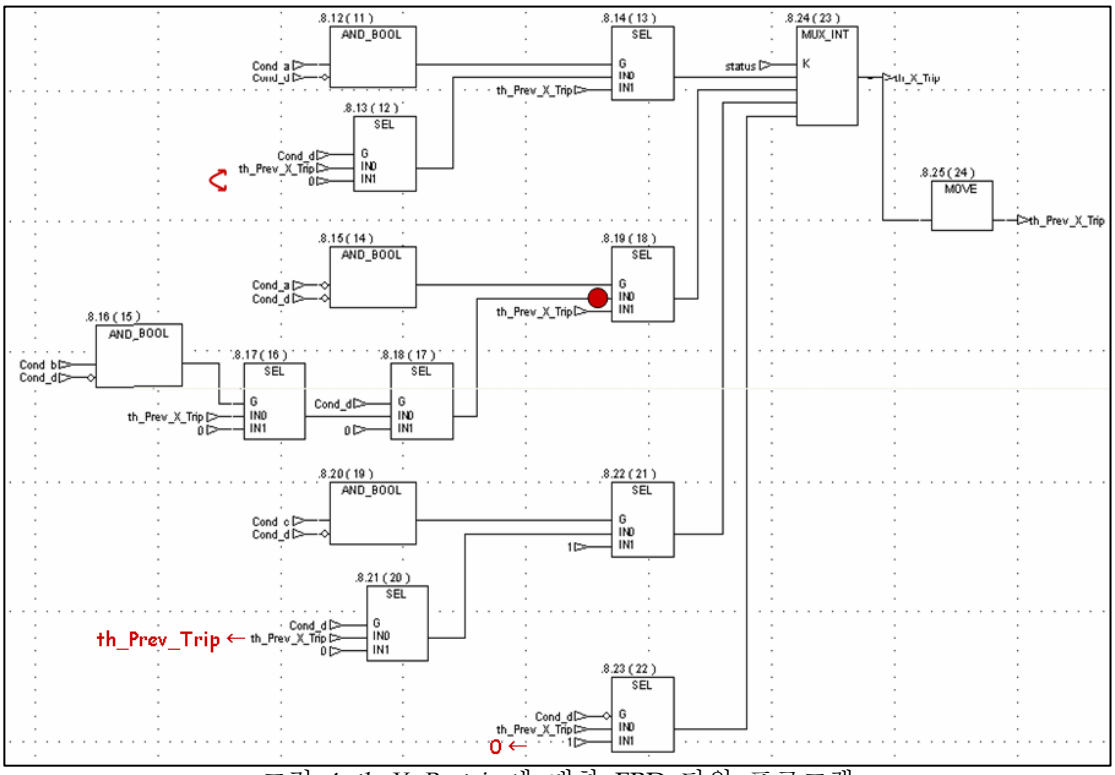


그림 4. *th_X_Pretrip* 에 대한 FBD 단위 프로그램

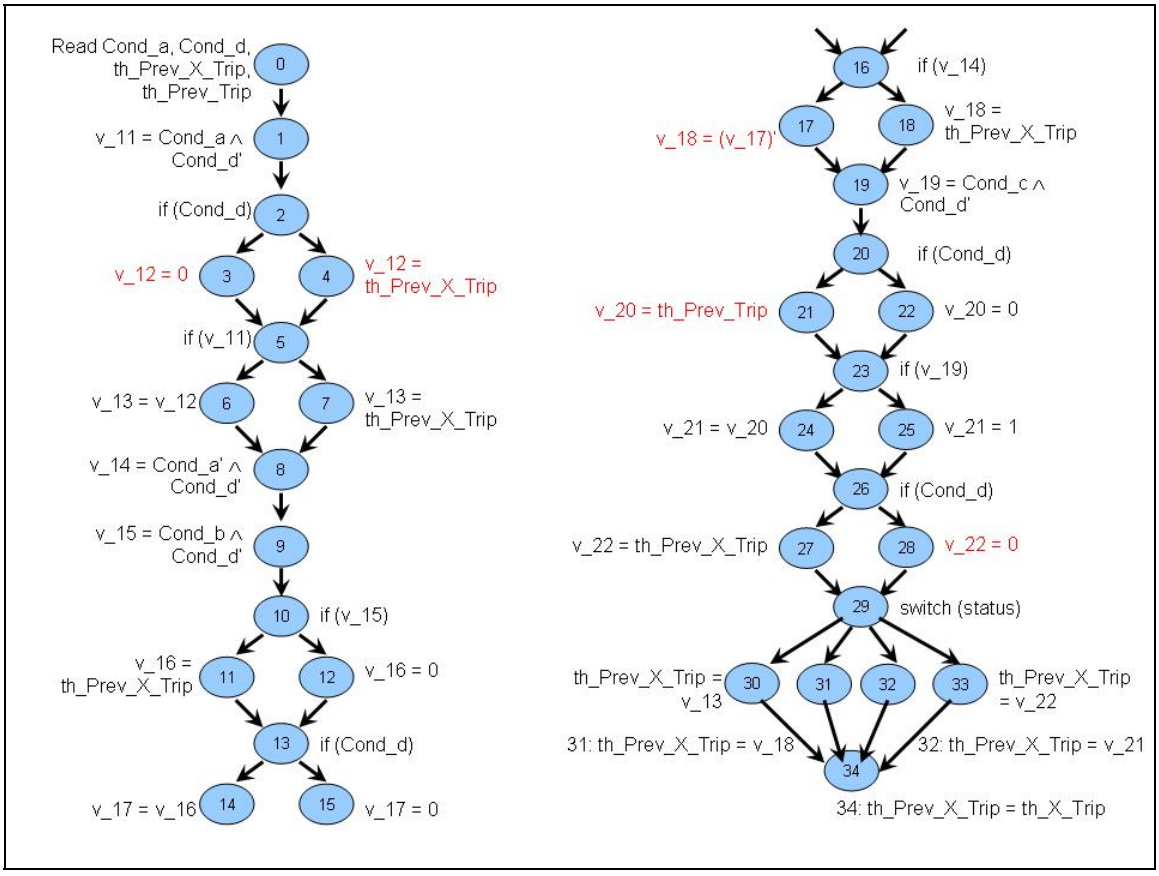


그림 5. *th_X_Pretrip* FBD 단위 프로그램에 대한 흐름그래프

그래프를 만드는 과정은 알고리즘 1 에서 설명한다. 이 알고리즘은 FBD 함수 블록들을 모두 다루지는 않으며, Arithmetic, Bitwise Boolean, Comparison, Selection 그룹의 함수 블록들만으로 이루어진 FBD 프로그램을 흐름 그래프로 변환한다.

알고리즘 1 . FBD 단위 프로그램으로부터 흐름 그래프를 생성하는 알고리즘

```
1:  procedure GenFlowGraphFromFBDs (
2:          fbArray : FBArray;
                { FB(함수블록) 배열}
3:          startFbNum : Integer;
                {첫번째 실행 FB 의 실행번호}
4:          endFbNum : Integer;
                {마지막 실행 FB 의 실행번호}
5:          inputVarList : StringList;
                { FBD 단위프로그램 입력 변수들} )

6:  var
7:      flowGraph : FlowGraph;
8:      fb : FunctionBlock;
9:      node : Node;
10:     childNodeList : NodeList;
11:     curNodeNum : Integer;
12:     outVar, contentString : String;

13: begin { GenFlowGraphFromFBDs }
14:     {흐름 그래프의 첫번째 노드(노드 0)를 생성,
        첫번째 노드는 FBD 입력 변수들을 모두 Read 하는
        content 를 가짐}
15:     curNodeNum := 0;
16:     node := CreateNode(curNodeNum);
17:     contentString := MakeContent(READ, inputVarList);
18:     node.content := contentString;
19:     InsertNode(flowGraph, node);

20:     {첫 실행 FB 부터 마지막 실행 FB 까지 각각 flow
        graph 의 노드(들)로 변환}
21:     for curFbIndex := startFbNum to endFbNum
22:         fb := fbArray[curFbIndex];
23:         switch(fb.group)
24:           case ARITHMETIC :
25:           case BITWISE_BOOLEAN :
26:           case COMPARISON :
27:             { Arithmetic, Bitwise Boolean, Comparison 그룹에
                속한 FB 들은 fb.outputSpec 의 내용을
                outVar 에 할당(assign) 하는 content 를 가진
                하나의 노드를 생성하고 flow graph 에 추가}
28:             curNodeNum := curNodeNum + 1;
29:              node := CreateNode(curNodeNum);
30:              outVar := SetOutVariable( fb.executionNo,
                                        fb.outputVar);
31:              contentString := MakeContent( ASSIGN,
                                        outVar, fb.outputSpec);
32:              node.content := contentString;
33:              InsertNode(flowGraph, node);
34:           case SELECTION :
35:             { Selection 그룹에 속한 FB 중 일부는 if-then-
                else 또는 switch 형태의 노드 구조로 변환}
36:             if fb.name = SEL or
                   fb.name contain MUX then
37:             {조건 노드 생성}
38:              curNodeNum := curNodeNum + 1;
39:              node := CreateNode(curNodeNum);
40:              contentString := MakeContent( IF,
                                        fb.condVar);
41:              node.content := contentString;
42:              InsertNode(flowGraph, node);
43:              {출력 변수 지정}
44:              outVar := SetOutVariable ( fb.executionNo,
                                        fb.outputVar);
45:              {조건에 따라 하위 노드 생성}
46:              for i=0 to fb.inputNum-1
47:                 curNodeNum := curNodeNum + 1;
48:                 node := CreateNode(curNodeNum);
49:                 contentString := MakeContent( ASSIGN,
                                        outVar, fb.in[i]);
50:                 node.content := contentString;
51:                 AddNode(childNodeList, node);
52:              end { for_i }
53:              InsertMultipleNodes(flowGraph,
                                        childNodeList);
54:           else
55:              curNodeNum := curNodeNum + 1;
56:              node := CreateNode(curNodeNum);
57:              outVar := SetOutVariable ( fb.executionNo,
                                        fb.outputVar);
58:              contentString := MakeContent( ASSIGN,
                                        outVar, fb.outputSpec);
59:              node.content := contentString;
60:              InsertNode(flowGraph, node);
61:           end { if fb.name }
62:         case default : break;
63:       end { switch }
64:     end { for curFbIndex }
65:     output flowGraph;
66: end { GenFlowGraphFromFBDs }
```

라인 2~5 사이에 기술된 것은 단위 FBD 프로그램을 흐름 그래프로 변환시 필요한 정보를 담은 입력들로서, 실행순서를 인덱스로 하여 저장된 전체 함수 블록들의 정보와, 함수 블록 실행번호의 시작과 끝 값, FBD 단위 프로그램에서 입력으로 쓰인 변수들에 대한 정보이다.

그림 4 의 FBD 에 위의 알고리즘을 적용하면, 먼저 라인 15~19 사이에 기술된 알고리즘을 따라, 단위 FBD 프로그램 내에서 입력변수로 사용된 변수들을 읽어 들이는 노드가 만들어진다. 그림 4 의 FBD 에서 입력변수들은 *Cond_a, Cond_b, Cond_c, Cond_d, status, th_Prev_X_Trip, th_Prev_Trip* 이며, 이들을 읽어들이는 노드가 그림 5 의 흐름 그래프 첫 번째 노드인 노드 0 이 된다.

흐름 그래프에 첫 번째 노드를 삽입한 이후에는, FBD 프로그램의 함수 블록들을 실행순서에 따라 하나하나씩 불러들여 흐름 그래프의 노드로 변환해

나간다. 그림 4 에서 첫 번째 실행노드는 (11)번 AND_BOOL 함수 블록이다. AND_BOOL 함수 블록은 Bitwise Boolean 함수 그룹이므로 라인 23 에서 fb.group 값이 BITWISE_BOOLEAN 이고, 따라서 라인 28~33 부분이 실행된다. ARITHMETIC, BITWISE_BOOLEAN, COMPARISON 함수 그룹에 속한 함수 블록들에 대해서는 변환 프로세스가 같다. 라인 28~29 에서 노드 1 이 새로이 생성되고, 라인 30 에서 v_11 이라는 변수가 생성된다.

라인 30 의 outVar:=SetOutVariable(fb.executionNo, fb.outputVar) 은 함수 블록의 출력변수의 이름을 결정해서 outVar 에 할당해 주는 기능을 한다. 그림 4 의 실행번호 (23) 함수 블록 MUX_INT 는 출력변수가 *th_Prev_X_Trip* 이고, (24) 함수 블록 MOVE 는 출력변수가 *th_Prev_Trip* 이다. 이들처럼 함수 블록의 출력변수가 지정되어 있는 경우는 outVar 변수에 지정된 출력변수의 이름이 할당된다. 그림 4 의 실행번호 (11) AND_BOOL 함수 블록이나, (12)번 SEL 함수 블록처럼 출력 변수의 이름이 지정되어 있지 않은 경우에는 출력에 대해서 변수를 생성해 주고, outVar 에 새로 생성된 변수의 이름을 할당한다. 그림 5 의 흐름 그래프에 나타난 v_11, v_12,…,v_22 와 같은 변수들이 기존의 FBD 에는 명시되어 있지 않지만, 변환과정에서 SetOutVariable 에 의해 생성되는 변수들이다.

라인 30 에서 출력변수가 지정되면, 라인 31 의 contentString := MakeContent(ASSIGN, outVar, fb.outputSpec) 에서, 지정된 출력변수에 값을 할당해주는 명령문이 노드의 내용으로 만들어진다. (11)번 AND_BOOL 은 *Cond_a, Cond_d* 두 개의 입력을 받아서 AND 연산을 수행하는 함수 블록으로서, 이 함수 블록의 출력형식(fb.outputSpec) 은 $Cond\_a \wedge Cond\_d$ 이다. 전체 알고리즘에 입력으로 들어오는 FunctionBlock 구조의 배열인 fbArray 에 이러한 정보는 이미 저장되어 있다고 가정한다. outVar 는 앞에서 v_11 로 지정되었기 때문에, 라인 30 에서 MakeContent 의 결과값으로는 예를 들면 "v_11 = $Cond\_a \wedge Cond\_d$"와 같은 형태의 명령문이 만들어지게 된다. 이것은 라인 32 에서 노드 1 의 내용(content)으로 지정이 된다. 노드의 내용은 다양하게 기술될 수 있기 때문에 MakeContent 와 같이 추상적 함수로만 기술하고 자세하게 지정하지는 않았다.

그림 4 의 (11)번 AND_BOOL 함수 블록은 함수 블록의 연산 수행시 입력 조건에 따라서 제어 흐름이 나뉘어지지 않기 때문에 그림 5 의 노드 1 과 같이 하나의 노드로 변환된다. 생성된 노드는 흐름 그래프에 추가된다.

(12)번 SEL 함수 블록은 조건입력값 *Cond_d* 가 0 인지 1 인지에 따라 출력값이 *th_Prev_X_Trip* 이 될 수도 있고 0 이 될 수도 있다. 즉, 출력에 대한 제어 흐름이 입력값에 따라 두 가지(branch)로 나뉘어지는 경우인데, 이는 라인 34~53 에서 기술한 알고리즘을 따라 그림 5 의 노드 2, 3, 4 와 같이 두 가지(branch)를 가지는 if-then-else 구조로 변환되게 된다. 라인 37~42 까지가 노드 2 를 생성하는 부분이고, 라인 43~53 까지가 노드 3, 4 를 생성하는 부분이다.

그림 4 의 (23)번 MUX_INT 함수 블록은 제어 흐름이 여러 가지로 나뉘어 지는 경우인데, SEL 함수 블록의 경우와 마찬가지로 라인 34~53 에 기술된 알고리즘이 적용되어, 그림 5 의 노드 29~33 처럼 여러 개의 가지를 가지는 구조로 변환된다.

5. FBD 에 대한 단위 테스팅

이 장에서는 FBD 로부터 변환된 흐름 그래프에 제어 흐름 테스팅과 데이터 흐름 테스팅 기법을 적용한다. 먼저, 제어 흐름 테스팅 커버리지들을 명시하고, 그 커버리지를 만족하는 테스트 케이스들을 생성하며, 이후 데이터 흐름 테스팅 커버리지들을 명시하고, 마찬가지로 데이터 흐름 테스팅 커버리지를 만족하는 테스트 케이스들을 생성한다.

5.1 제어 흐름 테스팅(Control flow testing)

FBD 테스팅을 위해, FBD 단위 프로그램을 흐름 그래프로 변환한 후에는 적합한 테스트 커버리지를 선별하고, 그 테스트 커버리지를 만족하는 테스트 케이스를 생성한다. 테스트 커버리지란 테스트 케이스들이 프로그램을 커버하는 영역을 나타내는 것으로서, 테스팅이 얼마만큼 충분히 수행되었는지를 판별하기 위한 기준으로 사용된다. 테스트 커버리지는 테스트 케이스에 의해 수행되어야 할 프로그램 내의 최소 집합을 명시한다고 할 수 있다. 테스트 케이스 집합이 주어지면 대상 프로그램에 대해 테스트 케이스들을 실행하고, 실행을 통해 프로그램에서 얼마만큼의 영역이 커버되었는지 조사함으로써, 테스트 커버리지 기준이 만족되었는지 안 되었는지 검사할 수 있다.

제어 흐름을 테스팅하기 위한 커버리지는 All-nodes, All-edges, All-path 등이 있다[7]. All-nodes 테스트 커버리지를 만족하기 위해서는, 흐름 그래프에 있는 모든 노드들이 최소 한번은 테스트 케이스에 의해 실행되어야 한다. All-edges 테스트 커버리지는 테스트 케이스들 실행으로 흐름 그래프 내의 모든 에지(edge)가 한번 이상 실행될 것을 요구하는 기준이다. All-edges 테스트 커버리지는 All-nodes 커버리지를 포함하는데, 모든 에지를 최소 한 번 실행하는 테스트 케이스들이라면 모든 노드를 한 번 이상 방문한다는 것도 보장되기 때문이다. All-paths 테스트 커버리지는 프로그램 내에 존재하는

가능한 완전한 경로(complete path)가 모두 테스트 되어야 한다는 기준이다. 완전한 경로라 함은 흐름 그래프의 시작 노드로부터 마지막 노드까지의 경로를 말한다. All-paths 커버리지는 All-edges 커버리지를 포함하고, 따라서, All-nodes 커버리지도 포함한다. All-paths 커버리지는 매우 엄격한 커버리지로서, 만족시키기가 거의 불가능한 경우가 많다.

Test Case	Cond_ a	Cond_ b	Cond_c	Cond_ d	status	th_Prev_ X_Trip	Expected Output
T1	1	0	1	0	0	1	1
T2	1	1	0	1	1	0	0
T3	0	0	0	0	2	1	1
T4	0	1	1	1	3	0	1

표 1. All-edges 테스트 커버리지를 만족시키는 테스트 케이스들

표 1 은 그림 5 의 흐름 그래프에 대해서 All-edges 커버리지를 만족하는 테스트 케이스들이다. 왼쪽으로부터 *Cond_a, Cond_b, Cond_c, Cond_d, status, th_Prev_X_Trip* 여섯 개의 열은 *th_X_Pretrip* 단위 프로그램에 대한 입력 변수들이다. 마지막 열의 Expected Output 은 같은 행의 여섯 개의 입력을 가지고 프로그램을 실행했을 때 나와야 할 값으로, *th_X_Pretrip* 에 대한 예상 결과값이다. 테스트 케이스를 다 만들고 나면, 각 입력값들을 가지고 흐름 그래프를 따라가면서 출력 결과값을 얻고 그것을 기대 결과값(expected output)과 비교한다. 두 값이 다르면, 테스트 케이스에 대해서 프로그램이 잘못된 결과값을 내보내는 것이므로, 오류가 있다는 증거이다.

5.2 데이터 흐름 테스팅(Data flow testing)

데이터 흐름 테스팅은 변수가 정의되고 사용된 지점에 초점을 맞춘 구조적 테스팅의 한 형태이다[1]. 흐름 그래프에서 어떤 변수 v 에 대해 그 변수의 값이 정의된 노드를 변수 v 의 정의 노드(definition node)라고 하는데, 어떤 노드가 변수 v 의 정의노드가 되는 경우는 그 노드에서 변수 v 가 입력으로 불러들여지거나(입력에 의한 정의), 변수 v 에 어떤 값이 할당되는 경우(할당문에 의한 정의)이다. 어떤 노드에서 변수 v 가 계산이나 조건에 사용된다면 그 노드는 변수 v 에 대한 사용 노드(usage node)라고 한다[8]. 정의 노드로부터 사용 노드까지의 경로는 du-path 라고 하며, du-path 중에서 정의 노드와 사용 노드 사이에 또 다른 정의 노드를 포함하지 않는 경로를 definition clear path 라고 한다. 그림 5 의 노드 1 은 v_11 = *Cond_a* ∧ *Cond_d*'인데, 이 노드 1 은 변수 v_11 에 대해서는 정의 노드이며, *Cond_a* 와 *Cond_d* 에 대해서는 사용 노드이다.

데이터 흐름 테스팅을 위해서는 프로그램의 모든 변수들에 대한 정의 노드와 사용 노드, 각 변수에 대한 du-path 들을 확인하고 그것들을 기반으로 All-Defs, All-Uses, All-DU-paths 등의 테스트 커버리지를 적용한다[9]. 프로그램 그래프 내의 경로들로 이루어진 어떤 집합이 프로그램의 모든 변수에 대해서, 각 변수의 모든 정의 노드로부터 하나의 사용 노드까지 definition clear path 를 가진다면 이 경로들의 집합은 All-Defs 커버리지를 만족한다. 각 변수들에 대해서 모든 정의 노드로부터 모든 사용 노드까지 definition clear path 를 가진다면 All-Uses 커버리지를 만족하며, All-DU-paths 커버리지를 만족하는 경우는 모든 정의 노드로부터 모든 사용 노드들과 그 이후 사용 노드들까지 definition clear path 를 가지는 경우이다.

FBD 를 변환해서 생성한 흐름 그래프에는 두 가지 종류의 변수가 있다. 한 가지는 시스템에 명시적으로 정의하여 사용하는 입력 및 출력변수들이며, 다른 한 가지는 기존 FBD 상에는 나타나지 않지만, 흐름 그래프로 변환하는 과정에서 추가해 준, v_number 형태의 함수 블록 출력값을 저장한 변수들이다. FBD 데이터 흐름 테스팅에서는 이 두 가지 종류의 변수를 모두 프로그램 내 변수로 고려하여, 각 변수에 대한 정의 노드, 사용 노드, du-path 정보를 추출해야 한다.

보통 FBD 프로그램에서는 변수들을 모두 사용 전에 미리 정의하도록 제한하고 있기 때문에, FBD 단위 프로그램 내에서 입력 및 출력변수로 사용된 것들은 모두 앞서 선언된 것들이다. 따라서, 데이터 흐름 테스팅에서 발견되는 오류 중에서, 선언되지 않은 변수가 사용되는 오류는 FBD 에서 발생하지 않는다. 이러한 특성을 반영하여 FBD 데이터 흐름 테스팅에서 변수의 선언은 따로 정의노드로 여기지 않고, 입력에 의한 정의와 할당문에 의한 정의만을 정의 노드로 여긴다. FBD 단위 프로그램을 흐름 그래프로 변환시, 단위 프로그램 내에서 입력 변수로 사용된 변수들은 흐름 그래프의 첫 번째 노드에서 모두 불러들이는 것으로 변환했기 때문에, 변환된 그래프의 첫 번째 노드는 모든 입력 변수들에 대한 정의 노드가 된다. 예를 들어, 그림 4 의 FBD 단위 프로그램에는 *Cond_a*, *Cond_b*, *Cond_c*, *Cond_d*, *th_Prev_X_Trip*, *th_Prev_Trip*, *status* 와 같은 입력변수들이 사용되었는데, 이들은 모두 흐름 그래프의 첫번째 노드인 그림 5 의 노드 0 에서 불러들여지고 있으며, 따라서, 노드 0 은 이 입력 변수들에 대해서 각각 정의 노드가 된다.

v_number 로 표시된 함수 블록의 출력을 나타내는 변수들과 단위 프로그램에서 출력변수로 사용되는 변수들은 입력에 의한 정의는 없고, 할당문에 의한 정의만 존재한다. 그림 4 의 FBD 에서 *th_X_Trip* 변수는 실행번호 (23)번 함수 블록에서

출력 변수로 사용되고 있는데, 이 변수는 그림 5 의 노드 30~33 에서 각각 할당문에 의해 정의되고 있다. *th_X_Trip* 변수에 대한 정의 노드는 노드 30, 31, 32, 33 네 개가 된다.

함수 블록의 출력을 나타내는 v_number 형태의 변수들은 한 경로에서 정의 노드와 사용 노드가 하나씩만 존재하기 때문에 그 변수들에 대한 du-path 는 모두 definition clear path 이다. 또한, FBD 는 기본적으로 루프(loop)구조를 포함하지 않기 때문에, 테스트 케이스 생성시 루프 구조는 고려하지 않아도 된다.

표 2 는 그림 5 의 흐름 그래프에 존재하는 모든 변수들에 대한 du-path 정보들을 바탕으로 생성한, All-DU-paths 커버리지를 만족하는 테스트 케이스들이다.

Test Case	Cond_ a	Cond_ b	Cond_c	Cond_ d	status	th_Prev_ X_Trip	Expected Output
T1	0	0	0	0	0	1	1
T2	1	0	0	1	1	1	0
T3	1	1	0	0	0	1	1
T4	0	0	0	0	1	0	0
T5	0	1	0	0	2	0	0
T6	1	1	1	0	2	0	1
T7	1	1	1	0	3	0	0
T8	0	0	0	1	3	1	1

표 2. All-DU-paths 커버리지를 만족시키는 테스트 케이스

5.3 사례 연구

우리는 본 논문에 제안한 방법을 현재 KNICS[3]에서 개발중인 디지털 발전소 보호계통의 원자로 보호계통 중 BP 의 트립 논리에 적용하였다. 본 절에서는 BP 트립 논리에 대한 FBD 프로그램을 예제로 사용하여, 제안된 FBD 테스팅 방법을 통해 어떻게 FBD 프로그램 내의 다양한 오류들을 찾아낼 수 있는지 설명한다. 우리는 그림 4 의 *th_X_Pretrip* FBD 프로그램에 네 가지 오류를 삽입하였다. 이 오류들은 모두 FBD 프로그래밍을 할 때 자주 발생되는 것들이다. FBD 프로그래밍을 할 때 발생할 수 있는 다양한 오류들에 대한 설명과 분류는 [10]을 참조한다. 심겨진 오류들은, 제어 흐름 테스팅에서 All-edges 커버리지를 만족하도록 만든 표 1 의 테스트 케이스들에 의해서 모두 발견될 수 있었으며, 데이터 흐름 테스팅에서 All-DU-paths 커버리지를 만족하도록 만든 표 2 의 테스트 케이스들을 통해서도 모두 발견될 수 있었다.

- **오류 케이스 1 (입력 순서의 치환)** : FBD 프로그래밍을 하다 보면 종종 입력 변수들의 순서를 뒤바뀌어 기술하는 실수가 발생한다. AND_BOOL 같은 함수 블록은 입력값들의 순서가 바뀌어도 관계가 없지만, SEL, MUX, GE_INT 와 같은 함수 블록들에는 입력값들의 순서가 바뀔 경우 의미가 반대가 되면서 잘못된 결과가 나오게 되기 때문에 입력값들의 순서를 정확하게 하는 것이 중요하다. 우리는 그림 4 의 실행번호 (12)번 SEL 함수 블록의 입력을 치환하여서, 각각 *th_Prev_X_Trip* 과 0 이어야 하는 IN0 입력과 IN1 입력을 0 과 *th_Prev_X_Trip* 이 되게 하였다. 이 오류는 표 1 에 있는 제어 흐름 테스팅의 테스트 케이스 T1 에 의해서 발견되었고, 표 3 의 데이터 흐름 테스팅의 테스트 케이스 T1 에 의해 발견된다. expected output 은 1 이지만, 테스트 케이스를 실행해 보면 출력값이 0 이 나오게 되어 오류가 발견된다.
- **오류 케이스 2 (인버터 사용 오류)** : 필요한 함수 블록을 빠뜨리거나 잘못 배치하는 경우가 FBD 프로그래밍에서 종종 발생하는데, 특히 인버터(inverter) 블록은 그런 일들이 자주 발생한다. 우리는 그림 4 의 실행순서 (18)번 SEL 함수 블록에서 IN0 입력에 불필요한 인버터를 삽입하였다. 이 오류는 제어 흐름 테스팅의 T2 테스트 케이스에 의해서, 데이터 흐름 테스팅의 T2 테스트 케이스에 의해서 발견된다.
- **오류 케이스 3 (변수 입력 오류)** : 입력 또는 출력 변수 이름을 부정확하게 입력하게 되면, 함수 블록에 잘못된 값이 입력되게 된다. 변수의 초기값이 잘못 할당되는 것도 입력 오류로 여긴다. 그림 4 의 실행순서 (20) 번 SEL 함수 블록의 *th_Prev_X_Trip* 입력을 *th_Prev_Trip* 로 잘못 기술하는 오류를 삽입하였는데, 이 오류는 제어 흐름 테스팅의 T3 테스트 케이스에 의해서, 데이터 흐름 테스팅의 T5 테스트 케이스에 의해서 발견된다.
- **오류 케이스 4 (상수 입력 오류)** : 입력 오류의 또 다른 예로 그림 4 의 실행순서 (22)번 SEL 함수 블록의 IN1 입력을 1 에서 0 으로 바꾸었다. 0 을 1 로, 1 을 0 으로 잘못 기재하는 오류는 흔히 발생한다. 이 오류의 경우는 제어 흐름 테스팅의 T4 테스트 케이스에 의해서, 데이터 흐름 테스팅의 T8 테스트 케이스에 의해서 발견된다.

6. 결 론

본 논문에서 우리는 FBD 프로그램으로부터 중간코드를 생성하는 과정을 거치지 않고 직접 FBD 프로그램을 테스팅하는 방안을 제안하였다. 기존에 PLC 기반 소프트웨어들은 FBD 로부터 C 프로그램 같은 별도의 중간코드를 생성해서 그

중간코드에 대해서 테스팅을 수행하였다. FBD 의 행위는 절차적 프로그램의 프로시저나 함수와 비슷하지만, FBD 에 소프트웨어 테스팅 기법을 적용하는 체계적인 방안에 대한 연구는 부재했었다. 본 논문에서는 FBD 에 직접 소프트웨어 테스팅 기법을 적용 가능하게 함으로, 중간코드 생성의 부담을 없애는 효율적인 방안을 제안하였다.

본 논문에서는 먼저 FBD 프로그램의 관점에서 단위 테스팅, 결합테스팅의 대상인 단위와 모듈에 대해 개념을 정의하였다. 이어서, FBD 직접 테스팅을 가능하게 해 주는 핵심이 되는 프로세스인 FBD 프로그램을 흐름 그래프로 변환하는 알고리즘을 제안하였다. FBD 프로그램을 흐름 그래프로 변환하고 나면, 흐름 그래프를 대상으로 기존에 존재하는 소프트웨어 테스팅 기법들을 효과적으로 적용할 수 있다. 본 논문에서는 FBD 로부터 변환된 흐름 그래프에 제어 흐름 테스팅 기법과 데이터 흐름 테스팅 기법을 모두 적용하였고, 제어 흐름 테스팅에서는 All-nodes 커버리지를, 데이터 흐름 테스팅에서 All-DU-paths 커버리지를 만족하는 테스트 케이스들을 생성하였다. 제안된 기법의 효과를 설명하기 위해서, 현재 KNICS 에서 개발중인 디지털 원자로 보호 시스템의 BP 트립 논리 예제를 사용하였다. 입력 순서 치환, 인버터 사용 오류, 변수 입력 오류, 상수 입력 오류 등 FBD 프로그래밍에서 자주 발생하는 오류들을 FBD 프로그램에 삽입하였고, 제안된 기법들을 적용해서 얻은 테스트 케이스들에 의해 이 오류들이 모두 발견될 수 있음을 확인하였다.

향후 연구에서는, 입출력 뿐만 아니라 시간이나 상태까지도 고려해야 하는 Timer 그룹의 함수 블록들을 포함한 FBD 프로그램을 테스팅할 수 있도록, 시간에 관련된 함수 블록들을 흐름 그래프로 변환하는 알고리즘이 보완되어야 한다. 또한, FBD 단위 프로그램들이 모두 테스팅 된 상태에서, 단위들 간의 인터페이스나 상호작용들을 테스팅하는 FBD 결합 테스팅 방안에 대한 연구가 필요하다.

참고문헌

[1] Paul C. Jorgensen, “Software testing: a craftsman’s approach”, CRC Press, 1995

[2] A. Mader, “A Classification of PLC Models and Applications”, In Proc. *WODES 2000: 5th Workshop on Discrete Event Systems*, August 21-23, Gent, Belgium, 2000.

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

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

[5] J. Yoo, T. Kim, S. Cha, J-S. Lee, H.S. Son, “A Formal Software Requirements Specification Method for Digital Nuclear Plants Protection Systems”, *Journal of Systems and Software*, in press, 2003.

[6] J. Cho, J. Yoo, S. Cha, “NuEditor – A Tool Suite for Specification and Verification of NuSCR”, In proc. *Second ACIS International Conference on Software Engineering Research, Management and Applications (SERA2004)*, pp298-304, LA, USA, May 5-7, 2004.

[7] E. F. Miller, “Tutorial : Program Testing Techniques”, at COMPSAC ’77 IEEE Computer Society, 1977.

[8] P. G. Frankl, E. J. Weyuker, “An applicable family of data flow testing criteria”, *IEEE Trans. Software Engineering*, 14(10), pp1483–1498, Oct. 1988.

[9] S. Rapps and E. J. Weyuker, “Selecting software test data using data flow information”, *IEEE Transactions on Software Engineering*, vol. SE-11, no. 4, pp.367-375, April, 1985.

[10] Y. Oh, J. Yoo, S. Cha, H.S. Son, “Software Safety Analysis of Function Block Diagrams using Fault Trees”, *Reliability Engineering and System Safety*, in press, 2004.