(19)대한민국특허청(KR)
(12) 공개특허공보(A)

(51) 。Int. Cl.
G06F 11/26 (2006.01)
G06F 9/06 (2006.01)
G11C 29/00 (2006.01)
G06F 9/45 (2006.01)

(11) 공개번호 10-2007-0014206
(43) 공개일자 2007년01월31일

(21) 출원번호 10-2006-7027010
(22) 출원일자 2006년12월21일
심사청구일자 없음
번역문 제출일자 2006년12월21일
(86) 국제출원번호 PCT/JP2005/009813
국제출원일자 2005년05월23일
(87) 국제공개번호 WO 2005/114235
국제공개일자 2005년12월01일

(30) 우선권주장 10/918,714 2004년08월13일 미국(US)
60/573,577 2004년05월22일 미국(US)

(71) 출원인 주식회사 아도반테스토
일본 도쿄, 네리마-쿠, 아사히-쵸 1-쵸메, 32-1

(72) 발명자 프라마닉 안칸
미국 캘리포니아 95318 산 호세 캐필라노 드라이브 5839
엘스톤 마크
미국 캘리포니아 93906 살리나스 머롯 웨이 1777
크리쉬나스와미 라마찬드란
미국 캘리포니아 95014 쿠퍼티노 메테오르 드라이브 21327
아다치 토시아키
미국 캘리포니아 95120 산 호세 니켓트 웨이 1196

(74) 대리인 오병석

전체 청구항 수 : 총 32 항

(54) 반도체 집적 회로를 위한 테스트 프로그램을 개발하는 방법및 구조

(57) 요약

반도체 테스트 시스템을 위한 테스트 프로그램을 개발하는 방법이 개시된다. 상기 방법은 테스트 프로그램 언어(test program language: TPL)로 테스트 계획 파일(test plan file)을 기술하는 단계(여기서, 상기 테스트 계획 파일은 상기 테스트 프로그램의 하나 이상의 테스트를 기술한다), 시스템 프로그램 언어(system program language; SPL)로 테스트 클래스 파일을 기술하고 상기 TPL로 상기 테스트 클래스 파일의 대응되는 프리헤더(pre-header) 파일을 기술하는 단계(여기서, 상기 테스트 클래스 파일은 상기 테스트 프로그램의 하나 이상의 테스트의 구현을 기술한다), 및 상기 테스트 계획 파일, 상기 테스트 클래스 파일 및 상기 프리헤더 파일을 사용하여 상기 테스트 프로그램을 생성하는 단계를 포함한다.

대표도

도 1

특허청구의 범위

청구항 1.

피시험 디바이스에 대하여 하나 이상의 테스트를 적용하는 하나 이상의 테스트 모듈을 포함하는 반도체 테스트 시스템을 위한 테스트 프로그램을 개발하는 방법에 있어서,

테스트 프로그램 언어(test program language; TPL)로 테스트 계획 파일(test plan file)을 기술하는 단계 - 상기 테스트 계획 파일은 상기 테스트 프로그램의 하나 이상의 테스트를 기술함 -;

시스템 프로그램 언어(system program language; SPL)로 테스트 클래스 파일을 기술하고 상기 TPL로 상기 테스트 클래스 파일의 대응되는 프리헤더(pre-header) 파일을 기술하는 단계 - 상기 테스트 클래스 파일은 상기 테스트 프로그램의 하나 이상의 테스트의 구현을 기술함 -; 및

상기 테스트 계획 파일, 상기 테스트 클래스 파일 및 상기 프리헤더 파일을 사용하여 상기 테스트 프로그램을 생성하는 단계

를 포함하는 테스트 프로그램 개발 방법.

청구항 2.

제1항에 있어서,

상기 시스템 프로그램 언어는 C/C++인 테스트 프로그램 개발 방법.

청구항 3.

제1항에 있어서,

상기 테스트 프로그램을 생성하는 단계는,

상기 SPL로 기술된 유도된 테스트 계획 파일(derived test plan file)을 형성하기 위하여 TPL 컴파일러에 의하여 상기 테스트 계획 파일을 컴파일하는 단계; 및

상기 SPL로 기술된 헤더 파일을 형성하기 위하여 상기 TPL 컴파일러에 의하여 상기 프리헤더를 컴파일하는 단계

를 포함하는 테스트 프로그램 개발 방법.

청구항 4.

제3항에 있어서,

상기 테스트 계획 파일을 컴파일하는 단계는,

상기 TPL로부터 상기 SPL로 상기 테스트 계획 파일을 번역하는 단계; 및

상기 프리헤더 파일을 사용하여 상기 테스트 계획 파일을 유효화하는 단계

를 포함하는 테스트 프로그램 개발 방법.

청구항 5.

제3항에 있어서,

상기 프리헤더 파일을 컴파일하는 단계는,

상기 프리헤더 파일의 파라미터 섹션에 기술된 파라미터를 상기 헤더 파일에 삽입하는 단계; 및

상기 프리헤더 파일의 템플리트(template) 섹션에 참조된 소스 코드(source code)를 상기 헤더 파일에 삽입하는 단계

를 포함하는 테스트 프로그램 개발 방법.

청구항 6.

제3항에 있어서,

상기 헤더 파일을 참조하는 테스트 계획 객체 파일(test plan object file)을 형성하기 위하여 SPL 컴파일러에 의하여 상기 유도된 테스트 계획 파일을 컴파일하는 단계; 및

테스트 클래스 객체 파일을 형성하기 위하여 상기 SPL 컴파일러에 의하여 상기 테스트 클래스 파일을 컴파일하는 단계 - 상기 SPL 컴파일러는 상기 테스트 클래스 객체 파일을 형성하기 위하여 상기 헤더 파일을 사용함 -

를 포함하는 테스트 프로그램 개발 방법.

청구항 7.

제6항에 있어서,

상기 반도체 테스트 시스템에 의하여 실행 가능한 상기 테스트 프로그램을 형성하기 위하여 상기 테스트 계획 객체 파일과 상기 테스트 클래스 객체 파일을 링크하는 단계를 더 포함하는 테스트 프로그램 개발 방법.

청구항 8.

반도체 테스트 시스템을 위한 테스트 프로그램을 생성하기 위한 프리헤더에 있어서,

상기 테스트 프로그램의 하나 이상의 속성을 특정하는 파라미터 섹션; 및

상기 테스트 프로그램의 헤더 파일로 소스 코드를 삽입하는 템플리트 섹션을 포함하는 프리헤더.

청구항 9.

제8항에 있어서,

상기 테스트 프로그램의 하나 이상의 속성은 패턴 목록 및 테스트 조건에 관련되는 프리헤더.

청구항 10.

제9항에 있어서,

상기 패턴 목록은,

카디낼리티(cardinality);

추적 레벨(trace level);

멤버 명칭(member name);

설정 함수(set function); 및

파라미터 기술(parameter description)

등의 파라미터들의 하나 이상을 포함하는 프리헤더.

청구항 11.

제9항에 있어서,

상기 테스트 조건은,

카디낼리티(cardinality);

추적 레벨(trace level);

멤버 명칭(member name);

설정 함수(set function); 및

파라미터 기술(parameter description)

등의 파라미터들의 하나 이상을 포함하는 프리헤더.

청구항 12.

제8항에 있어서,

상기 템플리트 섹션은,

클래스 명칭(class name);

인입 선언(import declaration);

파라미터 배열 형식(parameter array type);

파라미터 속성(parameter attribute);

파라미터 함수; 및

파라미터 함수 구현

을 포함하는 프리헤더.

청구항 13.

제8항에 있어서,

상기 프리헤더는 헤더 파일을 생성하기 위하여 테스트 프로그램 언어(TPL) 컴파일러에 의하여 사용되며, 상기 헤더 파일은 시스템 프로그램 언어(SPL)로 기술되는 프리헤더.

청구항 14.

제13항에 있어서,

상기 프리헤더는 테스트 클래스 객체 파일을 생성하기 위하여 SPL 컴파일러에 의하여 사용되며, 상기 헤더 파일은 테스트 계획 객체 파일을 생성하기 위하여 상기 SPL 컴파일러에 의하여 참조되는 프리헤더.

청구항 15.

제8항에 있어서,

상기 프리헤더는 상기 테스트 프로그램에서 사용자 정의 테스트를 지원하기 위하여 사용되는 프리헤더.

청구항 16.

피시험 디바이스에 대하여 하나 이상의 테스트를 적용하는 하나 이상의 테스트 모듈을 포함하는 반도체 테스트 시스템을 위한 테스트 프로그램을 유효화하는 방법에 있어서,

테스트 프로그램 언어(test program language; TPL)로 테스트 계획 파일(test plan file)을 기술하는 단계 - 상기 테스트 계획 파일은 상기 테스트 프로그램의 하나 이상의 테스트를 기술함 -;

상기 TPL로 프리헤더 파일을 기술하는 단계; 및

상기 프리헤더 파일을 사용하여 테스트 계획 파일을 유효화하는 단계

를 포함하는 테스트 프로그램 유효화 방법.

청구항 17.

제16항에 있어서,

상기 프리헤더 파일은,

상기 테스트 프로그램의 하나 이상의 속성을 특정하는 파라미터 섹션; 및

상기 테스트 프로그램의 헤더 파일로 소스 코드를 삽입하는 템플리트 섹션을 포함하는 테스트 프로그램 유효화 방법.

청구항 18.

제17항에 있어서,

상기 테스트 프로그램의 하나 이상의 속성은 패턴 목록 및 테스트 조건과 관련되는 테스트 프로그램 유효화 방법.

청구항 19.

제16항에 있어서,

상기 테스트 계획 파일을 유효화하는 단계는,

상기 테스트 계획 파일의 의미 체계(semantics)를 확인하는 단계를 포함하는 테스트 프로그램 유효화 방법.

청구항 20.

제16항에 있어서,

상기 테스트 계획 파일을 유효화하는 단계는,

상기 테스트 계획 파일의 테스트 엔티티 속성을 확인하는 단계를 더 포함하는 테스트 프로그램 유효화 방법.

청구항 21.

제16항에 있어서,

상기 테스트 계획 파일을 유효화하는 단계는,

상기 테스트 계획의 함수 호출 및 그들의 대응 파라미터들을 확인하는 단계를 더 포함하는 테스트 프로그램 유효화 방법.

청구항 22.

반도체 테스트 시스템에 있어서,

피시험 디바이스에 하나 이상의 테스트를 적용하기 위한 하나 이상의 테스트 모듈;

테스트 계획 파일, 테스트 클래스 파일 및 상기 테스트 클래스 파일의 대응되는 프리헤더 파일을 수신하는 제어기 - 상기 테스트 계획 파일은 테스트 프로그램 언어(TPL)로 하나 이상의 테스트를 기술하며, 상기 테스트 클래스 파일은 시스템 프로그램 언어(SPL)로 하나 이상의 테스트의 구현을 기술함 -;

유도된 테스트 계획 파일 및 헤더 파일을 형성하기 위하여 상기 테스트 계획 파일 및 상기 프리헤더 파일을 각각 컴파일하기 위한 수단; 및

상기 테스트 클래스 파일, 상기 유도된 테스트 계획 파일 및 상기 헤더 파일을 사용하여 테스트 프로그램을 생성하기 위한 수단

을 포함하는 반도체 테스트 시스템.

청구항 23.

제22항에 있어서,

상기 프리헤더 파일은,

상기 테스트 프로그램의 하나 이상의 속성을 특정하는 파라미터 섹션; 및

상기 테스트 프로그램의 상기 헤더 파일로 소스 코드를 삽입하기 위한 템플리트 섹션을 포함하는 반도체 테스트 시스템.

청구항 24.

제23항에 있어서,

상기 테스트 프로그램의 상기 하나 이상의 속성은 패턴 목록 및 테스트 조건과 관련되는 반도체 테스트 시스템.

청구항 25.

제24항에 있어서,

상기 패턴 목록은,

카디낼리티(cardinality);

추적 레벨(trace level);

멤버 명칭(member name);

설정 함수(set function); 및

파라미터 기술(parameter description)

등의 파라미터들의 하나 이상을 포함하는 반도체 테스트 시스템.

청구항 26.

제24항에 있어서,

상기 테스트 조건은,

카디낼리티(cardinality);

추적 레벨(trace level);

멤버 명칭(member name);

설정 함수(set function); 및

파라미터 기술(parameter description)

등의 파라미터들의 하나 이상을 포함하는 반도체 테스트 시스템.

청구항 27.

제23항에 있어서,

상기 템플리트 섹션은,

클래스 명칭(class name);

인입 선언(import declaration);

파라미터 배열 형식(parameter array type);

파라미터 속성(parameter attribute);

파라미터 함수; 및

파라미터 함수 구현

을 포함하는 반도체 테스트 시스템.

청구항 28.

제22항에 있어서,

상기 시스템 프로그램 언어는 C/C++인 반도체 테스트 시스템.

청구항 29.

제22항에 있어서,

상기 테스트 계획 파일을 컴파일하기 위한 수단은,

상기 테스트 계획 파일을 상기 TPL로부터 상기 SPL로 번역하기 위한 수단; 및

상기 프리헤더 파일을 사용하여 상기 테스트 계획을 유효화하기 위한 수단

을 포함하는 반도체 테스트 시스템.

청구항 30.

제22항에 있어서,

상기 프리헤더 파일을 컴파일하기 위한 수단은,

상기 프리헤더 파일의 파라미터 섹션에 기술된 파라미터를 상기 헤더 파일에 삽입하기 위한 수단; 및

상기 프리헤더 파일의 템플리트 섹션에 참조된 소스 코드를 상기 헤더 파일에 삽입하기 위한 수단

을 포함하는 반도체 테스트 시스템.

청구항 31.

제22항에 있어서,

상기 테스트 프로그램을 생성하기 위한 수단은,

상기 헤더 파일을 참조하는 테스트 계획 객체 파일을 형성하기 위하여 상기 유도된 테스트 계획 파일을 컴파일하기 위한 수단; 및

테스트 클래스 객체 파일을 형성하기 위하여 상기 테스트 클래스 파일을 컴파일하기 위한 수단 - 상기 헤더 파일은 상기 테스트 클래스 객체 파일을 형성하기 위하여 사용됨 -

을 포함하는 반도체 테스트 시스템.

청구항 32.

제31항에 있어서,

상기 반도체 테스트 시스템에 의하여 실행 가능한 상기 테스트 프로그램을 형성하기 위하여 상기 테스트 계획 객체 파일과 상기 테스트 클래스 객체 파일을 링크하기 위한 수단을 더 포함하는 반도체 테스트 시스템.

명세서

기술분야

본 출원은, 2003.2.14.자 미국 출원 제60/447,839호인 "반도체 집적 회로의 테스트 프로그램 개발 방법 및 구조"를 우선권 주장의 기초로 하는 2004.2.6.자 미국 출원 제10/772,434호인 "반도체 집적 회로를 위한 테스트 프로그램 개발 방법 및 구조"의 부분 계속 출원(continuation-in-part application)이며 이를 우선권 주장의 기초로 한다; 본 출원은 또한, 2004.5.22.자 미국 임시 출원 제60/573,577호 "개방형 구조 테스트 시스템의 소프트웨어 개발"을 우선권 주장의 기초로 한다; 이들 모두는 주식회사 아도반테스토에 양도되었으며 그 전체가 참조에 의하여 본 명세서에 편입된다.

본 발명은 반도체 시험을 위한 자동 테스트 장비의 분야에 관한 것이다. 구체적으로, 본 발명은 반도체 테스트 시스템을 위한 테스트 프로그램을 개발하는 방법에 관한 것이다.

배경기술

반도체 테스트 시스템은 함께 동작하는 다수의 디바이스를 포함하는 복잡한 시스템 환경을 갖는다. 상기 테스트 시스템은, 복수의 벤더 모듈과 그에 대응하는 피시험 디바이스(device-under-tests; DUTs)의 동작을 제어하는 복수의 제어기를 포함할 수 있다. 반도체 테스트 시스템을 위한 테스트 프로그램은 복잡한 시스템 환경에서 동작하도록 요구된다. 그러한 테스트 프로그램을 개발하는 하나의 방법은 테스트 코드를 직접 C++과 같은 시스템 프로그램 언어(SPL)로 작성하는 것이다. 그러나, 이 방법으로는, 테스트 엔지니어가 복잡한 테스트 시스템 환경에 부담을 갖게 된다. 특히, 테스트 엔지니어는 많은 수의 복잡한 시스템 프로그래밍 규칙을 따라야 하고 테스트 프로그램을 개발하기 위해 시스템 프로그램 언어의 요구를 만족시켜야 한다.

현존하는 방법론의 또 다른 문제는, 테스트 프로그램이 SPL로 컴파일될 때까지 또는 테스트 프로그램이 DUT를 시험하는 동안 상기 테스트 프로그램이 동작할 때까지, 테스트 프로그램 내의 오류가 식별되지 않을 수 있다는 것이다. 또한, 복잡한 테스트 시스템 환경에서 테스트 계획(test plan) 오류를 식별하는 것은 종종 어렵고 시간 소모적이다. 그러한 한계는 DUT의 생산을 지연시킬 수 있고, 이는 제품 출하를 지연시켜 시장 기회를 잃을 수도 있다.

따라서, 테스트 계획의 개발로부터 시스템 프로그래밍의 복잡성을 제거할 필요가 있다. 특히, 테스트 시스템을 복잡하게 만드는 언어와 달리, 테스트 엔지니어가 친숙한 문제 도메인에 더 가까운 언어로 테스트 계획을 개발할 필요가 있다. 또한, 시스템 컴파일 전에 테스트 계획의 오류를 식별할 필요가 있다. 테스트 계획 오류의 빠른 검출은 테스트 프로그램의 개발과 DUT를 테스트하는 시간의 감소를 이끌어 낼 수 있다.

발명의 상세한 설명

본원은, 예를 들어 C++ 객체(object) 및 클래스(class)와 같은 객체 지향적 구조(object-oriented constructs)를 사용한 테스트 프로그램의 개발에 관하여 기술한다. 구체적으로, 이 방법은, 본 발명의 양수인에게 양도된, 미국 특허 출원 제60/449,622, 10/404,002 및 제10/403,817호에 기재된 것과 같은, 개방형 구조의 테스터를 위한 테스트 프로그램의 개발에 적합하다.

본 발명의 일 실시예에 의하면, 자동 테스트 장비(ATE)와 같은 반도체 테스트 시스템 상에서 피시험 디바이스(예컨대 IC)를 시험하기 위해 테스트 시스템 리소스, 테스트 시스템 구성, 모듈 구성, 테스트 시퀀스(test sequence), 테스트 계획(test plan), 테스트 조건, 테스트 패턴 및 타이밍 정보를 범용 객체 지향적(예컨대, C/C++) 구조로 기술함으로써 테스트 프로그램을 개발하는 방법을 제공한다. 이러한 기술(description)을 수록한 파일들은, 상기 테스트 시스템 또는 상기 파일을 사용하는 관련 장비에 억세스 가능한 컴퓨터로 판독 가능한 매체와 같은 메모리에 저장된다.

테스트 시스템 리소스(test system resource)의 기술에는 리소스 형식(resource type)(여기서, 상기 리소스 형식은 IC에 시험을 적용하기 위한 적어도 하나의 테스트 모듈과 관련된다.)의 특정, 상기 리소스 형식과 관련된 파라미터 형식(parameter type)의 특정 및 상기 파라미터 타입의 파라미터의 특정이 포함될 수 있다.

상기 테스트 시스템 구성의 기술에는, 적어도 하나의 테스트 모듈의 제어를 위한 사이트 제어기(site controller)의 특정(여기서 각 테스트 모듈은 IC에 시험을 적용한다.) 및 모듈 접속 인에이블러(module connection enabler)의 입력 포트(input port)의 특정이 포함될 수 있다. 상기 테스트 시스템은 특정된 입력 포트에서 상기 사이트 제어기를 상기 모듈 접속 인에이블러와 연결시키고, 상기 모듈 접속 인에이블러는 상기 사이트 제어기를 테스트 모듈(test module)과 연결시킨다. 상기 모듈 접속 인에이블러는 스위치 매트릭스(switch matrix)로서 구현될 수도 있다.

모듈 구성의 기술에는, 모듈 형식(module type)을 특정하기 위한 모듈 식별자(module identifier)의 특정, 상기 모듈 식별자에 의하여 특정된 모듈 형식의 테스트 모듈을 제어하기 위한 실행 가능 코드(executable code)의 특정, 및 상기 테스트 모듈과 관련된 리소스 형식(resource type)의 특정이 포함될 수 있다. 상기 실행 가능 코드는 동적 링크 라이브러리(dynamic link library)의 포맷을 취할 수도 있다.

나아가, 모듈 구성의 기술은, 상기 모듈 접속 인에이블러의 출력 포트(output port)를 특정하는 슬롯 식별자(slot identifier)의 사용자 특정을 포함할 수도 있으며, 여기서 상기 테스트 시스템은 상기 출력 포트에서 상기 테스트 모듈을 상기 모듈 접속 인에이블러와 연결시키며, 상기 모듈 접속 인에이블러는 상기 테스트 모듈을 대응하는 사이트 제어기와 연결시킨다. 사용자는 또한, 상기 테스트 모듈의 공급자를 식별하기 위한 벤더 식별자(vendor identifier), 및 상기 리소스 형식과 관련하여 활용 가능한 리소스의 최대 개수의 식별자를 특정할 수도 있다. 상기 리소스 형식은, 예를 들어 디지털 테스트 핀(digital test pins) 및 리소스 유닛 테스터 채널(resource units test channels)일 수도 있다. 또는, 상기 테스터 채널 리

소스 유닛(tester channel resource units)은 또한, 예를 들어 아날로그 테스터 핀(analog tester pins), RF 테스터 핀, 전원 공급 핀, 디지털화 핀 및 임의 파형 생성 핀(arbitrary waveform generation pins) 등과 같은 리소스 형식에 대응될 수도 있다. 어느 리소스 유닛이 활용 불가능(disabled)하게 되었는지에 관련된 표지(indicator)도 또한 제공될 수 있다. 활용 불가능하다고 표지된 리소스 유닛은 상기 테스트 모듈의 결함있는(defective) 리소스 유닛을 나타낼 수도 있다.

테스트 조건의 기술에는, 적어도 하나의 테스트 조건 그룹(test condition group)의 특정, 적어도 하나의 변수(variable)를 포함하는 규정 세트(specification set)의 특정, 특정 변수의 필수적 표현(expression)의 선택을 위한 선택자(selector)의 특정이 포함될 수 있다. 상기 테스트 조건 그룹을 상기 규정 세트에 대한 선택자와 관련시킴으로써 테스트 조건을 정의한다.

테스트 시퀀스의 기술에는, 다양한 테스트가 적용될 수 있는 순서(또는 흐름)의 특정이 포함될 수 있다.

테스트 패턴의 기술에는, 테스트 패턴, 관련 전압 및 전류 레벨, 신호 값의 전이(transition), 대응 상승 및 하강 시간, 및 관련 타이밍의 특정이 포함될 수 있다.

본 발명의 일 실시예에 의한 패턴 컴파일러는, 적어도 하나의 모듈 특유적 패턴 컴파일러(module-specific pattern compiler), 및 패턴 소스 파일의 대응되는 모듈 특유적 섹션(module-specific section)과 상기 패턴 소스 파일의 공통 섹션(common section) 모두를 컴파일하도록 각각의 모듈 특유적 컴파일러를 지도(direct)하는 객체 파일 관리자(object file manager)를 포함한다. 상기 공통 섹션은 모든 모듈 특유적 컴파일러에 액세스 할 수 있는 정보를 포함한다. 상기 컴파일러로부터의 출력은 적어도 하나의 모듈 특유적 패턴 데이터 섹션을 포함한다. 모듈 특유적 섹션 로더(loader)는, 실행을 위해, 대응되는 모듈 특유적 패턴 데이터 섹션으로부터의 모듈 특유적 패턴 데이터를 대응되는 테스트 모듈로 로딩(loading)한다.

테스트 프로그램을 개발하는 개시된 방법은, 테스트 계획이 처리하도록 의도된 문제 도메인에 더 가까운 언어로 테스트 계획을 기술함으로써, 테스트 계획 개발자로부터 시스템 프로그래밍의 복잡성을 분리시킨다. 상기 방법은 또한 테스트 엔지니어가 테스트 프로그램 개발의 초기 단계에서 테스트 계획의 오류를 식별하도록 하며, 이는 테스트 프로그램의 개발 시간과 대응하는 DUT의 시험 시간을 줄일 수 있다.

일 실시예에서, 반도체 테스트 시스템을 위한 테스트 프로그램을 개발하는 방법은, 테스트 프로그램 언어(test program language: TPL)로 테스트 계획 파일(test plan file)을 기술하는 단계(여기서, 상기 테스트 계획 파일은 상기 테스트 프로그램의 하나 이상의 테스트를 기술한다), 시스템 프로그램 언어(system program language; SPL)로 테스트 클래스 파일을 기술하고 상기 TPL로 상기 테스트 클래스 파일의 대응되는 프리헤더(pre-header) 파일을 기술하는 단계(여기서, 상기 테스트 클래스 파일은 상기 테스트 프로그램의 하나 이상의 테스트의 구현을 기술한다), 및 상기 테스트 계획 파일, 상기 테스트 클래스 파일 및 상기 프리헤더 파일을 사용하여 상기 테스트 프로그램을 생성하는 단계를 포함한다.

다른 실시예에서, 반도체 테스트 시스템을 위한 테스트 프로그램을 생성하기 위한 프리헤더는, 상기 테스트 프로그램의 하나 이상의 속성을 특정하는 파라미터 섹션, 및 상기 테스트 프로그램의 헤더 파일로 소스 코드를 삽입하는 템플리트 섹션을 포함하며, 상기 테스트 프로그램의 하나 이상의 속성은 패턴 목록 및 테스트 조건에 관련된다.

또 다른 실시예에서, 반도체 테스트 시스템을 위한 테스트 프로그램을 유효화하는 방법은, 테스트 프로그램 언어(test program language; TPL)로 테스트 계획 파일(test plan file)을 기술하는 단계(여기서, 상기 테스트 계획 파일은 상기 테스트 프로그램의 하나 이상의 테스트를 기술한다), 상기 TPL로 프리헤더 파일을 기술하는 단계, 및 상기 프리헤더 파일을 사용하여 테스트 계획 파일을 유효화하는 단계를 포함한다.

또 다른 실시예에서, 반도체 테스트 시스템은, 피시험 디바이스에 하나 이상의 테스트를 적용하기 위한 하나 이상의 테스트 모듈, 테스트 계획 파일, 및 테스트 클래스 파일 및 상기 테스트 클래스 파일의 대응되는 프리헤더 파일을 수신하는 제어기(여기서, 상기 테스트 계획 파일은 테스트 프로그램 언어(TPL)로 하나 이상의 테스트를 기술하며, 상기 테스트 클래스 파일은 시스템 프로그램 언어(SPL)로 하나 이상의 테스트의 구현을 기술한다)를 포함한다. 상기 반도체 테스트 시스템은 또한, 유도된 테스트 계획 파일 및 헤더 파일을 형성하기 위하여 상기 테스트 계획 파일 및 상기 프리헤더 파일을 각각 컴파일하기 위한 수단, 및 상기 테스트 클래스 파일, 상기 유도된 테스트 계획 파일 및 상기 헤더 파일을 사용하여 테스트 프로그램을 생성하기 위한 수단을 포함한다.

실시예

반도체 테스트 시스템을 위한 테스트 프로그램을 개발하는 방법 및 시스템이 제공된다. 이하의 기술은 본 발명의 기술 분야에서 숙련된 자가 본 발명을 만들고 이용할 수 있도록 제공된다. 특정 실시예 및 응용에 관한 기술은 단지 예로서 제공된다. 본 발명의 기술 분야에서 숙련된 자에게는 본 명세서에 기술된 예들의 다양한 변경 및 조합이 명백할 것이며, 본 명세서에 정의된 일반적인 원리들은 본 발명의 사상과 범위를 벗어나지 않고 다른 예 및 응용에 적용될 수 있다. 이렇게 해서, 본 발명은 기술된 그리고 도시된 예들로 한정되도록 의도되지 않으며, 본 명세서에 개시된 원리 및 특성과 일치하는 가장 넓은 범위에 합치한다.

본 발명은 일반적으로, 동일한 양수인에 의하여 미국 특허 출원 제60/449,622호, 10/404,002호 및 제10/403,817호에 개시된 개방형 구조의 테스트 시스템에 관하여 기술된다. 그러나, 본 발명의 기술 분야에서 숙련된 자라면, 본 발명의 테스트 프로그램 개발 시스템 및 방법의 실시예들이 개방형 테스터 구조뿐만 아니라, 고정형 테스터 구조(fixed tester architecture)에도 역시 적용될 수 있음을 인식할 수 있을 것이다.

상기 개방형 구조의 테스트 시스템에 관한 기재는, 동일한 양수인에 의한 미국 출원 제60/449,622호를 우선권 주장의 기초로 하는 미국 출원 제10/772,327호 "집적 회로를 시험하기 위한 방법 및 장치"에서 발견할 수 있다.

도 1은, 어떻게 신호가 생성되고 피시험 디바이스(device-under-test; DUT)에 인가되는가를 도시하는 통상적인 테스터의 일반화된 구조를 도시한다. 각 피시험 디바이스의 입력 핀은 테스트 데이터를 인가하는 드라이버 2에 접속되며, 한편 각 피시험 디바이스의 출력 핀은 비교기 4에 접속된다. 대부부의 경우에, 각 테스터 핀(채널)이 입력 핀 또는 출력 핀의 어느 하나로 동작할 수 있도록 3상 드라이버-비교기(tri-state driver-comparator)가 사용된다. 하나의 피시험 디바이스에 전용인 테스터 핀들은 집합적으로, 관련된 타이밍 생성기 6, 파형 생성기 8, 패턴 메모리 10, 타이밍 데이터 메모리 12, 파형 데이터 메모리 14 및 데이터 전송률(data rate)을 정의하는 블록 16과 함께 작업하는 테스트 사이트를 형성한다.

도 2는, 본 발명의 일 실시예에 의한 시스템 구조 100을 도시한다. 시스템 제어기(SysC) 102는 다수의 사이트 제어기(SiteCs) 104와 연결된다. 상기 시스템 제어기는 또한 관련된 파일(file)에 액세스(access)하기 위하여 네트워크에 연결될 수도 있다. 각 사이트 제어기는, 모듈 접속 인에이블러(module connection enabler) 106을 통해, 테스트 사이트 110에 위치한 하나 또는 그 이상의 테스트 모듈 108을 제어하기 위하여 연결된다. 상기 모듈 접속 인에이블러 106은 접속된 하드웨어 모듈 108의 재구성을 가능하게 하며, 또한 (패턴 데이터의 로딩(loading), 응답 데이터의 수집, 제어 공급, 등을 위한) 데이터 전송을 위한 버스로서도 기능한다. 가능한 하드웨어 구현으로서는, 전용 접속, 스위치 접속, 버스 접속, 링(ring) 접속, 및 스타(star) 접속 등이 포함된다. 상기 모듈 접속 인에이블러 106은, 예를 들어, 스위치 행렬(switch matrix)에 의하여 구현될 수도 있다. 각 테스트 사이트 110은 피시험 디바이스 112와 관련되며, 상기 피시험 디바이스는 로드 보드(loadboard) 114를 통해 대응되는 사이트의 모듈에 접속된다. 본 발명의 일 실시예에 의하면, 하나의 사이트 제어기는 복수의 피시험 디바이스 사이트들과 접속될 수 있다.

상기 시스템 제어기 102는 전체적인 시스템 관리자로서 기능한다. 그것은 사이트 제어 활동을 조직하며, 시스템 레벨의 병렬 테스트 전략을 관리하며, 또한 추가적으로 시스템 레벨의 데이터 로깅(data-logging) 및 오류 처리 지원과 함께 처리기(handler)/프로브(probe) 제어를 제공한다. 동작 설정(operational setting)에 따라, 상기 시스템 제어기 102는 상기 사이트 제어기 104의 동작과 분리된 CPU에 배치될 수도 있다. 또는, 공통의 CPU를 상기 시스템 제어기 102와 상기 사이트 제어기 104가 공유할 수도 있다. 이와 유사하게, 각 사이트 제어기 104는 그 자신의 전용 CPU(중앙 처리 장치)에 배치되거나, 하나의 CPU 내의 개별 프로세스 또는 쓰레드(thread)로서 배치될 수도 있다.

상기 시스템 구조는, 개별 시스템 컴포넌트가 반드시 분산형 시스템의 물리적 컴포넌트일 필요는 없으며, 집적된 단일 시스템의 논리적 컴포넌트로서 간주될 수도 있다는 이해를 바탕으로, 도 2에 도시된 바와 같은 분산형 시스템으로서 개념적으로 가시화될 수 있다.

도 3은, 본 발명의 일 실시예에 의한 소프트웨어 구조 200을 도시한다. 상기 소프트웨어 구조 200은, 관련 하드웨어 시스템 구성요소 102, 104, 108에 대응하여, 상기 시스템 제어기 220, 적어도 하나의 사이트 제어기 240, 및 적어도 하나의 모듈 260을 위한 구성요소를 갖는 분산형 운영 체계를 나타낸다. 상기 모듈 260에 추가하여, 상기 구조 200은 소프트웨어 중에 모듈 에뮬레이션 280을 위한 대응되는 구성요소를 포함한다.

예시적인 선택으로서, 이 플랫폼을 위한 개발 환경은 마이크로소프트의 윈도우즈를 기반으로 할 수 있다. 이러한 구조의 사용으로 인해, 프로그램과 지원 이전성에서 부수적인 이점이 있다(예를 들어, 현장의 서비스 엔지니어는 진보된 진단을 수행하기 위한 테스터 운영 시스템을 실행시키는 노트북 컴퓨터를 접속시킬 수 있다.). 그러나 대규모의 연산 집약적인 동

작(예컨대, 테스트 패턴 컴파일(compile)과 같은)을 위하여는, 관련 소프트웨어를 분산된 플랫폼들 사이에서 작업 스케줄링(job scheduling)이 가능하도록 독립적으로 실행될 수 있는 독립적 존재로서 제작할 수도 있다. 그리하여, 일괄 작업(batch job)을 위한 관련된 소프트웨어 도구들은 다수의 플랫폼 형식에서 실행될 수 있다.

예시적 선택으로서, ANSI/ISO 표준 C++이 상기 소프트웨어를 위한 네이티브 언어(native language)로서 채택될 수 있다. 물론, 제3자가 자신의 선택에 따라 다른 언어로 상기 시스템에 집적할 수 있게 하는 다수의 선택 사항(공칭 C++ 인터페이스 상층의 레이어를 제공하는)이 존재한다.

도 3은, 상기 테스터 운영 체계, (예를 들어, 테스트를 목적으로 사용자에게 공급되는) 사용자 컴포넌트 292, (예를 들어, 기초적 접속성 및 통신을 위한 소프트웨어 인프라스트럭쳐(infrastructure)로서 공급되는) 시스템 컴포넌트 294, (예를 들어, 모듈 개발기(module developer)로서 공급되는) 모듈 개발 컴포넌트 296, 및 (예를 들어, 모듈 개발자 이외의 외부 소스(external source)로서 공급되는) 외부 컴포넌트 298을 포함하는 공칭 소스(nominal source)(또는 서브-시스템으로서 집합적 개발)에 따라 구성요소의 조직에 의한 음영 처리(shading)를 도시한다.

소스 기반 조직(source-based organization)의 관점에서 볼 때, 상기 테스터 운영 체계(Tester Operating System; TOS) 인터페이스 290은 다음을 포함한다. 즉, 시스템 제어기와 사이트 제어기 간 인터페이스 222, 프레임워크 클래스 224, 사이트 제어기와 모듈 간 인터페이스 245, 프레임워크 클래스 246, 예정된 모듈 레벨 인터페이스, 백플레인 통신 라이브러리 249, 섀시 슬롯 인터페이스(chassis slot IF (Interface)) 262, 로드 보드 하드웨어 인터페이스 264, 백플레인 시뮬레이션 인터페이스 283, 로드보드 시뮬레이션 인터페이스 285, 피시험 디바이스 시뮬레이션 인터페이스 287, 피시험 디바이스의 베리로그 모델을 위한 베리로그 프로그래밍 언어 인터페이스(Verilog Programming Language Interface; PLI) 288 및 피시험 디바이스의 C/C++ 모델을 위한 C/C++ 언어 지원 289가 포함된다.

사용자 컴포넌트 292는 다음을 포함한다. 즉, 사용자 테스트 계획(user test plan) 242, 사용자 테스트 클래스(user test class) 243, 하드웨어 로드보드(hardware loadboard) 265, 피시험 디바이스 266, 피시험 디바이스 베리로그 모델 (DUT Verilog model) 293 및 피시험 디바이스 C/C++ 모델 291을 포함한다.

시스템 컴포넌트 294는 다음을 포함한다. 즉, 시스템 도구 226, 통신 라이브러리 230, 테스트 클래스 244, 백플레인 드라이버 250, 하드웨어 백플레인 261, 시뮬레이션 프레임워크 281, 백플레인 에뮬레이션 282 및 로드보드 시뮬레이션 286을 포함한다.

모듈-개발 컴포넌트 296은 다음을 포함한다. 즉, 모듈 명령 구현(module commands implementation) 248, 모듈 하드웨어 263, 및 모듈 에뮬레이션 284를 포함한다.

외부 컴포넌트 298은 외부 도구(external tools) 225를 포함한다.

상기 시스템 제어기 220은 사이트 제어기와의 인터페이스 222, 프레임워크 클래스 224, 시스템 도구 226, 외부 도구 225 및 통신 라이브러리 230을 포함한다. 상기 시스템 제어기 소프트웨어는 사용자를 위한 상호 작용의 제1차 지점이다. 그것은, 본 발명의 사이트 제어기로의 게이트웨이(gateway)와, 본 출원의 출원인에 의한 미국 특허 출원 제60/449,622호에 개시된 다중 사이트/피시험 디바이스 환경에서의 사이트 제어기의 동기화를 제공한다. 그래픽 사용자 인터페이스(graphical user interface; GUI) 기반 또는 기타의 사용자 응용 프로그램과 도구들은 상기 시스템 제어기 상에서 실행된다. 상기 시스템 제어기는 또한, 테스트 계획, 테스트 패턴 및 테스트 파라미터 파일을 포함하는 모든 테스트 계획과 관련된 정보들을 위한 저장고로서 동작할 수도 있다. 이러한 파일들을 저장하는 상기 메모리는 상기 시스템 제어기에게는 지역적인 것이거나 또는, 예를 들어 네트워크를 통해 상기 시스템 제어기에 접속되어 오프라인인 것일 수도 있다. 테스트 파라미터 파일은, 본 발명의 일 실시예에 의한 객체 지향적 환경에서의 테스트 클래스를 위한 파라미터화 데이터를 포함한다.

제3 개발자들은 표준 시스템 도구들 226에 추가하여 (또는 그 대신에) 도구들을 제공할 수도 있다. 상기 시스템 제어기 220 상의 상기 표준 인터페이스 222는 상기 도구들이 상기 테스터와 테스트 객체에 억세스하기 위하여 사용하는 인터페이스들을 포함한다. 상기 도구들(응용 프로그램들) 225, 226은 상기 테스트 및 테스터 객체의 상호 작용적이며 일괄적인 제어를 가능하게 한다. 상기 도구들은 (예를 들어 SECS/TSEM의 사용을 통해) 자동화 기능을 제공하기 위한 응용 프로그램을 포함한다.

상기 시스템 제어기 220 상에 존재하는 상기 통신 라이브러리 230은, 사용자 응용 프로그램과 테스트 프로그램에 투명한 방식으로 상기 사이트 제어기 240과 통신할 수 있는 메커니즘을 제공한다.

상기 시스템 제어기 220과 관련된 메모리에 존재하는 상기 인터페이스 222는, 상기 시스템 제어기 상에서 실행되는 프레임워크 객체와의 개방형 인터페이스를 제공한다. 상기 사이트 제어기 기반 모듈 소프트웨어가 패턴 데이터에 억세스하고 이를 검색(retrieve)할 수 있게 하는 인터페이스도 포함된다. 또한, 스크립팅 엔진(scripting engine)을 통해 상기 테스터와 테스트 컴포넌트에 억세스하고 조작할 수 있는 능력을 제공하는 스크립팅 인터페이스(scripting interface)와 함께, 응용 프로그램과 도구들이 상기 테스터와 테스트 객체들에 억세스하기 위하여 사용하는 인터페이스도 포함된다. 이것은 그들의 기능을 수행하기 위한 상호작용적이고, 일괄적이며 원격적인 응용 프로그램을 위한 공통의 메커니즘을 가능하게 한다.

상기 시스템 제어기 220과 관련된 상기 프레임워크 클래스 224는, 표준 인터페이스의 참조 구현(reference implementation)을 제공하면서 상기한 객체들과의 상호작용을 위한 메커니즘을 제공한다. 예를 들어, 본 발명의 상기 사이트 제어기 240은 기능적 테스트 객체(functional test object)를 제공한다. 상기 시스템 제어기 프레임워크 클래스는 상기 기능적 테스트 객체의 원격 시스템 제어기 기반 대행자(remote system controller-based surrogate)로서 대응하는 기능적 테스트 프록시(functional test proxy)를 제공한다. 따라서, 상기 표준 기능적 테스트 인터페이스는 시스템 제어기 220 상의 도구들에게 이용 가능하게 된다. 상기 프레임워크 클래스는 상기 호스트 시스템 제어기와 관련된 운영 체계 인터페이스를 효과적으로 제공한다. 그것은 또한, 상기 사이트 제어기로의 게이트웨이를 제공하는 소프트웨어 구성요소를 구성하며, 또한 다중 사이트/피시험 디바이스 환경에서의 상기 사이트 제어기와의 동기(synchronization)를 제공한다. 따라서, 이 계층(layer)은, 상기 통신 계층을 직접 다룰 필요 없이 사이트 제어기를 조작하고 그에 억세스하기에 적합한 본 발명의 일 실시예에 의한 객체 모델을 제공한다.

상기 사이트 제어기 240은, 사용자 테스트 계획 242, 사용자 테스트 클래스 243, 표준 테스트 클래스 244, 표준 인터페이스 245, 사이트 제어기 프레임워크 클래스 246, 모듈 고레벨 명령 인터페이스(module high level command interface)(예를 들어, 예정된 모듈 레벨 인터페이스) 247, 모듈 명령 구현 248, 백플레인 통신 라이브러리 249, 및 백플레인 드라이버 250을 주재한다(host). 바람직하게, 대부분의 테스트 기능성은 상기 사이트 제어기 104/240에 의하여 처리되며, 그리하여 상기 테스트 사이트 110의 독립적인 동작이 가능하게 된다.

테스트 계획 242는 사용자에 의하여 작성된다. 상기 계획은, C++과 같은 표준 컴퓨터 언어로 직접 작성될 수도 있고, C++ 코드를 생성할 수 있는 보다 상위 레벨의 테스트 프로그래밍 언어로 기술되어 실행가능한 테스트 프로그램으로 컴파일될 수도 있다. 테스트 프로그램의 개발에 대하여, 본 발명의 일 실시예에 의하면, 본 출원인의 테스트 프로그램 언어(Test Program Language; TPL) 컴파일러를 채용한다. 도 4를 참조하면, 상기 테스트 프로그램 컴파일러 400은, 부분적으로는, 테스트 및 관련 파라미터를 기술하는 테스트 프로그램 개발자의 소스 파일 404를 C++ 코드와 같은 객체 지향적 구조로 번역하는 번역기 섹션(translator section) 402를 포함하는 코드 생성기(code generator)로서 동작한다. 차례로, 컴파일러 섹션 406이 상기 코드를, 상기 테스터 시스템에 의하여 실행될 수 있는 테스트 프로그램을 창출하기 위하여 DLL과 같이 실행 가능한(executable) 것으로 컴파일하고 링크한다. TPL코드 생성기/번역기를 테스트 시스템에 응용하는 것이 신규한 것임에도 불구하고, 코드 생성기는 본 발명의 기술 분야에 이미 알려져 있다는 점을 주의하기 바란다. 또한, 상기 컴파일러 섹션도 본 발명의 기술 분야에 이미 알려져 있는 표준 C++ 컴파일러일 수도 있다.

상기 테스트 계획은, 상기 프레임워크 클래스 246 및/또는 상기 사이트 제어기에 관련된 표준 또는 사용자 공급 테스트 클래스 244를 이용함으로써 테스트 객체를 창출하고, 상기 표준 인터페이스 245를 사용하여 하드웨어를 구성하며, 상기 테스트 계획 흐름을 정의한다. 그것은 또한, 상기 테스트 계획을 실행하는 동안 필요한 임의의 추가적인 로직(logic)도 제공한다. 상기 테스트 계획은 소정의 기초적 서비스를 지원하며, 디버그 서비스(예컨대, 브레이크-포인팅(break-pointing))과 같은 기초적 객체들(underlying objects)의 서비스로의 인터페이스와, 기초적 프레임워크(underlying framework)와 표준 클래스로의 억세스를 제공한다.

상기 테스트 프로그램 컴파일러 400으로의 소스 코드의 입력은, 테스트 계획에 사용되는 객체들과 그들 상호간의 관계를 특정하는 테스트 계획 기술 파일(Test Plan description file)을 포함한다. 이 파일은, 표준 인터페이스 구현의 형태로 상기 사이트 제어기상에서 실행되는 C++ 코드로 번역되며, 그것은 "ITestPlan"이라고 표기될 수 있다. 이 코드는 윈도우즈 동적 링크 라이브러리(DLL)로 패키지화 될 수 있으며, 그것은 사이트 제어기상으로 로드될 수 있다. 상기 테스트 프로그램 DLL은, 상기 사이트 제어기 소프트웨어가 자신이 수록하고 있는 테스트 계획 객체를 생성하고 리턴(return)하기 위하여 사용할 수 있는 표준 기지 진입점(standard known entry point)를 구비하기 위하여 생성된다. 상기 사이트 제어기 소프트웨어는, 상기 테스트 프로그램 DLL을 그의 프로세스 공간(process space)으로 로드하고, 상기 테스트 계획 객체의 인스턴스를 창출하기 위하여 진입점 중의 하나를 사용한다. 일단 상기 테스트 계획 객체가 창출되면, 상기 사이트 제어기 소프트웨어는 상기 테스트 계획을 실행할 수 있다.

상기 사이트 제어기와 관련된 상기 프레임워크 클래스 246은, 공통의 테스트 관련 동작을 구현하는 클래스들과 메소드들의 세트(set)이다. 상기 사이트 제어기 레벨 프레임워크는, 예를 들어 전력 공급 및 핀 일렉트로닉스 계열화, 레벨 및 타이밍 조건 설정, 측정 획득, 테스트 흐름 제어 등을 위한 클래스를 포함한다. 상기 프레임워크는 또한, 실행 시간 서비스(runtime service) 및 디버깅(debugging)을 위한 메소드도 제공한다. 상기 프레임워크 객체는 표준 인터페이스의 구현을 통해 작업할 수도 있다. 예를 들어, 상기 테스터 핀 프레임워크 클래스의 구현은, 테스트 클래스가 하드웨어 모듈 핀과 상호작용하기 위하여 사용할 수도 있는 일반적인 테스터 핀 인터페이스를 구현하도록 표준화된다.

특정의 프레임워크 객체는, 상기 모듈들과 통신하기 위하여, 상기 모듈 레벨 인터페이스 247의 도움을 받아 작업하도록 구현될 수도 있다. 상기 사이트 제어기 프레임워크 클래스는, 각 사이트 제어기를 지원하는 지역 운영 체계 인터페이스로서 효과적으로 동작할 수 있다.

일반적으로, 90퍼센트 이상의 프로그램 코드가 장치 테스트를 위한 데이터이며, 상기 코드의 나머지 10퍼센트가 테스트 방법론을 실현한다. 상기 장치 테스트 데이터는 피시험 디바이스에 의존된다(예를 들어, 전력 공급 조건, 신호 전압 조건, 타이밍 조건, 등). 상기 테스트 코드는, 특정의 장치 조건을 ATE 하드웨어에 로드(load)하는 메소드와, 사용자에 특정한 객체들(예컨대, 데이터 로깅(data logging))을 실체화하는데 필요한 것들로 이루어진다. 본 발명의 일 실시예에 의한 프레임워크는, 하드웨어에 독립적인 테스트와, 사용자가 피시험 디바이스 테스트 프로그래밍의 작업을 수행할 수 있도록 하는 테스터 객체 모델을 제공한다.

테스트 코드의 재사용 가능성을 증대시키기 위하여, 상기 코드는, 어떠한 장치 특유적 데이터(예컨대, 핀 명칭, 자극 데이터(stimulus data), 등) 또는 장치-시험 특유적 데이터(예컨대, DC 유닛을 위한 조건들, 측정 핀들, 목표 핀의 개수, 패턴 파일의 명칭, 패턴 프로그램의 어드레스)로부터도 독립적으로 만들어질 수 있다. 만약 테스트를 위한 코드가 이러한 형식의 데이터와 함께 컴파일된다면, 상기 테스트 코드의 재사용 가능성은 감소될 것이다. 따라서, 본 발명의 일 실시예에 의하면, 어떠한 장치 특유적 데이터 또는 장치-시험 특유적 데이터도, 코드의 실행 시간 동안의 입력으로서, 상기 테스트 코드가 외부적으로 활용할 수 있도록 만들어질 수 있다.

본 발명의 일 실시예에 의하면, 본 명세서에서 'ITest'로 표기되는 것으로서, 표준 테스트 인터페이스의 구현인 테스트 클래스(Test Class)는 특정 형식의 시험을 위하여 테스트와 코드의 구별(따라서, 코드의 재사용 가능성)을 실현한다. 이러한 테스트 클래스는 그 자신의 별도의 인스턴스(instance)를 위한 템플리트(template)로서 간주될 수도 있으며, 상기 인스턴스들은 상호 간에 단지 장치 특유적 및/또는 장치-시험 특유적 데이터의 기반 상에서만 서로 다르다. 테스트 클래스들은 상기 테스트 계획 파일에 명기된다. 각 테스트 클래스는 전형적으로 특정 형식의 장치 테스트 또는 장치 테스트를 위한 셋업을 구현한다. 예를 들어, 본 발명의 일시예에 의하면, 피시험 디바이스의 모든 기능적 시험을 위한 기본 클래스(base class)로서, 예를 들어 기능적 테스트(FunctionalTest)와 같은 ITest의 특정적인 구현이 제공된다. 그것은, 테스트 조건 설정, 패턴 실행 및 실패한 스트로브(failed strobes)의 존재에 기초한 테스트 상태 결정과 같은 기초적인 기능성(basic functionality)을 제공한다. 다른 형식의 구현은, 본 명세서에서 'ACParametricTests' 및 'DCParametricTests'로서 표기되는 AC 및 DC 테스트 클래스들을 포함한다.

모든 테스트 형식들은 소정의 가상 메소드들(virtual methods)(예컨대, init(), preExec() 및 postExec())의 디폴트 구현들(default implementations)을 제공할 수 있다. 이러한 메소드들은 디폴트 행위(default behavior)의 오버라이드(override) 및 임의의 시험 특유적 파라미터의 설정을 위한 테스트 엔지니어의 진입점(entry points)이 된다. 그러나 커스텀 테스트 클래스들(custom test classes)도 또한 테스트 계획에 사용될 수 있다.

테스트 클래스에 의하여, 사용자는 특정 테스트의 특정 인스턴스를 위한 옵션(opetion)을 지정하는데 사용되는 파라미터들을 제공함으로써 클래스 행위(class behavior)를 구성할 수 있다. 예를 들어, 기능적 테스트는, 실행할 패턴 리스트를 지정하기 위하여 PList 및 TestConditions이라는 파라미터와, 시험을 위하여 레벨(Level) 및 타이밍 조건(Timing conditions)을 각각 취할 수 있다. (테스트 계획 기술 파일에서 서로 다른 "테스트" 블록들(blocks)을 사용함으로써) 이러한 파라미터를 위한 서로 다른 값들을 지정함으로써, 사용자는 기능적 테스트의 서로 다른 인스턴스를 생성할 수 있다. 도 5는 하나의 테스트 클래스로부터 어떻게 서로 다른 테스트 인스턴스들이 유도될 수 있는지를 도시한다. 이러한 클래스들은 C++ 코드와 같은 객체 지향적 구조에서 직접적으로 프로그램되거나, 테스트 프로그램 컴파일러가 상기 테스트 및 그들의 파라미터들을 테스트 계획 파일로부터 취하여 대응되는 C++ 코드를 생성하도록 설계될 수도 있으며, 상기 C++ 코드는 상기 테스트 프로그램을 생성하기 위하여 컴파일되고 링크될 수 있다. 템플리트 라이브러리(Template Library)는 일반 알고리즘 및 데이터 구조(generic algorithm and data structure)의 범용 라이브러리로(general-purpose library)서 채용될 수 있다. 이 라이브러리는 상기 테스터의 사용자에게 가시적일 수 있으며, 그리하여 상기 사용자는, 예를 들어, 사용자 정의 테스트 클래스를 생성하기 위하여 테스트 클래스의 구현을 수정할 수도 있다.

사용자 개발 테스트 클래스로서, 상기 시스템의 일 실시예는, 모든 테스트 클래스들이, 예컨대 ITest와 같은, 하나의 테스트 인터페이스로부터 유래한다는 점에서 그러한 테스트 클래스들을 상기 프레임워크로서 통합하는 것을 지원하며, 그리하여 상기 프레임워크는 그들을 상기 시스템 테스트 클래스들의 표준 세트와 같은 방식으로 조작할 수 있다. 사용자들은, 이러한 추가적인 설비들의 이점을 이용하기 위하여 자신의 테스트 프로그램들에 커스텀 코드(custom code)를 사용하여야 한다는 점을 이해하면서, 자신의 테스트 클래스들에 추가적인 기능성을 자유롭게 편입시킬 수 있다.

각 테스트 사이트 110은 하나 또는 그 이상의 피시험 디바이스 106의 테스트 전용이며, 테스트 모듈 112의 구성 가능한 집합을 통해 기능한다. 각 테스트 모듈 112는 특정 테스트 과제를 수행하기 위한 엔티티(entity)이다. 예를 들어, 테스트 모듈 112는 피시험 디바이스의 전력 공급기, 핀 카드, 아날로그 카드 등의 어느 것일 수 있다. 이러한 모듈식 접근 방법(modular approach)에 의하여 높은 수준의 유연성과 구성 가능성이 제공된다.

모듈 명령 구현 클래스들 248은 모듈 하드웨어 벤더(module hardware vendor)에 의하여 제공될 수 있으며, 벤더에 의하여 선택된 명령 구현 메소드에 따라 하드웨어 모듈들을 위한 모듈 레벨 인터페이스들을 구현하거나, 표준 인터페이스의 모듈 특유 구현을 제공할 수 있다. 이러한 클래스들의 외부 인터페이스들은 예정된 모듈 레벨 인터페이스 요건 및 백플레인 통신 라이브러리 요건에 의하여 정의된다. 이러한 레이어는 또한, 메소드들(함수들)과 데이터 구성요소들의 추가를 허용하기 위한 테스트 명령들의 표준 세트의 확장을 위하여 제공한다.

상기 백플레인 통신 라이브러리 249는, 백플레인 간의 표준 통신을 위한 인터페이스를 제공하며, 그리하여 상기 테스트 사이트에 접속된 모듈들과 통신하기 위하여 필요한 기능들을 제공한다. 이것은, 벤더에 특유한 모듈 소프트웨어가 대응하는 하드웨어 모듈들과 통신하기 위하여 백플레인 드라이버 250을 사용할 수 있게 한다. 상기 백플레인 통신 프로토콜은 패킷 기반 포맷(packet based format)을 사용할 수도 있다.

테스터 핀 객체들(Tester Pin objects)은 물리적인 테스터 채널들을 나타내며, 본 명세서에서 'ITesterPin'으로 표기된 테스터 핀 인터페이스(tester pin interface)로부터 유래한다. 본 발명의 일 실시예에 의한 소프트웨어 개발 키트(software development kit; SDK)는, 예정된 모듈 레벨 인터페이스인 IChannel의 형식으로 구현되는 것으로서, 테스터핀(TesterPin)으로 불릴 수도 있는 ITesterPin의 디폴트 구현(default implementation)을 제공한다. 벤더들은 IChannel의 형식으로 그들의 모듈의 기능성을 구현할 수 있다면, 상기 테스터핀을 자유롭게 활용할 수 있으며, 그렇지 않으면 그들은 자신의 모듈과 작업할 수 있도록 ITesterPin의 구현을 제공하여야만 한다.

본 발명의 상기 테스터 시스템에 의하여 제공되는 것으로서, 본 명세서에서 'IMode'로 표기되는 상기 표준 모듈 인터페이스는, 포괄적으로 벤더의 하드웨어 모듈을 나타낸다. 상기 시스템을 위한 벤더 제공 모듈 특유의 소프트웨어는 동적 링크 라이브러리(dynamic link library: DLL)와 같은 실행 가능한(executable)의 형식으로 제공될 수 있다. 하나의 벤더로부터의 각 모듈 형식을 위한 소프트웨어는 하나의 DLL로 캡슐화(encapsulated)될 수 있다. 그러한 각 소프트웨어 모듈은, 상기 모듈 인터페이스 명령을 위한 벤더 특유의 구현을 제공할 책임이 있으며, 상기 모듈 인터페이스 명령은 모듈 소프트웨어 개발을 위한 API를 포함한다.

상기 모듈 인터페이스 명령에는 두 가지 국면이 존재한다. 즉, 첫째로, 그들은 사용자가 상기 시스템의 특정 하드웨어 모듈과 (간접적으로) 통신하기 위한 인터페이스로서 복무하며, 또한, 둘째로, 그들은 제3자가 자기 자신의 모듈들을 상기 사이트 제어기 레벨 프레임워크로서 통합하기 위하여 이용할 수 있는 인터페이스를 제공한다. 그리하여, 상기 프레임워크에 의하여 제공되는 상기 모듈 인터페이스 명령은 두 개의 형식으로 구분된다. 즉,

첫째로, 그리고 가장 명백하게는, 상기 프레임워크 인터페이스를 통해 상기 사용자에게 노출되는 "명령들(commands)"이다. 그리하여, 예를 들어, 테스터 핀 인터페이스(ITesterPin)는 레벨과 타이밍 값을 취하고 설정할 메소드들을 제공하며, 한편 전력 공급기 인터페이스(IPowerSupply)는 전력을 공급하거나 공급하지 않기 위한 메소드들을 제공한다.

나아가, 상기 프레임워크는 예정된 모듈 레벨 인터페이스의 특별한 카테고리를 제공하며, 상기 예정된 모듈 레벨 인터페이스는 상기 모듈들과 통신하는 데에 사용될 수 있다. 이들은 벤더 모듈들과 통신하기 위하여 프레임워크 클래스들(즉, 프레임워크 인터페이스들의 "표준" 구현)에 의하여 사용되는 인터페이스들이다.

그러나 상기 두 번째 국면, 즉 모듈 레벨 인터페이스의 사용은 선택적이다. 그렇게 하는 것의 이점은, 벤더들이 ITesterPin 및 IPowerSupply와 같은 클래스들의 구현을 이용할 수 있다는 것이다. 그러나 만약 이러한 인터페이스들이 상기 벤더들

에게 적절하지 않다면, 그들은 그들의 상기 프레임워크 인터페이스의 커스텀 구현(예를 들어, ITesterPin, IPowerSupply, 등의 벤더 구현)을 제공하는 것을 선택할 수도 있다. 그리고나서 이들은 그들의 하드웨어에 적합한 커스텀 기능성을 제공할 수 있다.

이러한 개방형 구조를 배경으로 하여, 본 발명의 테스트 프로그램 개발 시스템에 관하여 이하에서 보다 상세히 설명한다. 이하의 섹션 A에서는 테스트 프로그램이 사용될 테스트 환경을 기술하기 위한 규칙을 기술하며, 섹션 B에서는 테스트 프로그램 개발을 위한 방법과 규칙을 기술하며, 섹션 C에서는 테스트 계획을 개발하고 상기 테스트 프로그램의 주요 구조를 정의하기 위한 방법과 규칙을 특정하며, 섹션 D에서는, 개방형 구조의 테스트 시스템에서 테스트 프로그램을 실행시키는 방법에 관하여 기술하며, 섹션 E에서는 테스트 패턴에 관한 방법과 규칙을 기술하고, 섹션 F에서는 상기 테스트 패턴의 타이밍을 기술하는 규칙에 관하여 기술하며, 그리고, 섹션 G에서는 전체적인 테스터의 동작에 관한 규칙을 기술한다.

A. 컴포넌트

테스트 환경은, 테스터를 준비하고 그것이 한 세트의 시험을 실행시키도록 준비하기 위하여 필요한 조건들을 규정하는 파일의 한 세트를 포함한다. 상기 테스트 환경은 바람직하게 다음의 파일들을 포함한다. 즉,

1. 테스터 리소스 정의: 상기 개방형 구조의 테스트 시스템에서 활용 가능한 테스터 리소스의 형식(그리고, 그러한 리소스를 위하여 지원되는 파라미터들)의 규정에 대한 것.

2. 테스터 구성: 사이트 제어기, 사이트 및 대응되는 매핑(mapping)에 대한 것.

3. 모듈 구성: 각 사이트의 하드웨어 모듈의 규정에 대한 것.

4. 핀 기술(pin descriptions): 신호 핀, 전력 공급기 등과 같은 피시험 디바이스의 핀들의 명명을 위한 것과 핀 그룹을 기술하기 위한 것.

5. 소켓: 피시험 디바이스의 핀에 대한 테스터 핀 할당(DUT pin-to-tester pin assignment)의 규정에 대한 것.

6. 핀 옵션(pin option): 핀에 대한 특별한 옵션 또는 모드의 규정에 대한 것.

7. 패턴 리스트: 테스트 패턴 및 그들의 시퀀스의 규정에 대한 것.

8. 패턴: 테스트 벡터(test vector)의 규정에 대한 것.

상기한 것들 중에서, 1번 내지 3번 항목은 구성 관리 데이터베이스(configuration management database; CMD)로부터의 정보를 가지고 설치 및 구성 파일(installation and configuration files; ICF)에 의하여 창출되며, 잘 알려진 위치에서 활용가능하게 되며, 한편 4번 내지 8번 항목은 사용자가 특정한다. 본 섹션에서는 상기 1번 내지 6번 항목에 관하여 기술하며, 7번 및 8번 항목에 관하여는 섹션 E에서 보다 상세히 기술한다. 바람직하게, 이러한 컴포넌트들 각각의 개발에 특별한 방법과 규칙들이 사용되며, 이들 방법과 규칙들에 관하여 예를 들며 본 섹션에서 상세히 기술한다.

A1. 리소스 정의(Resource Definition)

각 하드웨어 모듈은, 상기 테스트 시스템에 의하여 사용되기 위하여 하나 또는 그 이상의 형식의 하드웨어 리소스(간략히 '리소스'이라 함)를 제공한다. 상기 테스터 리소스 정의는, 바람직하게 사용가능한 리소스 형식에 대한 리소스 명칭의 세트, 및 각 특정 리소스 형식에 관련된 파라미터의 명칭과 형식의 세트를 선언하는 데에 사용된다. 예를 들어, 'dpin'이라는 리소스의 명칭은 디지털 테스터 핀을 가리키는 데에 사용된다. 이러한 리소스들은 VIL(입력 저전압에 대하여), VIH(입력 고전압에 대하여), VOL(출력 저전압에 대하여), VOH(출력 고전압에 대하여) 등의 파라미터를 갖는다. 리소스 정의 파일은 ".rsc"라는 확장자를 가질 것이다. 이하에 개시된 내용은 일부 테스터 리소스를 수록하고 있는 리소스 정의의 예이다.

```
#

#File Resources.rsc

#
```

```
Version 0.1.2;

ResourceDefs

{

#디지털 핀들

dpin

{

#입력 핀에 대한 저전압 및 고전압

voltage VIL, VIH;

#출력 핀에 대한 저전압 및 고전압

voltage VOL, VOH

}

#전력 공급기

dps

{

#

#PRE_WAIT는 전압이 그의 최종값에 도달한 후

# 패턴 생성을 시작할 때까지의 시간을 규정한다.

# 시스템이 대기하여야 할 실제 시간은 시스템이

# 규정하는 작은 범위이다.

# PRE_WAIT-delta <= 실제 <= PRE_WAIT+ delta

#PRE_WAIT_MIN은 전압이 그의 최종값에 도달한 후

# 패턴 생성을 시작할 때까지 대기하여야 할 최소값이다.

# 그것은 시스템이 규정하는 범위이다.

# PRE_WAIT_MIN-delta <= 실제 <= PRE_WAIT_MIN+ delta

#

#POST_WAIT는 패턴 생성이 종료된 후 전력이 차단될

# 때까지 대기하여야 할 시간을 규정한다.
```

시스템이 대기하여야 할 실제 시간은

시스템이 규정하는 작은 범위이다.

POST_WAIT-delta <= 실제 <= POST_WAIT+ delta

#POST_WAIT_MIN은 패턴 생성이 종료된 후에

전력이 차단될 때까지 대기하여야 할 최소값이다.

시스템이 대기하여야 할 실제 시간은

시스템이 규정하는 작은 범위이다.

POST_WAIT_MIN-delta <= 실제 <= POST_WAIT_MIN+ delta

#

time PRE_WAIT;

time PRE_WAIT_MIN;

time POST_WAIT;

time POST_WAIT_MIN;

#전압

voltage VCC;

}

}

리소스 파라미터(전압 또는 시간과 같은)의 형식이 표준 엔지니어링 단위라는 점을 주의하라. 다른 파라미터 규정을 선호하는 벤더 공급 전용 리소스는 그들 자신의 정의 파일을 제공하여야만 하다.

리소스 정의의 구조

이하에 개시된 내용은, 본 발명의 일 실시예에 의한 리소스 정의 파일의 구조이다. 즉,

resource-file:

version-info resource-defs

version-info:

Version *version-identifier*;

resource-defs:

ResourceDefs{*resource-def-list*}

resource-def-list;

resource-def

resource-def-list resource-def

resource-def:

resource-name{*resource-params-decl-list*}

resource-params-decl-list:

resource-params-decl

resource-params-decl-list resource-params-decl

resource-params-decl:

elementary-type-name resource-param-name-list;

resource-params-name-list:

resource-param-name

resource-param-name-list, resource-param-name

상기에 있어서, 정의되지 않은 비종단자(non-terminal)는 이하와 같이 규정된다. 즉,

1. 버전-식별자(version-identifier): 세트 [0-9a-zA-Z]로부터의 하나 또는 그 이상의 문자의 시퀀스. 버전 번호를 나타냄.

2. 리소스-명칭(resource-name): 세트 [a-zA-Z_0-9]로부터의 하나 또는 그 이상의 문자의 시퀀스로서 숫자로 시작되지 않는 것. dpin 또는 dps와 같이, 리소스의 명칭을 나타냄.

3. 기초-형식-명칭(elementary-type-name): 세트 [a-zA-Z_0-9]로부터의 하나 또는 그 이상의 문자의 시퀀스로서 숫자로 시작되지 않는 것. Voltage(cf.)와 같이, 기초적 형식의 명칭을 나타냄.

4. 리소스-파라미터-명칭(resource-param-name): 세트 [a-zA-Z_0-9]로부터의 하나 또는 그 이상의 문자의 시퀀스로서 숫자로 시작되지 않는 것. VIL과 같이, 리소스 파라미터의 명칭을 나타냄.

A2. 테스터 구성

상기 테스터 구성은, 바람직하게, 특정의 시스템 구성에서의 사이트 제어기 및 상기 사이트 제어기들과 상기 스위치 행렬의 입력 포트들의 접속을 나열하는 데에 사용되는 규칙의 세트이다. 본 발명의 일 실시예에 의한 구조에서는, 하나의 사이트 제어기가 하나의 스위치 행렬의 입력 포트에 접속될 수 있다. 그리하여, 이러한 상황에서, 상기 스위치 행렬 접속은 상기 시스템에 대한 암시적인 식별자로서 기능한다(다른 구성도 가능하다.). 이하는 전형적인 테스터 구성의 예이다.

#

테스터 구성, Sys.cfg

#

Version 1.2.5;

SysConfig

{

#첫번째 필드는 상기 사이트 제어기 머신(Site Controller machine)

#의 호스트 명칭(hostname)이다. 그것은 점과 숫자의 IP 어드레스 또는

#도메인 한정 호스트 명칭(domain-qualified hostname)의

#어느 것으로서 특정될 수 있다.

#

#두번째 필드는 스위치 행렬의 입력 포트 번호인데,

#그것은 암시적으로 그에 접속된 사이트 제어기의

#식별자로서 기능한다.

#

zeus.olypus.deities.org 2;

127.0.0.2 4;

127.0.0.0 1;#SITEC-1

127.0.0.3 3;

}

특정의 테스트 플로어 시스템(test-floor system)을 위한 시스템 구성은 상기 시스템 프로파일의 일부이며, 시스템 구성 파일인 Sys.cfg로서 활용 가능하게 된다. 본 발명의 일 실시예에 의하면, 1번 포트에 접속된 사이트 제어기(상기 예에서는 "127.0.0.0")는 특별한 상태를 향유할 수 있는데, 여기서 그것은 단독으로 스위치 행렬을 구성한다는 점을 주의하여야 한다. 이러한 "특별한(special)" 사이트 제어기는 "SITEC-1"이라 불린다. 또한, 상기 사이트 제어기는 내부 네트워크에 의하여 상기 시스템 제어기에 접속될 수도 있기 때문에, 이 예에서의 상기 사이트 제어기 어드레스는 IP 어드레스라는 점도 또한 주의하여야 한다. 역으로, 상기 시스템 제어기는 패턴 데이터와 같은 파일들에 억세스하기 위하여 외부 네트워크에 접속될 수도 있다.

상기 테스터 구성을 위한 구조

이하는 본 발명의 일 실시예에 의한 시스템 구성 파일을 위한 구조를 개시한다. 즉,

system-config-file:

version-info system-config

version-info:

Version *version-identifier;*

system-config:

SysConfig {*site-controller-connection-list*}

site-controller-connection-list:

site-controller-connection

site-controller-connection-list site-controller-connection

site-controller-connection:

site-controller-hostname input-port;

site-controller-hostname:

ip-address

domain-qualified-hostname

ip-address:

octet.octet.octet.octet

domain-qualified-hostname:

name

domain-qualified-hostname.name

상기에 있어서, 정의되지 않은 비종단자는 이하와 같이 규정된다. 즉,

1. 버전-식별자(version-identifier): 세트 [0-9a-zA-Z]로부터의 하나 또는 그 이상의 문자의 시퀀스. 버전 번호를 나타냄.

2. 옥텟(octet): (십진 표기법의) 0 내지 255 사이의 음이 아닌 정수.

3. 명칭(name): 세트 [a-zA-Z_0-9]로부터의 하나 또는 그 이상의 문자의 시퀀스로서 숫자로 시작되지 않는 것. 도메인 한정 호스트 명칭의 명칭 부분을 나타냄.

4. 입력-포트(input-port): 십진 표기법의 음이 아닌 정수

A3. 모듈 구성

상기 모듈 구성은, 예를 들어 시스템 섀시에서의 각 모듈의 물리적 위치 및 형식과 같은 테스터의 물리적 구성의 사양을 말한다. 이것은 상기 테스터 버스 구성의 동적 성격에 의하여 요구되는 것으로서, 그것으로 인하여 테스터 버스 어드레스의 물리적 슬롯 위치로의 매핑이 가능하게 된다. 이러한 정보는, 시스템 구성을 유효화하기 위한 시스템 부트-업(boot-up) 시간에 발생하는 하드웨어 발견 프로세스(hardware discovery process)를 가능하게 한다. 상기 스위치 행렬의 각 출력 포트는 물리적 슬롯을 정의하며, 그것은 바람직하게 하나의 하드웨어 모듈에 의하여 점유된다. 이하에 개시된 내용은, 본 발명의 일 실시예에 의한 Modules.cfg라는 파일에 규정된 모듈 구성의 일예이다. 즉,

#

```
#모듈 구성 파일, Modules.cfg
#
Version 0.0.1;
ModuleConfig
{
#
#슬롯 1 내지 12 및 32 내지 48에 부착된 모듈 형식에
#관한 정보를 제공하는 구성 정의.
#하나의 모듈이, 단일 형식의 리소스를
#하나 이상 제공할 수 있음을 주의할 것.
#
Slot 1-12, 32-48 #이하의 구성 정의를
#사용하는 스위치
#행렬 출력 포트.
{
VendorID 1; #벤더 코드를 정의.
ModuleID 1; #벤더-정의 식별 코드.
ModuleDriver mod1.dll; #모듈 소프트웨어.
#
#dpin으로 명명된 리소스가 디지털 데이터를
#위한 채널을 규정함. dpin이라는 명칭은
#키워드가 아님. 그것은 단순히
#하드웨어 리소스의 명칭이며,
#리소스 정의 파일로부터 획득될 수 있음.
#
Resource dpin
{
```

MaxAvailble 32; #1번 내지 32번 리소스 유닛.

}

Resource analog

{

MaxAvailble 16; #1번 내지 16번 리소스 유닛.

Disabled 1-8; #1번 내지 8번 리소스 활용 불가능.

#따라서, 활성화된 것은 9번 내지 16번

}

}

#

#슬롯 16 내지 30, 50 및 61 내지 64에 부착된 모듈 형식에

#관한 정보를 제공하는 구성 정의.

#

Slot 16-30, 50, 61-64

{

Resource dpin

{

MaxAvailble 32; #활용 가능한 최대 리소스 유닛.

Disabled 3,30-32; #활용 불가능한 리소스.

}

ModuleDriver "module two.dll";

VendorID 2;

ModuleID 2;

}

#

#슬롯 16 내지 30, 50 및 61 내지 64에 부착된 모듈 형식에

#관한 정보를 제공하는 구성 정의.

```
#
Slot 16-30, 50, 61-64
{
Resource dpin
{
MaxAvailble 32; #활용 가능한 최대 리소스 유닛.
Disabled 3,30-32; #활용 불가능한 리소스.
}
ModuleDriver "module two.dll";
VendorID 2;
ModuleID 2;
}
#
#슬롯 65 및 66에 부착된 모듈 형식에
#관한 정보를 제공하는 구성 정의.
#
Slot 65-66
{
ModuleID 4; #8개의 제공자를 가진 DPS 모듈.
ModuleDriver mod4.dll;
VendorID 1;
#
#dps라는 리소스 형식은 장치 전력 공급기를
#위한 리소스 유닛을 규정함.
#
Resource dps
{
```

MaxAvailble 4;

Disabled 1;

}

}

}

상기한 바와 같이, 본 발명의 일 실시예에 의하면, 슬롯은, 상기 스위치 행렬의 출력 포트와 같은, 하드웨어 모듈이 접속될 수 있는 코넥터(connector)를 가리킨다. 각 구성 정의는 하나 또는 그 이상의 슬롯에 부착될 수 있는 모듈에 관한 정보를 제공한다. 상기 구성 정의에 규정된 'VendorID'는 특정 벤더에 관련된 고유한 식별자이다. 상기 'ModuleID'는 이 벤더에 의하여 제공되는 모듈의 형식을 가리킨다. 특정의 테스터 구성에는 동일한 ModuleID를 갖는 몇 개의 인스턴스들이 있을 수 있다. 상기 'ModuleDriver'는 상기 모듈을 서비스하기 위한 벤더 공급 DLL을 가리킨다. 마지막으로, 상기 'Resource'는 이 모듈에 의하여 서비스를 받는 유닛을 가리키며, 이 리소스 형식에 대한 명칭을 제공하는데, 그 리소스 명칭은 상기 리소스 정의 파일로부터 획득될 수 있다.

상기 예에서는 하나의 모듈 구성 파일에서 세 개의 구성 블록이 기술되어 있다. 한 구현에 의하면, 상기 첫 번째 구성 블록, 즉 슬롯 1 내지 12 및 32 내지 48은 벤더 1에 의하여 생산된 모듈에 의하여 서비스되고 있다. 이 벤더는 상기 모듈(식별자 "1"은 이 모듈 형식을 가리키기 위한 것이다.), 및 상기 모듈을 제어하기 위한 모듈 드라이버 라이브러리슬 제공한다. 이 모듈은 두 가지 형식의 리소스 유닛을 제공할 수 있는데, 그 하나는 "dpin"이라는 리소스 명칭으로 표기된 것으로서, 바람직하게 전체 개수 32개의 리소스 유닛들(즉, "채널들")을 가지며 이들 모두가 활용 가능하며, 다른 하나는 "analog"라는 리소스 명칭으로 표기된 것으로서, 전체 개수 16개의 리소스 유닛들을 가지며, 이들 중 단 9 내지 16번만이 활용 가능하다. 두 번째 및 세 번째 구성 블록들도 상기 첫 번째 구성과 유사한 방식으로 규정되어 있다.

채널들이 "활용 불가능"이라고 표기되도록 허용하는 규정(provision)은, 다른 상태에서는 여전히 기능할 수 있는 모듈들 상의 결함있는 리소스 유닛들의 식별을 가능하게 한다는 점을 주의하여야 한다. 또한, 구성 블록은 하나 또는 그 이상의 슬롯 식별자를 구비할 수 있음을 주의하여야 한다. 하나의 블록이 하나 이상의 슬롯 식별자를 가지는 경우, 상기 식별된 슬롯은 모조(clone)되었다고 한다.

상기 모듈 구성 파일, 즉 Modules.cfg는 (사용자에 의하여 제공되는 테스트 플로어에 특유한 정보를 가지고) 상기 ICM (installation configuration management system; 설치 구성 관리 시스템)에 의하여 시스템 프로파일(system profile)의 일부로서 창출되며, 잘 알려진 위치에서 활용 가능하게 되어 있다. 상기 ICM은, 예컨대 시스템 제어기상에서와 같이, 상기 테스트 시스템에 대하여 지역적일 수 있거나, 또는 상기 시스템 제어기가 접속된 네트워크의 다른 곳에 위치할 수도 있는 유틸리티이다. 상기 ICM은 상기 CMD(configuration management database; 구성 관리 데이터베이스)를 관리하며, 전형적으로 하드웨어의 변경에 따라 상기 시스템 구성으로 업데이트 된다. ICM은 사용자가 예를 들어 사이트 제어기 및 모듈과 같은 상기 시스템을 구성할 수 있게 한다. 상기 CMD는 상기 구성을 저장하는 데이터베이스이다. 실제의 테스터 구성/동작에 대하여, ICM은, 예를 들어 모듈 구성과 같은 구성 파일과 기타 파일들을 생성하며, 그들 및, 예를 들어 특정 모듈의 DLL과 같이, 그들에 관련된 파일들을 상기 테스터상으로 복사한다.

모듈 구성을 위한 구조

이하는 본 발명의 바람직한 실시예에 의한 모듈 구성 구조이다. 즉,

file-contents:

version-info module-config-def

version-info:

Version *version-identifier*;

module-config-def:

ModuleConfig{*slot-entry-list*}

slot-entry-list:

slot-entry

slot-entry-list slot-entry

slot-entry:

slot *positive-integer-list*{*slot-info*}

slot-info:

required-config-list

required-config-list:

required-config

required-config-list required-config

required-config:

VendorID *id-code*;

ModuleID *id-code*;

ModuleDriver *file-name*;

Resource *resource-name*{*max-spec disabled-spec*$_{opt}$}

max-spec:

MaxAvailable *positive-integer*;

disabled-spec:

Disabled *positive-integer-list*;

positive-integer-list:

positive-integer-list-entry

positive-integer-list, positive-integer-list-entry

positive-integer-list-entry:

positive-integer

positive-integer-number-range

positive-integer-number-range:

positive-integer - pos-integer

상기에 있어서, 정의되지 않은 비종단자는 이하와 같이 규정된다. 즉,

1. 버전-식별자(version-identifier): 세트 [0-9a-zA-Z]로부터의 하나 또는 그 이상의 문자의 시퀀스로서, 첫번째 문자는 반드시 세트 [0-9]로부터 채택되어야 함.

2. 양의-정수(positive-integer): 세트 [0-9]로부터의 하나 또는 그 이상의 문자의 시퀀스로서, 0으로 시작되지 않는 것.

3. 식별-코드(id-code): 세트 [a-zA-Z_0-9]로부터의 하나 또는 그 이상의 문자의 시퀀스.

4. 리소스-명칭(resource-name): 세트 [a-zA-Z_0-9]로부터의 하나 또는 그 이상의 문자의 시퀀스로서, 첫번째 문자는 반드시 세트 [a-zA-Z]로부터 채택되어야 함.

주석(comment)도 지원된다. 즉, 주석은 '#' 문자로 시작되며, 그 행의 끝까지 연장될 수 있다.

A4. 핀 기술(Pin Descriptions)

피시험 디바이스의 핀 기술은 핀 기술 파일(Pin Description file)을 사용하여 기술된다. 사용자는 핀 기술 파일에서 피시험 디바이스의 핀의 기술을 활용 가능하게 하고, 상기 핀 기술 파일은 .pin이라는 확장자를 갖는다. 이것은 평범한 텍스트 파일(plain text file)이며, 적어도 다음의 것들을 포함한다. 즉, 피시험 디바이스의 핀 명칭들의 리스트, 및 명명된 핀 그룹(named pin group)들의 초기 정의(initial definition)으로서, 상기 명명된 핀 그룹은 정의된 피시험 디바이스의 핀 명칭들을 사용한다("초기"라는 것은 예를 들어 프로그래밍적으로 이후에 수정되거나 추가될 수 있기 때문이다.).

상기 테스트 계획의 기술로부터 이 데이터 규정을 분리시킴으로써, 상기 피시험 디바이스의 핀 정의를 일반적으로 재사용할 수 있게 되며, 프로세스가 특정의 테스트 계획에 구속되도록 하지 않으면서 상기 패턴 컴파일러가 상기 핀 기술 파일로부터 (벡터 규정에 사용되는 핀 명칭들로의 참조를 해결하기 위하여 요구되는) 핀 명칭들을 도출할 수 있게 한다.

이하에 개시된 내용은 핀 기술 파일이다. 즉,

#

#핀 기술 파일, myDUT.pin.

#

#이것은 묵시적으로 상기 리소스 구성 파일,

#Resources.rsc를 인입한다는 점을 주의할 것.

#

Version 1.1.3a;

PinDescription

{

Resource dpin

```
{
A0;
A1;
A2;
A3;
A4;
#이 구문(syntax)은 "ABUS[1]" 및 "ABUS[2]"라는 명칭으로 확장
된다.
ABUS[1:2];
A5;
BBUS[1:8];
DIR;
CLK;
Group Grp1
{
DIR, CLK, A0, A1, A2, A3, A4, BBUS[1:4]
}
Group Grp2
{
A5,
#
#다음 행은 "DIR, A1, A2, A4, A5, BBUS[1:4]":로
#확장될 것이다:
Grp1-CLK-A0-A3-BBUS[1]-BBUS[3:4]+ A5,
BBUS[5:8]
}
}
```

```
Resource dps
{
vcc1;
vcc2;
vcc3;
Group PSG
{
vcc1, vcc2
}
}
}
```

상기 피시험 디바이스 핀과 핀 그룹 정의는, 컴파일러가 핀 및 핀 그룹 정의를 레벨(Level) 등의 허용 가능한 파라미터 설정과 상호 관련시킬 수 있도록, 리소스 형식 블록으로 캡슐화된다는 점을 주의하여야 한다.

핀 기술에 관한 이하의 점을 주의하여야 한다. 즉,

1. 핀 그룹 및 핀들은 동일한 명칭 공간(namespace)을 공유하며 전체적(global)(예컨대 테스트 계획) 범위(scope)를 갖는다. 이러한 명칭들이 전체적 범위를 갖는 점의 결과 중 하나는, 서로 다른 리소스 블록에서 선언되더라도 핀들 및 핀 그룹들이 중복된 명칭을 갖지 못한다는 것이다.

2. 적어도 하나의 리소스 정의가 상기 핀 기술 파일에서 요구된다.

3. 적어도 하나의 핀 명칭이 각 리소스에서 정의되어 있어야만 한다.

4. 핀 및 그룹의 명칭들은 리소스의 경계 내에서는 고유할 것이 요구된다.

5. 동일한 핀 또는 그룹의 명칭이 두개 또는 그 이상의 리소스에 대하여 정의될 수 있다. 그러나, 동일한 리소스 내에서의 중복은 무시된다.

6. 하나의 그룹 정의에 나타나는 모든 핀 명칭과 그룹 명칭은 그 리소스 내에서 이미 정의되어 있어야만 한다.

7. 만약 주어진다면, 그룹 정의는 적어도 하나의 핀 명칭 또는 그룹 명칭을 구비하여야만 한다(즉, 그룹 정의는 공백일 수는 없다.).

8. 핀 그룹의 정의는 이전에 정의된 핀 그룹에 대한 참조를 포함할 수 있다.

9. 핀 그룹의 정의는 이전에 정의된 핀 및/또는 핀 그룹의 더하기 및 빼기와 같은 세트 동작(set operation)을 포함할 수 있다.

핀 기술을 위한 구조(Structure for the Pin Description)

이하에 개시된 내용은 본 발명의 일 실시예에 의한 핀 기술을 위한 구조를 개시한다. 즉,

pin-description-file:

version-info pin-description

version-info:

Version *version-identifier*;

pin-description:

PinDescription{*resource-pins-def-list*}

resource-pins-def-list:

resource-pins-def

resource-pins-def-list resource-pins-def

resource-pins-def:

Resource *resource-name*{*pin-or-pin-group-def-list*}

pin-or-pin-group-def-list:

pin-or-pin-group-def-list

pin-or-pin-group-def-list pin-or-pin-group-def

pindef-or-pin-groupdef:

pindef;

pin-groupdef

pindef:

pin-name

pin-name[*index:index*]

pin-groupdef

Group *pin-group-name*{*pin-group-def-item-list*}

pin-group-def-item-list:

pin-def

pin-group-def-item-list, pin-def

상기에 있어서, 정의되지 않은 비종단자(non-terminal)는 이하와 같이 규정된다. 즉,

1. 버전-식별자(version-identifier): 세트 [0-9a-zA-Z]로부터의 하나 또는 그 이상의 문자의 시퀀스. 버전 번호를 나타냄.

2. 리소스-명칭(resource-name): 세트 [a-zA-Z_0-9]로부터의 하나 또는 그 이상의 문자의 시퀀스로서 숫자로 시작되지 않는 것. dpin 또는 dps와 같이, 리소스의 명칭을 나타냄.

3. 핀-명칭(pin-name): 세트 [a-zA-Z_0-9]로부터의 하나 또는 그 이상의 문자의 시퀀스로서 숫자로 시작되지 않는 것. A0와 같이, 핀의 명칭을 나타냄.

4. 핀-그룹-명칭(pin-group-name): 세트 [a-zA-Z_0-9]로부터의 하나 또는 그 이상의 문자의 시퀀스로서 숫자로 시작되지 않는 것. ABUS와 같이, 핀 그룹의 명칭을 나타냄.

5. 인덱스(index): 음이 아닌 정수. 관련된 핀들의 그룹의 하한 또는 상한을 나타냄.

A5. 소켓(The Socket)

상기 소켓은 피시험 디바이스의 핀 명칭과 물리적 테스터 핀(채널) 할당 사이의 매핑을 규정한다(상기 물리적 테스터 채널 번호는 모듈 구성 파일에서 정의된다.). 서로 다른 소켓이 서로 다른 피시험 디바이스 패키지 및 서로다른 로드보드 구성 등을 지원하기 위하여 사용될 수 있음을 주의하여야 한다. 다중 피시험 디바이스 시스템에 있어서, 피시험 디바이스/채널 할당에 대한 상기 소켓 정의는 기초적 소켓을 복수의 사이트에 "모조(cloning)"하는 것을 지원한다. 그러나, 서로 다른 소켓들(즉, 동일한 논리적 핀에 대한 서로 다른 물리적 매핑)은 사이트 모듈 분할(site module partition)을 존중하여야 한다. 그리하여, 피시험 디바이스 핀을 테스터 채널 할당에 제공하는 것에 추가하여, 상기 소켓은 또한 상기 사이트 분할하기를 효과적으로 정의하여야 한다. 그러므로, 소켓 파일은 몇몇 개별적 사이트 소켓에 대한 정의를 포함할 수 있어야 한다. 이하에 개시된 내용은 세 개의 피시험 디바이스 사이트를 정의하는 소켓 파일의 샘플이다. 즉,

Version 1.1.3

SocketDef

{

DUTType CHIP3

{

PinDescription dutP3.pin; #CHIP3에 대한 핀 기술 파일

DUT 2 #전 규정 구문(full-specification syntax)를 사용

{

SiteController 1; #스위치 행렬 입력 포트

Resource dpin

{

#

#CLK 핀은 dpin 리소스, 슬롯 2,

#리소스 유닛(채널) 13에 할당됨.

```
#
CLK 2.13;
#
#DIR 핀은 dpin 리소스, 슬롯 5,
#리소스 유닛 15에 할당됨.
DIR 5.15;
#
#이하의 문장은
# BBUS[7] 5.4
# BBUS[6] 5.5
# BBUS[5] 5.6
#으로 확장됨.
#따라서, 예를 들어, 핀 시퀀스 BBUS[7], BBUS[6],
#BBUS[5]는 동일한 슬롯 5와, 리소스 유닛들 4, 5 및
#6에 각각 할당됨.
#
BBUS[7:5] 5.[4:6];
BBUS[1:4] 7.[21:18];
BBUS[8] 9.16;
}
Resource dps
{
#
#V1 핀은 dps 리소스, 슬롯 1,
#리소스 유닛(채널) 1에 할당됨.
#
VCC1 1.1;
```

```
#
#V1 핀은 dps 리소스, 슬롯 1,
#리소스 유닛(채널) 2에 할당됨.
#
VCC2 1.2;
}
} #End DUT 2
DUT 1 #이것은 상기 DUT 2로부터 "모조"된 것임.
{
SiteController 1; #DUT 2와 동일한 사이트 제어기
Resource dpin
{
SlotOffset 1; #슬롯을 위한 오프셋 값
}
Resource dps
{
SlotOffset 10; #슬롯을 위한 오프셋 값
}
#
#상기한 오프셋 구문은 상기 슬롯/리소스 유닛의 할당이,
#상기 SlotOffset 값에 의한 슬롯 오프셋으로
#이 DUTType, 즉 피시험 디바이스 2에 대하여 정의된
#상기 제1 피시험 디바이스로부터 "모조"된 것이라는
#것을 나타낸다.
#
#피시험 디바이스 2에 대한 dpin 리소스 유닛의 정의를
#살펴 보면, CLK는 슬롯 2에 한정된다.
```

#그리하여, 현재의 피시험 디바이스에 대하여는

#CLK는 슬롯 2+ 1=3에 한정된다.

#

#오프셋 할당에 의한 실제의 새로운 한정(binding)은

#이하의 표에 개시된 바와 같다. 즉,

#---

Pin Resource RUnit Slot

#---

CLK dpin 13 2+ 1=3

DIR dpin 15 5+ 1=6

BBUS[8] dpin 16 9+ 1=10

VCC1 dps 1 1+ 10=11

VCC2 dps 2 1+ 10=11

#

}#End DUT 1

} # End DUTType CHIP3

DUTType 74LS245

{

PinDescription dutLS.pin;

DUT 3 **disabled** #이 피시험 디바이스 사이트는 활용 불가능하며,

#무시됨.

{

...

}

} # End DUTType 74LS245

} # End SocketDef

소켓 파일에 관하여는 이하의 점을 주의하여야 한다. 즉,

1. 상기 소켓 파일은, 주어진 DUT 형식(상기 예시한 핀 기술에 대한 규정을 참조하라.)에 대하여 모듈 구성 파일 및 사용자의 핀 기술 파일 모두로부터의 정보를 사용한다. 상기 모듈 구성 정보는 상기 소켓 파일 컴파일러에 대하여 묵시적으로 활용 가능하게 되어 있다. 상기 소켓 컴파일러는, 상기 패턴 컴파일러에 의하여 사용되는 테스터 핀과 피시험 디바이스 핀의 매핑을 설정하기 위하여, 테스터 채널 매핑에 대한 상기 소켓 피시험 디바이스의 명칭, 및 모듈 구성과 핀 기술 파일을 독출하고 분석하는 상기 패턴 컴파일러의 서브 파트(sub-part)이다.

2. 피시험 디바이스의 형식마다 적어도 하나의 피시험 디바이스 사이트 정의가 요구되며, 그것은 슬롯 오프셋 구문과 반대로 전 규정 구문(full-specification syntax)을 사용하여야 한다. 만약 동일한 피시험 디바이스 형식에 대하여 하나 이상의 피시험 디바이스 정의가 제공되면, 첫번째 것은 상기 전 규정 구문을 사용하여야 한다.

3. (동일한 피시험 디바이스 형식에 대한) 이후의 각 피시험 디바이스 사이트 정의는, 전 규정 구문이나 슬롯 오프셋 구문의 어느 하나를 사용할 수 있으나 둘 다 사용할 수는 없다. 이렇게 함으로써, (예를 들어, 비작동 채널(inoperative channel)에 의하여) 개별 사이트가 표준 패턴으로부터 도출될 수 있게 된다.

4. 상기 슬롯 오프셋 구문으로부터 도출되는 한정은, (전 규정 구문을 사용하는) 그 피시험 디바이스 형식에 대하여 정의된 첫번째 사이트와 관련하여 정의된다.

5. 피시험 디바이스 사이트는 실제 물리적 순서에서 선언될 필요는 없다. 이렇게 함으로써, 첫번째 (물리적) 사이트가 상기 패턴으로부터 벗어나는 경우가 허용된다.

6. 상기 피시험 디바이스 사이트 식별자는 전체 소켓에 걸쳐(즉, 거기에서 정의된 모든 피시험 디바이스 형식에 걸쳐) 고유할 것이 요구된다.

7. 피시험 디바이스 사이트 정의마다, 적어도 하나의 리소스 정의가 요구된다.

8. 사이트 정의는, 테스트 구성이 단일 사이트/단일 피시험 디바이스 또는 단일 사이트/다중 피시험 디바이스의 어느 것인가를 판단하기 위하여 상기 모듈 구성과 연계되어 사용되어야 한다.

9. 모든 경우에, 상기 소켓 파일은, 핀 기술 파일 및 모듈 구성 파일과 모순이 없는 한 세트의 피시험 디바이스 채널 매핑을 규정하여야 한다.

10. 어떠한 경우에는, 상기 소켓 정의가 하나 또는 그 이상의 피시험 디바이스 채널이 (예를 들어, "0.0"과 같은 특별한 식별자로 상기 할당된 물리적 채널을 지정함으로써) 상기 테스터와 접속되지 않았다는 것을 규정할 수 있도록 허용하는 것이 바람직하다. 이 경우에, 이러한 피시험 디바이스 채널은 상기 테스트 프로그램의 문맥에서 사용되고 참조될 수도 있다. 그러한 채널상에서의 동작은 시스템 경고(오류가 아닌)로 귀결된다. 로드 시간에, 접속되지 않은 채널에 대한 패턴 데이터는 버려질 것이다.

소켓을 위한 구조

이하는 본 발명의 바람직한 실시예에 의한 모듈 구성을 위한 구조를 개시한다. 즉,

socket-file:

version-info socket-def

version-info:

Version *version-identifer;*

socket-def:

SocketDef{*device-specific-socket-def-list*}

device-specific-socket-def-list:

device-specific-socket-def

device-specific-socket-def-list device-specific-socket-def

device-specific-socket-def:

DUTType *DUT-type-name{pin-description-file dut-info-list}*

pin-description-file:

PinDesc *pin-description-file-name;*

dut-info-list:

dut-info

dut-info-list dut-info

dut-info:

DUT *dut-id{site-controller-input-port resource-info-list}*

site-controller-input-port:

SiteController *switch-matrix-input-port-number;*

resource-info-list:

resource-info

resource-info-list resource-info

resource-info:

Resource *resource-name{resource-item-unit-assignment-list}*

resource-item-unit-assignment-list:

resource-item-unit-assignment

resource-item-unit-assignment-list resource-item-unit-assignment

resource-item-unit-assignment:

resource-item-name slot-number.resource-unit;

resource-item-name[*resource-item-index*] *slot-number.resource-* unit-index;

resource-item-name[*resource-item-index-range*]

slot-number.[*resource-unit-index-range*];

resource-item-index-range:

resource-item-index:resource-item-index

resource-unit-index-range:

resource-unit-index:resource-unit-index

상기에 있어서, 정의되지 않은 비종단자(non-terminal)는 이하와 같이 규정된다. 즉,

1. 버전-식별자(version-identifier): 세트 [0-9a-zA-Z]로부터의 하나 또는 그 이상의 문자의 시퀀스. 버전 번호를 나타냄.

2. 피시험장치-형식-명칭(DUT-type-name): 세트 [a-zA-Z_0-9]로부터의 하나 또는 그 이상의 문자의 시퀀스로서, 첫번째 문자는 세트 [0-9]로부터의 문자가 아닌 것. CHIP3과 같이, 피시험 디바이스의 형식을 나타냄.

3. 핀-기술-파일-명칭(pin-description-file-name): 디렉토리 명칭을 포함하지 않으며 모든 확장자를 포함하는 파일의 간단한 명칭. 상기 파일 명칭은 상기 호스트 운영 체계가 인식할 수 있는 구문으로 되며, 따옴표로 묶인 경우 빈칸 및 기타 문자를 허용함.

4. 스위치-행렬-입력-포트-번호(switch-matrix-input-port-number): 상기 사이트 제어기에 접속된 입력 포트의 포트 번호를 나타내는 십진 표기법의 음수가 아닌 정수.

5. 피시험 디바이스-식별자(dut-id): 피시험 디바이스의 인스턴스를 식별하기 위한 십진 표기법의 음수가 아닌 정수.

6. 리소스-명칭(resource-name): 세트 [0-9a-zA-Z]로부터의 하나 또는 그 이상의 문자의 시퀀스로서, 첫번째 문자는 숫자가 아닌 것. 리소스 파일에 정의된 리소스의 명칭을 나타냄.

7. 리소스-아이템-명칭(resouce-item-name): 세트 [0-9a-zA-Z]로부터의 하나 또는 그 이상의 문자의 시퀀스로서, 첫번째 문자는 숫자가 아닌 것. 핀 또는 핀 그룹과 같은 리소스 유닛의 명칭을 나타냄.

8. 리소스-아이템-인덱스(resouce-item-index): 리소스 아이템 그룹의 특정 멤버를 나타내는 십진 표기법의 음수가 아닌 정수. 리소스-아이템-인덱스-범위(resouce-item-index-range)의 문맥인 경우, 그것은 리소스 아이템 그룹의 인접 시퀀스의 하한 또는 상한을 나타낸다.

9. 리소스-유닛-인덱스(resource-unit-index): 리소스 유닛(채널) 그룹의 특정 멤버를 나타내는 십진 표기법의 음수가 아닌 정수. 리소스-유닛-인덱스-범위(resouce-unit-index-range)의 문맥인 경우, 그것은 리소스 유닛 그룹의 인접 시퀀스의 하한 또는 상한을 나타낸다.

A6. 핀

(소켓에 의하여 제공되는 바와 같이) 물리적 채널 매핑에 대한 논리적 핀 명칭에 추가하여, 테스터 리소스를 특정하기 위하여 몇 가지 속성들이 사용될 수 있음을 주의하여야 한다. 예를 들어, 채널에 대한 특정 하드웨어 구성을 정의하기 위하여 옵션이 사용될 수도 있는데, 그것은 시험 특유적이거나, 벤더 특유적이거나 테스트 시스템 특유적일 수 있다. 이들은 핀 모드 선택 사양(Pin Mode Option)을 사용함으로써 기술될 수 있고, 핀 모드 선택 사양 파일을 통하여 활용가능하게 된다.

핀 모드 선택 사양의 정의는 테스터 채널에 대한 특별한 옵션 또는 모드들의 구성을 지원한다. 예를 들어, 이것은, 채널 멀티플렉싱(channel multiplexing)을 선택하고 구성하는 데에 사용될 수 있다. 상기 핀 모드 선택 사양은 상당한 채널 구성을 요구할 수 있으므로, 그것은 테스트 계획 초기화 흐름의 일부분으로서만 사용되는 것이 바람직하다. 상기 핀 선택 사양 구문은 벤더 정의 선택 사양을 지원한다. 이하에서 그 예를 개시한다. 즉,

PinModeOptions

{

clock IN double;

a0 OUT single;

...

}

테스트 환경 구성(Test Environment Configuration)

앞서 지적한 바와 같이, 상기 리소스 정의 파일(Resources.rsc), 상기 시스템 구성 파일(Sys.cfg) 및 상기 모듈 구성 파일(Modules.cfg)은 바람직하게 "잘 알려진" 위치에서 활용 가능하게 되어 있다. 이러한 "잘 알려진" 위치는 시스템 환경 변수인 Tester_ACTIVE_CONFIGS의 값에 의하여 규정되는 디렉토리이다. 예를 들어, Tester_ACTIVE_CONFIGS의 값이 'F:₩Tester_SYS₩configs'라는 디렉토리라면, 상기 시스템은 다음 파일들이 존재할 것으로 기대할 것이다. 즉,

F:₩Tester_SYS₩configs₩Resource.rsc

F:₩Tester_SYS₩configs₩Sys.cfg

F:₩Tester_SYS₩configs₩Modules.cfg

설치하는 동안, 상기 호스트 컴퓨터상에 존재하는 상기 ICM(Installation and Configuration Management System)은 바람직하게 상기 Tester_ACTIVE_CONFIGS의 값을 설정한다. ICM이 상기 파일의 새로운 버전을 창출할 때마다, 그것은 Tester_ACTIVE_CONFIGS에 의하여 지적되는 위치에 새로운 버전을 위치시킬 것이다. 상기한 세 가지 파일들에 추가하여, 시뮬레이션 구성 파일과 같은 다른 시스템 구성 파일도 Tester_ACTIVE_CONFIGS에 의하여 지적되는 위치에 놓인다.

B. 테스트 프로그램 개발을 위한 규칙

상기 테스터 시스템의 두 가지 주요 최종 사용자 지향 컴포넌트 중의 하나는, 테스트 환경(test environment)이다. 다른 컴포넌트는 상기 테스터가 최종 사용자(즉, 테스트 엔지니어 및 테스트 클래스 개발자)를 위하여 활용 가능하게 하는 프로그래밍 설비(programming facility)이다.

상기 프로그래밍 환경의 주요 컴포넌트는 상기 테스트 계획이다. 상기 테스트 계획은 테스트 클래스(시험(Test)라고 표기되는 테스트 인터페이스의 구현과는 다르다.)를 사용하며, 그것은 특정 형식의 시험을 위한 테스트 데이터와 코드의 분리를 실현한다.

상기 계획은 C++ 테스트 프로그램으로서 직접 작성될 수도 있으며, 또는 테스트 계획 기술 파일로 기술될 수도 있는데, 상기 테스트 계획 기술 파일은 C++ 코드와 같은 객체 지향적 코드를 생성하기 위하여 테스트 프로그램 생성기(번역기 402)에 의하여 처리된다. 그리고나서, 생성된 C++ 코드는 실행 가능한 테스트 프로그램으로 컴파일된다. 레벨, 타이밍 등과 같은 테스트 클래스 인스턴스를 채우기 위하여 필요한 데이터는 테스트 계획 기술 파일에서 사용자에 의하여 규정된다.

테스트 프로그램은 장치에 대하여 시험을 실행시키기 위한 상세를 규정하는 사용자가 작성한 파일의 세트를 수록한다. 본 발명의 일 실시예에 의하면, 사용자가 C++ 구성을 사용하여 이러한 파일들을 작성하도록 허가하는 규칙들의 세트를 포함한다.

본 발명의 일 실시예에 의한 요건들 중의 하나는 개방향 구조의 테스트 시스템의 모듈성을 따르는 것이다. 모듈식 개발에 의하여 사용자는 시험의 서로 다른 국면을 다루는 개별 컴포넌트들을 작성할 수 있으며, 이러한 컴포넌트들이 다양한 방식으로 섞이고 합쳐져서 완전한 테스트 프로그램을 산출하도록 한다. 본 발명의 바람직한 실시예에 의한 테스트 프로그램은 다음과 같은 파일들의 세트를 포함한다. 즉,

*.usrv 파일 - 사용자 변수들 및 상수들을 위한 파일;

*.spec 파일 - 규정 세트를 위한 파일;

*.lvl 파일 - 레벨을 위한 파일;

*.tim 파일 - 타이밍을 위한 파일;

*.tcg 파일 - 테스트 조건 그룹을 위한 파일;

*.bdefs 파일 - bin 정의를 위한 파일;

*.ph 프리헤더(pre-header)를 위한 파일 - 커스텀 기능 및 테스트 클래스를 위한 파일.

*.ctyp 파일 - 커스텀 형식을 위한 파일;

*.cvar 파일 - 커스텀 변수들을 위한 파일; 및

*.tpl 파일 - 테스트 계획들을 위한 파일.

상기 파일 확장자들은 파일들의 협약 이행 추천 분류(recommended convention facilitating categorization)이다. 하나의 테스트 프로그램은 바람직하게 하나의 테스트 계획 파일과 그것이 인입하는 파일을 포함한다. "인입(import)"이라 함은, 인입측(importer)(인입을 규정하는 파일)에 의하여 직접 참조되거나 인입측에 의하여 직접 참조된 어떤 다른 파일에 의하여 인입된 데이터를 가진 다른 파일을 가리킨다. 상기 테스트 계획 파일은, 그것 내부의 전역 객체들(global), 흐름(flow) 및 다른 그러한 객체들을 정의할 수 있으며, 또는 이 정보들을 다른 파일들로부터 인입할 수 있다. 이러한 규칙에 의하여, 상기한 어떤 컴포넌트라도 그들 자신의 개별 파일들 내에 존재하거나 또는 테스트 계획 파일로 직접 인라인(inline) 될 수 있다. 상기 테스트 계획은 개념상 C 언어의 main() 함수와 유사하다는 점에 주의하여야 한다.

테스트 프로그램 특징(Test Program Features)

사용자 변수들 및 상수들,

규정 세트(specification set),

레벨들(Levels),

타이밍들(Timings),

테스트 조건(Test Conditions)

Bin 정의(Bin Definition)

프리헤더들(Pre-Headers)

커스텀 형식들(Custom Types)

커스텀 변수들(Custom Variables)

테스트 계획(Test Plan)

테스트 프로그램 식별자는 바람직하게, 대문자 또는 소문자의 알파벳 문자로 시작하며, 이어서 임의의 개수의 알파벳, 숫자 또는 밑줄(_) 문자들을 가질 수 있다. 그것은 이하의 기술에서 제공되는 몇 개의 키워드를 가진다. 이러한 키워드는, 본 명세서에서는 **Version**과 같이 볼드체(bold font)를 사용하여 코드 내에서 시각적으로 식별되도록 하였다. 키워드는 유예되어 바람직하게 식별자로서 사용되지 않는다. {, }, (,), :, 및 이하에 기술되는 기타 몇몇 특별한 기호(symbol)들이 있다.

테스트 객체의 합성(Elaboration of Test Objects)

시험 기술 파일의 인입에 의하여, 인입하는 파일이 인입되는 파일에 의하여 활용 가능하게 된 객체들의 명칭들을 언급할 수 있게 된다. 이렇게 함으로써, 인입하는 파일은 인입되는 파일에 의하여 명명된 객체들을 참조할 수 있게 된다. 소켓 파일인 aaa.soc가 핀 기술 파일 xxx.pin을 인입한다고 생각해 보자. xxx.pin을 인입하는 다른 파일 bbb.soc도 존재할 수 있다. 그러나 이러한 인입의 어느 것도 xxx.pin에 기술된 객체가 존재하도록 하지는 않는다. 그들은 단순히 이미 존재하는 것으로 가정된 객체를 참조하는 것이다.

문제는, 그러한 객체들은 언제 존재하게 되는가이다. 이것은 상기 테스트 계획 파일이 근본적으로 다른 부분이다. C와의 유추에 의하면, 그것은 내부에 main() 루틴(routine)을 가진 파일이 될 것이다. 테스트 계획 파일에서 "**인입(Import)**" 문장은 이러한 객체들을 *합성*할 것이고, 그것은 이러한 객체들이 존재하게 한다는 것이다. 이하에 개시된 테스트 계획 mickey.tpl은 xxx.pin 및 aaa.soc의 객체들이 합성되도록 한다. 즉,

```
#File for Mickey's TestPlan

Version 3.4.5;

#

#이러한 인입 문장은 객체를 실제로 존재하게 할 것이다

#

Import xxx.pin; #pin 및 pin-group 객체를 합성

Import aaa.soc; #사이트 소켓 맵 객체(site socket map objects)를 합성

#필요에 따른 다른 인입

...

Flow Flow1

{

...

}
```

상기 테스트 계획에서의 xxx.pin의 인입에 의하여 xxx.pin에서 선언된 모든 핀 및 핀 그룹 객체가 합성되게 된다. 이것은 다음과 같이 기술된다. 즉, "상기 파일 xxx.pin이 합성되었다." 테스트 계획이 모든 합성되어야 할 파일들을 직접 인입하여야 할 필요는 없다. 만약 이하의 두 문장이 모두 진실이라면, 파일 x는 파일 y에 의하여 인입된다. 즉,

1. y는 x를 명명하는 인입 문장을 가지고 있다. 또는

2. x는 z에 의하여 인입되고, y는 z를 명명하는 인입 문장을 가지고 있다.

테스트 프로그램이 컴파일될 때, 상기 테스트 계획에 의하여 인입된 파일의 모든 객체를 합성할 것이다. 테스트 계획에 의하여 인입된 파일의 세트는 파일이 합성되는 순서를 산출하기 위하여 위상적으로 정렬(topologically sorted)된다. 테스트 계획에 의하여 인입된 파일들의 세트는 상기 테스트 계획의 인입 종결(import closure)이라 불린다. 만약 테스트 계획의 인입 종결이 위상적으로 정렬되지 못한다면, 반드시 인입 순환(import cycle)이 존재한다. 이러한 상황에서은 오류가 발생한 것이며, 따라서 컴파일러에 의하여 거절된다.

사용자 변수 및 상수

전역 변수 및 상수(global variables and constants)는 사용자 변수 및 상수를 사용하여 정의된다. 상수는 그 값이 컴파일 시에 한정되는 객체이며, 변경될 수 없다. 예를 들어, 최대의 정수 값은 상수이다. 반면에, 변수로 한정된 표현은 API를 통해 실행 시간에 변경될 수 있다.

정수(Integer),

무부호 정수(UnsignedInteger),

더블(Double),

문자열(String),

볼트 단위의 전위(Voltage in Volts)(V),

시간당 볼트 단위의 전위 슬루(VoltageSlew in Volts per Second)(VPS),

암페어 단위의 전류(Current in Amps)(A),

와트 단위의 전력(Power in Watts)(W),

초 단위의 시간(Time in Seconds)(S),

미터 단위의 길이(Length in Meters)(M),

헤르츠 단위의 주파수(Frequency in Hertz)(Hz),

오옴 단위의 저항(Resistance in Ohms)(Ohms), 및

패럿 단위의 캐패시턴스(Capacitance in Farads)(F).

정수, 무부호 정수, 더블 및 문자열의 형식들은 기본 형식(Basic Type)이라 불린다. 상기 기본 형식은 측정 단위를 갖지 않는다. 기본 형식이 아닌 기초적 형식(Elementary Type)에는, 관련된 측정 단위와 스케일(scale)을 가진 더블(Double)이 있다. 스케일링 기호(scaling symbol)는 공통된 엔지니어링 스케일링 기호들이다. 즉,

p(피코) : 10^{-12}, 예컨대 pF(pico-farad)

n(나노) : 10^{-9}, 예컨대 nS (nano-second)

u(마이크로) : 10^{-6}, 예컨대 uS (micro-second)

m(밀리) : 10^{-3}, 예컨대 mV (milli-amp)

k(킬로) : 10^{+3}, 예컨대 kOhm (kilo-ohm)

M(메가) : 10^{+6}, 예컨대 MHz (mega-hertz)

G (기가) : 10^{+9}, 예컨대 GHz (giga-hertz)

사용자 변수 및 함수를 가진 별개의 파일이 .usrv라는 확장자를 가질 것이다. 이하는 특정의 전역 상수를 가진 파일의 예이다. 특정의 변수를 가진 파일의 예를 이후에 제시한다.

```
#-------------------------------------------------------

#File limits.usrv

#-------------------------------------------------------

Version 1.0.0;

#

#이 사용자 변수 모음 선언은 전역적으로 활용 가능한

#변수들 및 상수들의 세트를 선언함.

#

UserVars

{

#특정 전역 상수인 정수(Integer)가 다양한 곳에서 사용됨.

Const Integer MaxInteger=2147483647;

Const Integer MinInteger=-2147483648;

#1.0+ Epsilon(가장 작은 값) != 1.0

Const Double Epsilon=2.220446049250313e-016;

#몇몇 중요한 상수는 더블과 관련됨.

Const Double MaxDouble=1.7976931348623158e+ 308;

Const Double MinDouble-MaxDouble;

Const Double ZeroPlus=2.2250738585072014e-308;

Const Double ZeroMinus=-ZeroPlus;

}
```

상기에서 선언된 사용자 변수의 세트는 '='의 왼편의 변수의 *정의*로서 여겨진다. 결과적으로, 변수 또는 상수의 정의의 단일한 발생이 바람직하며, 초기화되어야만 한다.

상기한 바와 같이, 상수는 일단 정의된 후에는 변경되어서는 안된다. 상수로 한정된 표현은 이전에 정의된 상수와 글자 그대로의 값을 수반할 수 있다. 반면에, 변수들은 API를 통해 변경될 수 있다. 변수로 한정된 표현은 이전에 정의된 변수, 상수 및 글자 그대로의 값을 수반할 수 있다.

각 변수는 실행 시간에 유지되는 표현 객체(expression object)로 한정된다. 이것은 실행 시간의 변수와 관련된 표현을 변경할 수 있게 하며, 그리고 나서 모든 변수들을 재평가할 수 있게 한다. 상기 표현 객체는 변수 또는 상수 정의의 오른편의 분리된 형태(parsed form)이다. 본 발명의 일 실시예에 의하면, 실행 시간에 상수를 변경할 수 있는 어떠한 설비도 제공되지 않는다. 그들의 값은 컴파일 시에 바람직하게 고정된다.

전역 객체를 갖는 어떠한 개수의 그러한 파일도 테스트 계획의 인입 종결에 존재할 수 있다. 상기 전역 파일이 수치 한계(numeric limit)의 세트인 한편, 엔지니어링 측정 단위를 사용하는 엔지니어링 전역 객체, 및 특정의 임의의 사용자 변수의 세트가 있다. 즉,

```
#-------------------------------------------------------------

# File myvars.usrv

#-------------------------------------------------------------

Version 0.1;

#

#이것은 특정 엔지니어링 전역 객체의 사용자 변수 모음을 선언한다.

#

UserVars MyVars

{

#엔지니어링 양(engineering quantity)

Const Voltage VInLow = 0.0; #0 Volts

Const Voltage VInHigh = 5.0; #5 Volts

Const Voltage VOutLow = 400.0 mV; #400 milliVolts

Const Voltage VOutHigh = 5.1; #5.1 Volts

Const Time DeltaT = 2.0E-9; #2 nanoseconds

Const Time ClkTick = 1.0ns; #1 nanosecond

Const Resistance R1O = 10.0 kOhms; #10 kilo Ohms

#특정의 변수들이 이하에서 선언된다.

Current ILow = 1.0 mA; #1 milliAmp

Current IHigh = 2.0 mA; #2 milliAmp
```

Power PLow = ILow * VInLow; #낮은 전력 값

Power PHigh = IHigh * VInHigh; #높은 전력 값

#

#모든 A 버스 핀에 대한 낮은 값의 배열.

#AO에 대한 vil은 ABusVil[0] 내에,

#Al에 대하여는 ABusVil[l] 내에, 기타 이와 같음

#

Voltage ABusVil[8]={1.0, 1.2, Others=1.5};

}

상기 컴파일러는 유닛들과 형식들이 일치하는지의 여부를 점검한다. 전위에 전류를 곱하면 전력이 산출되므로, 상기의 PLow 및 PHigh에 대한 수학식이 컴파일 할 것이다. 그러나, 이하와 같은 문장들은 전형적으로 컴파일하지 않는다. 즉,

#

#전력을 산출하기 위하여 전류와 전압을 더할 수 없으므로,

#컴파일하지 않음.

#

Power Pxxx = IHigh + VInHigh;

상기 컴파일러는 일정한 자동 형식 변환을 허용할 것이다:

Power Pxxx = 2; #전력을 2.0watts로 설정.

Integer Y = 3.6; #Y는 3으로 할당됨.

Power Pyyy = Y; #Pyyy는 3.0watts로 할당됨.

Double Z = Pyyy; #Pyyy는 단위 없는 더블 형식으로 변환됨.

무부호 정수(UnsignedInteger) 및 정수(Integer)도 또한 더블 형식으로의 명시적 변환이 허용된다. 즉,

Power Pxxx = 3.5;

#명시적 형식 변환이 허용되지만, 필요하지 않음.

#X는 3.5가 됨.

Double X = Double(Pxxx); #X는 3.5가 됨.

Integer Y = Integer(Pxxx); #Y는 3이 됨.

중간의 기본 형식으로의 변환에 의하여, 관련되지 않은 형식들 사이에서의 변환도 또한 가능하다. 즉,

Power Pxxx = 3.5;

#명시적 형식 변환이 필요함.

Length L = **Double**(Pxxx); #L은 3.5meters가 됨.

Voltage V = **Integer**(Pxxx); #V는 3.0 Volts가 됨.

상기 테스트 계획 객체는, 명칭 및 그들의 관련된 표현들, 값들 및 형식들을 수록한 모음인 사용자 변수 클래스(*UserVars* class)를 제공한다. 사용자 변수들은 디폴트 사용자 변수 모음(*Default User Variables Collection*) 또는 명명된 사용자 변수 모음(*Named User Variables Collection*)으로 들어갈 수 있다. 특별히 규정된 명칭을 구비하지 않은 상기 예에서의 상기 사용자 변수 선언은 상기 디폴트 모음으로 들어갈 수 있다. 그러나, 이하와 같이 모음을 명시적으로 명명할 수도 있다. 즉,

#사용자 변수 모음인 MyVars에서 X 및 Y를 선언함.

UserVars MyVars

{

Integer X = 2.0;

#

#디폴트 사용자 변수 모음으로부터의

#상기 X 및 전역적으로 활용 가능한

#MaxInteger를 가리킴.

#

Integer Y = MaxInteger-X;

}

#사용자 변수 모음인 YourVars에서 X, Y1 및 Y2를 선언함.

UserVars YourVars

{

Integer X = 3.0;

#MyVars로부터의 X를 가리킴.

Integer Y1 = MaxInteger-MyVars.X;

#상기에서 선언된 X를 가리킴.

Integer Y2 = MaxInteger-X;

}

#MyVars 모음에 더 많은 변수들이 추가됨.

UserVars MyVars

{

#

#이전의 MyVars 선언으로부터의 X 및 Y를 가리킴.

#

Integer Z = X + Y;

}

사용자 변수 모음 내에서의 명칭 해석은 이하와 같이 진행된다. 즉,

하나의 명칭이 인가(qualified)되면(즉, 하나의 명칭이 점으로 구분된 두 개의 세그먼트(segment)를 포함하면), 그 변수는 그 점에 선행하는 세그먼트에 의하여 명명된 사용자 변수 모음으로부터 유래한다. 그리하여, 상기에서의 MyVars.X는 MyVars 모음에서의 X를 가리킨다. "_UserVars"라는 명칭은 디폴트 사용자 변수 모음을 명시적으로 지시하는 데에 사용될 수 있다.

만약 명칭이 인가되지 않고 또한 현재의 모음에 동일한 명칭의 상수 또는 변수가 존재한다면, 상기 명칭은 그 상수 또는 변수로 해석된다.

그렇지 않으면, 상기 명칭은 상기 디폴트 사용자 변수 모음의 상수 또는 변수로서 해석된다.

사용자 변수 모음에서의 정의 블록(definition block)의 평가는, 첫번째 정의로부터 마지막 정의까지 순차적으로 발생하는 것으로 생각될 수 있다. 이것은 각 변수들이 사용되기 전에 정의되어야 할 것을 요구할 수도 있다.

게다가, 사용자 변수 모음에 대한 복수의 정의 블록이 있을 수 있는데, 이들 각각은 복수의 변수들을 정의한다. 이러한 정의 블록 모두는 테스트 계획에서 정의된 순서로 평가되는 것으로 생각될 수 있으며, 각 블록의 변수들도 또한 선언된 순서로 점검된다.

마지막으로, 복수의 사용자 변수 모음이 있을 수 있고, 이들의 각각은 복수의 정의 블록에 변수들을 정의한다. 다시, 모든 변수들은 선언된 순서로 초기화되는 것으로 생각될 수 있다. 그러므로, 상기한 예에서, 평가 순서는, MyVars.X, MyVars.Y, YourVars.X, YourVars.Y1, YourVars.Y2, MyVarz.Z가 될 것이다.

사용자 변수 모음이 다른 모음으로부터의 변수를 사용하는 경우, 그것은 바람직하게 상기 변수의 원래 값(raw value)을 그대로 사용한다. 모음들 사이에 어떠한 의존성 정보(dependency information)도 유지되지 않는다. 그러므로, 의존성에 기반한 재평가는 하나의 모음에만 제한될 수 있다.

각 사용자 변수 모음은 C++ 사용자 변수 클래스의 인스턴스를 가리킨다. 상기 C++ 사용자 변수 클래스의 디폴트 객체는 "_UserVars"로 명명된다. 명명되지 않은 하나의 사용자 변수 선언의 변수들은 디폴트 사용자 변수 모음으로부터 유래되며, 이 디폴트 객체에 추가된다. 명명된 사용자 변수 모음의 변수들은, 그 명칭을 가진 C++ 사용자 변수 클래스의 객체에 추가된다. 상기 예에서는, 상기 "MyVars" C++ 객체는 변수들 X, Y 및 Z를 가지며 종료될 것이다.

사용자 변수를 위한 C++

사용자 변수는, 명칭 문자열, 상수/변수 진리값(*const/var* boolean), 계수된 값으로서의 형식 및 표현 트리(expression tree)로서의 표현을 가진 n개의 요소로 된 집합의 모음으로서 구현된다. 명칭의 표현은 호출에 의하여 설정될 수 있다. 즉,

```
enum ElemenaryType{UnsignedIntegerT, Integer,

DoubleT, VoltageT,...};

Status setExpression(const String&name,

const bool isConst,

const elementaryType,

const Expression& expression);
```

상기 표현 형식은 할당의 오른편에 대응되는 텍스트의 분리된 형태인 형식이다. 전역적으로 활용 가능한 사용자 변수의 인스턴스가 존재할 수 있다. 예를 들어, limits.usrv(페이지를 참조하라)의 사용자 변수들의 세트는 이하에 개시된 호출들의 세트에 의하여 구현될 수 있다.

```
_UserVars.setExpression("MaxInteger", true, Integer,

Expression(2147483647));

_UserVars.setExpression("MinInteger", true, Integer,

Expression(-2147483648));

_UserVars.setExpression("Epsilon", true, DoubleT,

Expression(2.2204460492503131e-016));

_UserVars.setExpression("MaxDouble", true, DoubleT,

Expression(1.7976931348623158e+ 308));

_UserVars.setExpression("MinDouble", true, DoubleT,

Expression("-MaxDouble"));

_UserVars.setExpression("ZeroPlus", true, DoubleT,

Expression(2.2250738585072014e-308));

_UserVars.setExpression("ZeroMinus", true, DoubleT,

Expression("-ZeroPlus"));
```

이하는 myvars.usrv에 선언된 변수들을 위하여 실행될 수 있는 C++ 문장들이다. 즉,

```
myVars.setExpression("VInLow", true, VoltageT,

Expression(0.0));
```

```
myVars.setExpression("VInHigh", true, VoltageT,
Expression(5.0));
myVars.setExpression("DeltaT", true, TimeT,
Expression(2.0E-9));
myVars.setExpression("ClkTick", true, TimeT,
Expression(1.0E-9));
myVars.setExpression("R10", true, ResistanceT,
Expression(10.0E+3));
myVars.setExpression("ILow", false, CurrentT,
Expression(1.0E-3));
myVars.setExpression("IHigh", false, CurrentT,
Expression(2.0E-3));
myVars.setExpression("PLow", false, PowerT,
Expression("ILow*VInLow"));
myVars.setExpression("PHigh", false, PowerT,
Expression("IHigh*VInHigh"));
myVars.setExpression("ABusVil[0]", false, VoltageT,
Expression(1.0));
myVars.setExpression("ABusVil[1]", false, VoltageT,
Expression(1.2));
myVars.setExpression("ABusVil[2]", false, VoltageT,
Expression(1.5));
myVars.setExpression("ABusVil[3]", false, VoltageT,
Expression(1.5));
myVars.setExpression("ABusVil[4]", false, VoltageT,
Expression(1.5));
myVars.setExpression("ABusVil[5]", false, VoltageT,
```

```
Expression(1.5));

myVars.setExpression("ABusVil[6]", false, VoltageT,

Expression(1.5));

myVars.setExpression("ABusVil[7]", false, VoltageT,

Expression (1.5));
```

상기 코드에 있어서, 상기 표현 클래스는 바람직하게 표현의 분리된 형태를 나타내는 구축자(constructor)를 가진다. 표현은 복수의 구축자를 가지며, 이들 중 하나는 문자열 리터럴(string literal)을 가지며 그것을 분리하며, 다른 것은 문자열 리터럴을 있는 그대로 사용하기 위하여 문자열 리터럴을 취한다. 이들은 가독성을 위하여 상기에서는 특정되지 않은 추가적인 파라미터들에 의하여 구별된다.

상기 디폴트 사용자 변수 모음의 사용자 변수는 사용자 변수 클래스의 _UserVar 객체에 의하여 관리된다. 명명된 사용자 변수 모음 Xxx의 사용자 변수는 Xxx로 명명된 UserVar 객체에 의하여 관리될 것이다.

사용자 변수를 위한 실행 시간 API

이러한 명칭들 및 표현들을 수록한 C++ 사용자 변수 클래스는 실행 시간에 이들 값들을 평가하고 수정하기 위하여 응용 프로그램 프로그래밍 인터페이스(application programming interface; API)를 이출(export)한다. 사용자 변수와 관련된 표현의 수정은 언제 사용자 변수가 재평가되는가와 평가의 영향은 무엇인가하는 문제들도 야기한다.

변경의 결과로서 사용자 변수의 재평가가 개시되어야 하는가하는 첫번째 문제를 생각해 보자. 만약 표현에 변화를 가한 후 즉시 재평가가 개시된다면, 사용자는 재평가의 개시 전에 일련의 관련된 변화를 만들지 못하게 될 것이다. 결과적으로 재평가는 사용자의 명시적 호출에 의하여 개시된다.

다음에, 재평가의 영향에 관하여 생각해 보자. 본 발명의 바람직한 실시예에 의하여 활용 가능한 세 가지 종류의 재평가가 존재한다. 즉,

'사용자 변수 모음 재평가(*UserVars Collection Re-evaluation*)'는, 하나의 사용자 변수 모음에 제한되는 재평가이다. 이 동작의 의미론은 이 모음의 모든 변수들을 다시 한 번 재평가한다는 것이다.

'사용자 변수 목표 재평가(*UserVars Targeted Re-evaluation*)는, 하나의 명칭에 한정된 표현의 변화에 제한되는 재평가이다. 이것은, 사용자로 하여금 하나의 명칭의 표현을 변경하고 단지 이 특정한 변경만을 고려하여 모음의 재평가가 발생되도록 하게 한다.

'사용자 변수 전역 재평가(*UserVars Global Re-evaluation*)는, 모든 사용자 변수 모음의 재평가이다. 이것은 기본적으로 선언된 순서로 모든 사용자 변수 모음의 재평가를 개시하며, 매우 소모적이다.

상기의 모든 재평가는, 상기 사용자 변수의 재평가 후에, 레벨, 타이밍 등과 같은 종속적 객체(dependent object)를 재평가한다. 종속적 객체들은 재평가가 필요하다는 것을 나타내는 오염 비트(dirty bit)를 가질 것이다. 사용자 변수 모음이 프로그램적으로 변경될 때마다, 그것은 또한 모든 종속적 변수들에 오염 비트를 설정할 것이다. 이것은 종속적 객체들의 재평가를 개시한다.

요컨대, 명명된 사용자 변수 모음은 재평가의 영향의 문제를 억제하도록 돕는다. 재평가는 정규적으로 하나의 모음에 제한된다. 사용자 변수를 사용하는 간단한 방법은 단지 디폴트 사용자 변수 모음을 사용하는 것이 될 것이다. 변경에 의한 파급 효과는 모든 사용자 변수에 발생할 수 있다. 이러한 파급 효과는 복수의 명명된 사용자 변수 모음을 구비함으로써 제한될 수 있다.

다수의 모음은 서로의 변수들을 가리킬 수 있으나, 변수에 한정된 값들은 사용시로 한정된다. 사용자 변수 모음 사이에는 종속성이 유지되지 않는다.

각각의 기초적 형식 Xxx(무부호 정수, 전류, 전위, 등)에 대하여, 값을 획득하는 메소드로서,

Status getXxx Value(const String& name, Xxx& value) const;

가 있다.

값을 직접 설정하는 메소드는 없으며, 그것은 그 표현을 설정하는 호출과 이어지는 reevaluateCollection()의 호출을 통하여 수행된다는 점을 주의하여야 한다.

표현을 획득하고 설정하는 메소드들이 있다. 상기 setExpression() 호출은 지금까지 정의되지 않은 새로운 변수를 정의하는 데에도 역시 사용될 수 있다. 즉,

enum elementaryType

{

UnsignedIntegerT, IntegerT, DoubleT, VoltageT,...

};

Status getExpression(const String& name, Expression& expression)const;

Status setExpression(const String& name,

const bool isConst,

const elementaryType,

const Expression& expression);

상기 setExpression() 호출은, 만약 상기 표현이 순환 종속성으로 귀결된다면 실패할 수 있다. 예를 들어, 이하의 두 개의 호출이 만들어진다면, 두 번째 호출은 순환 종속성 실패로 인하여 실패할 것이다. 즉,

setExpression("X", true, Integer, Expression("Y+1"));

setExpression("Y", true, Integer, Expression ("X+1"));

이것은 명칭에 한정된 값이 할당이 아니라 수학식이기 때문이다. 변수의 값이 변경되면, 메소드는 모든 직접적으로 또한 간접적으로 종속적인 명칭들을 재평가하도록 제공된다. 상기한 쌍과 같은 수학식은 허용되지 않는 순환 종속성으로 귀결된다.

이 API는 전형적으로 요구받지 않은 재평가(unsolicited re-evaluation)을 지원하지 않는다는 점을 주의하여야 한다. setExpression()의 호출은 자동적으로 상기 변수와 그에 종속된 모든 다른 변수들이 재평가되도록 한다. 모든 변수들에 한정된 값들은 (이하의) reevaluateCollection()의 호출이 발생할 때까지 변경되지 않은 채로 머무른다.

특정 명칭이 상수인지의 여부를 판단하는 메소드:

Status getIsConst(const String& name, bool& isConst);

형식을 획득하는 메소드:

enum ElementaryType

{

UnsingnedIntegerT, IntegerT, DoubleT, VoltageT,...

};

Status getType(const String& name,

ElementaryType& elementaryType) const;

사용자 변수 모음 재평가 메소드:

Status reevaluateCollection();

상기 클래스는 모든 변수들에 관련된 수학식과 그들의 종속성을 유지할 것이다. 이 메소드가 호출되면, 모든 변수들은 재평가될 것이다.

사용자 변수 목표 재평가 메소드:

Status reevaluateTargeted(const String& var);

상기 클래스는 모든 변수들에 관련된 수학식과 그들의 종속성을 유지할 것이다. 이 메소드가 호출되면, 명명된 변수 및 그의 모든 종속자(dependents)들이 재평가될 것이다.

사용자 변수 전역 재평가 메소드:

static Status reevaluateAllCollections();

상기 클래스는 모든 변수들에 관련된 수학식과 그들의 종속성을 유지할 것이다. 이 메소드가 호출되면, 특정되지 않은 순서로 모든 사용자 변수 모음에 대하여 reevaluationCollection()이 호출된다.

특정한 명칭이 정의되었는지의 여부를 판단하는 메소드:

Status getIsDefined(const String& name, bool& isDefined) const;

모든 사용자 변수들이 현재 정의되었는지의 여부를 판단하는 메소드:

Status getNames(StringList& names) const;

현재 정의된 변수를 삭제하는 메소드:

Status deleteName(const String&Name);

만약 명칭이 다른 변수를 필요로 하는 표현에 사용된다면 이 동작은 실패한다.

주어진 변수 또는 상수에 종속적인 변수들 및 상수들의 리스트를 획득하는 메소드:

Status getDependents(const String& name, StringList& dependents);

규정 세트(Specification Set)

규정 세트는, 선택자(Selector)에 기초하여 값들을 취할 수 있는 변수들의 모음을 지원하기 위하여 사용된다. 예를 들어, 선택자인 미니(Minnie), 미키(Mickey), 구피(Goofy) 및 데이지(Daisy)를 사용하는 이하의 규정 세트를 생각해 보자.

```
#-------------------------------------------------------

#File Aaa.spec

#-------------------------------------------------------

Version 1.0;

Import Limits.usrv;

SpecificationSet Aaa(Minnie, Mickey, Goofy, Daisy)

{

Double xxx = 1.0, 2.0, 3.0, 4.0;

Integer yyy = 10, 20, 30, 40;

Integer zzz = MaxInteger-xxx,

MaxInteger-xxx-1,

MaxInteger-xxx-2,

MaxInteger-xxx;

#이하의 선언은 하나의 값에 관련되며,

#그것은 선택자에 무관하게 선택될 것이다.

#그것은 다음과 등가이다. 즉,

# Integer www = yyy+ zzz,yyy+ zzz,yyy+ zzz,yyy+ zzz

Integer www = yyy+ zzz;

}
```

구피(Goofy)를 선택자로 갖는 상기 규정 세트는 다음의 관련을 만들 것이다. 즉,

```
xxx = 3.0;

yyy = 30;

zzz = MaxInteger-xxx-2;

www = yyy+ zzz;
```

규정 세트에 선택자를 설정하는 동작은 이후에, 테스트(Tests)가 기술될 때에, 기술될 것이다.

구문적으로(syntactically), 규정 세트는 변수의 정의의 리스트(상기 예에서의 xxx, yyy, zzz 및 www)과 함께 선택자들의 리스트(상기 예에서의 미니, 미키, 구피 및 데이지)이다. 변수들의 정의는 선택자들의 리스트만큼이나 긴 표현의 리스트를 수반하거나 또는 하나의 표현을 포함한다.

개념적으로 규정 세트는 표현의 행렬로 생각될 수 있는데, 그 행은 선택자들이며, 그 열은 변수들이고, 그 구성요소는 표현들이다. 특정 선택자(행)는 각 변수(열)을 특정의 표현(구성요소)으로 한정시킨다. 만약 리스트가 하나의 표현만을 갖는다면, 그것은 선택자만큼 복제되는 표현을 가진 열을 나타낸다.

규정 세트는 두 가지 별개의 상황에서 나타날 수 있다. 그들은 .spec 파일에서 별도로 선언될 수 있는데, 이 경우에 그들은 상기한 바와 같이 나타난다. 이들은 '명명된 규정 세트'이다. 그렇지 않으면, 지역 규정 세트가 테스트 조건 그룹 내에서 선언될 수 있다. 이러한 선언에서는, 상기 규정 세트에는 명칭이 제공되지 않을 것이다. 그것은, 단지 그를 수용한 테스트 조건 그룹에만 중요성을 갖는 지역적인 규정 세트가 될 것이다.

명명된 규정 세트는 명명된 사용자 변수 모음에 따라 모델화 될 수 있다. 상기 규정 세트는, Aaa로 명명된 사용자 변수 모음으로서 모델화 될 수 있으며, 그것은 xxx[Minnie], xxx[Mickey], xxx[Goofy], xxx[Daisy], yyy[Minnie], 등에 대한 표현을 가질 것이다. 특정 선택자(Mickey와 같은)가 시험 상황 중에 선택되면, xxx, yyy 및 zzz의 값들이 변수 명칭 및 규정 세트 명칭으로부터 획득된다.

테스트 조건 그룹은 많아야 하나의 규정 세트를 가질 수 있는데, 그것은 지역 규정 세트이거나 명명된 규정 세트로의 참조의 어느 하나이다. 지역 규정 세트는 테스트 조건 그룹의 상황에서만 나타나며, 명시적으로 규정된 명칭을 갖지 않는다. 그러한 규정 세트는 그를 수용한 테스트 조건 그룹의 명칭에 의하여 정의되는 묵시적인 명칭을 갖는다. 복수의 규정 세트와 복수의 사용자 변수 모음이 가시적인 지점에서 테스트 조건 그룹의 명칭을 해석하기 위하여, 이하의 규칙들이 적용된다. 즉,

1. 명칭이 인가되면, 그것은 명명된 사용자 변수 집합 내에서 해석되어야 한다.

2. 명칭이 인가되지 않으면, 상기 명칭은 그것이 테스트 조건 그룹에서 선언되었다면 지역 규정 세트에서, 또는 그것이 테스트 조건 그룹에서 참조되었다면 명명된 규정 세트 내에서 해석된다.

3. 상기 규칙에 의하여 명칭이 해석되지 않는다면, 디폴트 사용자 변수 모음에서 해석된다.

이들 규칙들을 예시하기 위하여, 테스트 조건 그룹(후에 기술됨)을 사용한 이하의 예를 개시한다.

```
Version 1.2.3;

Import limits.usrv; #상기한 제한 사용자 변수 파일을 입수.

Import aaa.spec; #상기 규정 세트 AAA를 입수.

TestConditionGroup TCG1

{

SpecificationSet(Min, Max, Typ)

{

vcc=4.9, 5.1, 5.0;

}

#규칙 1: 명명된 사용자 변수 모음에서 해석.

#MyVars.VInLow로의 참조는 MyVars의 VInLow를 가리킴.

#규칙 2: 지역 규정 세트에서 해석.
```

#여기에서의 "vcc"로의 참조는 상기 지역 규정 세트의

#상황에서 해석할 것.

#규칙 3: 디폴트 사용자 집합에서 해석.

#여기에서의 "MaxInteger"로의 참조는 limits.usrv로 해석됨.

#오류: Aaa.xxx의 해석

#xxx로의 참조는, 그것이 지역 규정 세트에도 limits.usrv에도

#속하지 않으므로, 해석하지 않음.

#오류: Aaa.xxx의 해석

#Aaa로 명명된 명명된 사용자 변수 모음을 찾음.

#명명된 규정 세트는 인가하지 않음.

}

TestConditionGroup TCG2

{

SpecificationSet Aaa; #인입된 규정 세트를 참조함.

#규칙: 명명된 사용자 변수 모음에서 해석.

#MyVars.VInLow로의 참조는 MyVars로부터의 VInLow를 가리킴.

#규칙 2: 명명된 규정 세트에서 해석.

#여기에서의 "xxx"로의 참조는 상기 지역 규정 세트의

#상황에서 해석할 것.

#규칙 3: 디폴트 사용자 집합에서 해석.

#여기에서의 "MaxInteger"로의 참조는 limits.usrv로 해석됨.

#오류: vcc의 해석

#vcc로의 참조는, 그것이 지역 규정 세트에도 limits.usrv에도

#속하지 않으므로, 해석하지 않음.

#오류: Aaa.xxx의 해석

#Aaa로 명명된 명명된 사용자 변수 모음을 찾음.

#명명된 규정 세트는 인가하지 않음.

}

규정 세트(상기 규칙)의 명칭의 해석은, 상기 집합의 선택자가 명칭 해석이 요구될 때에 가동되어(enabled) 있을 것을 요구한다. 이것은, 상기 테스트 조건 그룹이 선택자를 규정함으로써 시험에서 참조된다는 사실에 의하여 강제될 것이다.

규정 세트를 위한 C++

상기 규칙들을 사용함으로써, 규정 세트는 C++ 규정 세트 클래스에 의하여 구현될 수 있다. 상기 규정 세트 클래스는, 선택자를 위한 여분의 문자열 파라미터를 제외하고는 상기 사용자 변수 클래스와 본질적으로 같은 API를 구비한다. 결론적으로 이 API는 상세하게 기술하지 않는다.

모든 명명된 규정 세트는 바람직하게 그 명칭의 C++ 객체와 관련된다. 테스트 조건 그룹의 상황에서의 지역 규정 세트는 당해 테스트 조건 그룹에 고유한 명칭을 가질 것이다. 내부에 정의된 테스트 조건 그룹의 상황 밖에서 지역 규정 세트의 변수를 가리키는 것은 규정에 어긋난다.

레벨들(Levels)

상기 레벨들은 핀 및 핀 그룹의 파라미터를 규정하기 위하여 사용된다. 그것은 다음의 형태를 갖는 선언의 모음이다. 즉,

<pin-or-pin-group-name>

{

<pin-param-l> = xxx;

<pin-param-2> = yyy;

...

}

그러한 선언은 명명된 핀 또는 핀 그룹의 다양한 파라미터들의 설정을 규정한다. 예를 들어, 이하의 예에서 개시된 바와 같이, 그러한 문장은 *InputPins* 그룹에서 모든 핀들에 대한 VIL 값을 설정하는 데에 사용될 수 있다. 즉,

#---

File CHIPlevels.lvl

#---

Version 1.0;

Import CHIP3resources.rsc;

Import CHIP3pins.pin;

Levels CHIP3Levels

{

#

#상기 규정 세트로부터의 전역 객체들과

#값들을 이용하여 다양한 핀 및 핀 그룹에

#대한 핀 파라미터를 규정함.

#

#규정의 순서가 중요함.

#핀 파라미터들은 이 레벨 섹션에서

#첫번째로부터 마지막의 순서로,

#또한 각 핀 또는 핀 그룹 서브 섹션에서

#첫번째로부터 마지막의 순서로 설정됨.

#

#인입된 핀 기술 파일 CHIP3pins.pin으로부터,

#상기 InPins 그룹은 "dpin" 리소스내에 존재함.

#인입된 리소스 정의 파일 CHIP3resources.rsc로부터,

#상기 "dps" 리소스는 VIL 및 VIH로 명명된 파라미터를 가짐.

Inpins {VIL = v_il; VIH = v_ih + 1.0;}

#이하의 문장은, InPins 레벨들을 설정하기 위한

#호출 이후에 10uS의 지연을 요구한다.

#실제의 지연은 시스템이 정의한 작은 범위인 약 10.0E-6가 된다.

10.0E-6 - delta <= 실제 <= 10.0E-6 + delta

Delay 10.0E-6;

#

#OutPins에 대하여, 파라미터

#VOL 및 VOH에 대한 레벨이 규정된다.

#

OutPins {VOL = v_ol/2.0; VOH = v_oh;}

#클록 핀은 특별한 값을 가질 것이다.

Clock {VOL = 0.0; VOH = v_ih/2.0;}

#클록 레벨을 설정하는 호출 이후에 10uS의 지연.

#이것은 최소한의 지연으로서,

#사실은 조금 더 되지만,

#적어도 10.0uS는 된다는 것을 보장한다. 즉,

10.0E-6 <= 실제 <= 10.0E-6 + delta

MinDelay 10.0uS;

#

#전력(PowerPins) 핀 그룹은 "dps" 리소스내에 있다.

#이 핀 그룹의 핀들은 특별한 파라미터를 갖는다. 즉,

PRE-WAIT은, 전위가 그의 최종값에 도달한 후

패턴 생성을 시작할 때까지 대기하여야 할

시간을 규정한다. 실제의 대기 시간은

대략 PRE_WAIT(see)라는 시스템이 정의한 작은 범위이다.

POST-WAIT은, 패턴 생성이 완료된 후 전원을

차단할 때까지 대기하여야 할 시간을 규정한다.

실제의 대기 시간은 대략 POST_WAIT(see)라는

시스템이 정의한 작은 범위이다.

#

PowerPins

{

PRE_WAIT = 10.0ms;

POST_WAIT= 10.0ms;

#VCC는 램프(ramp) 방식으로 초당 ±.01 Volts의

#전위 슬루율(Voltage Slew Rate)로 그의

#현재값으로부터 그의 최종값인 2.0V에 도달한다.

VCC = Slew(0.01, 2.0V);

}

}

Levels CHIP4Levels

{

#...

}

상기에 개시된 바와 같이, 각 레벨 블록은 복수의 레벨 항목(levels items)으로 이루어지는데, 그 각각은 핀 또는 핀 그룹의 파라미터를 규정한다. 각 레벨 항목은 복수의 리소스 파라미터를 규정할 수 있다. 이 레벨 값들의 설정을 위한 실행 시간 구문은 다음과 같다. 즉,

레벨 블록의 레벨 항목은 선언된 순서대로 처리된다. 하나 이상의 레벨 항목에서 발생하는 어떠한 핀도 여러번 처리될 것이다. 하나의 파라미터에 대한 값의 복수의 규정은 유지되어야 하며 규정 순서에 따라 적용된다.

레벨 항목에서의 리소스 파라미터는 그들이 규정된 순서에 따라 처리된다.

지연(**Delay**) 문장은, 다음 그룹의 레벨을 설정하기에 앞서, 레벨 설정의 처리를 대체로 지시된 기간 동안 중지하게 한다. 실제의 대기 시간은 규정된 지연 전후의 시스템이 정의한 작은 범위에 속할 수 있다. 그러므로, 만약 상기 지연이 t초였다면, 실제의 지연은 다음을 만족할 것이다. 즉,

$t-\Delta t <=$ 실제 대기 $<= t+\Delta t$

상기 지연 문장은 상기 레벨 규정을 복수의 서브시퀀스(subsequence)들로 분할하는데, 그 각각은 처리를 위한 별도의 테스트 조건 메모리 설정을 요구할 것이다.

상기 최소 지연(**MinDelay**) 문장은, 다음 그룹의 레벨을 설정하기에 앞서, 레벨 설정의 처리를 대체로 지시된 기간 동안 중지하게 한다. 실제의 대기 시간은 규정된 최소 지연의 최소 값을 가진 시스템이 정의한 작은 범위에 속할 수 있다. 그러므로, 만약 상기 최소 지연이 t초였다면, 실제의 지연은 다음을 만족할 것이다. 즉,

$t <=$ 실제 대기 $<= t+\Delta t$

상기 지연 문장은 상기 레벨 규정을 복수의 서브시퀀스(subsequence)들로 분할하는데, 그 각각은 처리를 위한 별도의 테스트 조건 메모리 설정을 요구할 것이다.

각 핀 또는 핀 그룹의 명칭은 핀 기술 파일(접미사 .pin)에서 정확히 하나의 리소스에 규정되며, 그리하여 리소스 파일(접미사 .rsc)에 규정된 실용적인(viable) 리소스 파라미터의 소정 세트를 갖는다. 명명된 모든 파라미터들은 이 실용적인 리소스 파라미터의 세트로부터 유래하여야 하며, 그들의 값을 설정하기 위하여 사용된 표현과 동일한 기초적 형식을 가져야 한다. 리소스 파라미터들의 명칭들 및 형식들에 관한 정보는 상기 리소스 파일로부터 유래한다.

상기 리소스 파일 Resource.rsc는, **dpin** 및 **dps**와 같은 표준 리소스의 파라미터에 대한 명칭 및 형식을 테스터에 제공하기 위하여 묵시적으로 인입된다.

리소스 파라미터는, 사용자 변수, 그리고 명명된 규정 세트 또는 현재 가시적인 지역 규정 세트로부터의 값들을 사용할 수 있는 할당된 표현이다.

Dps 핀 리소스는 PRE_WAIT 및 POST_WAIT라는 특별한 파라미터를 갖는다. 상기 PRE_WAIT 파라미터는, 전력 핀이 그의 목적 전위에 도달한 시간으로부터 패턴 생성이 시작될 수 있는 시간까지 소요되어야 할 필요가 있는 시간을 규정한다. 상기 POST_WAIT 파라미터는, 패턴 생성이 중단된 시간으로부터 전력 핀이 전원을 차단하는 시간까지 소요되어야 할 필요가 있는 시간을 규정한다.

Dps 핀은 또한, 전위 파라미터가 어떻게 그의 최종 값에 도달하는지에 관하여도 규정한다. 그들은, 다른 모든 핀 파라미터와 같이, 수학식에 의하여 그것을 간단히 규정할 수 있다. 그 경우, 그 값은 하드웨어가 그것을 허용하는 만큼 도달될 수 있

다. 그들은 또한 슬루 문장(Slew statement)을 사용하여 그것을 규정할 수도 있다. 슬루 문장은, 전략 공급 전위가 램프(ramp) 방식으로 규정된 절대적 전위 슬루율(abolute Voltage Slew Rate)에 따라 그의 초기값으로부터 그의 최종값에 도달할 것을 규정한다.

레벨을 위한 C++

상기한 규칙을 바탕으로, 이하의 동작을 지원하도록 C++ 레벨 객체를 작성할 수 있다. 즉,

하나의 동작이 존재한다:

```
Status setParameter(const String& pinOrPinGroupName,

const String& parameterName,

ElementaryType elementaryType,

const Expression& Expression);
```

이 동작은 표현을 핀 또는 핀 그룹의 파라미터로 한정시킨다. 예를 들어, dpin.InPins VIH의 값은 다음과 같이 설정된다. 즉,

```
setParameter("InPins", "VIH", VoltageT,

Expression("v_ih +  1.0");
```

이 동작은 레벨 객체의 모든 선언에 대하여 여러 번 호출될 것이다.

하나의 동작이 존재한다:

```
Status assignLevels(const String& selector);
```

이것은, 상기한 바와 같이, 규정된 순서에 따라 파라미터의 모든 레벨을 할당하기 위하여 모든 미리 정의된 모듈 레벨 인터페이스를 하나 하나 조사하여 발행할 것이다. 선택자 파라미터는 상기에 규정된 규칙에 따라 표현에서 명칭을 해석하는 데에 사용된다.

테스트 조건 그룹(Test Condition Group)

테스트 조건 그룹 서브 언어(Sub-language)는 규정, 타이밍 및 레벨의 기술을 모두 패키지화 한다. 타이밍 객체는 종종 파라미터를 사용하여 규정된다. 파라미터들은 다양한 펄스들의 선두 및 후위 에지(leading and trailing edges)를 규정하기 위한 타이밍에 사용될 수 있다. 유사하게, 레벨들은 다양한 전위 레벨의 최대, 최소 및 전형적인 값들을 규정함으로써 파라미터화 될 수 있다. 테스트 조건 그룹(*Test Condition Group; TCG*) 객체는, 이러한 규정에 기초하여, 타이밍 및 레벨의 규정과 인스턴스화를 함께 처리한다.

테스트 조건 그룹 선언은 선택적인 규정 세트를 수록한다. 상기 규정 세트 선언은 인라인된(inlined)(그리고 명명되지 않은) 지역 규정 세트일 수도 있으며, 또는 다른 곳에서 선언된 명명된 규정 세트로의 참조일 수도 있다. TCG 선언의 선택적인 규정 세트 선언은 적어도 하나의 레벨 또는 타이밍 선언이 뒤따른다. 그것은 어떤 순서로든 레벨 및 타이밍 모두를 가질 수도 있다. 그러나, 그것은 하나 또는 그 이상의 레벨 및 타이밍 선언을 가지도록 허용되지는 않았다. 이러한 제한은 구문적으로 강제된다.

TCG에서의 규정 세트 선언은, 그것이 명칭을 갖지 않았다는 점 이외에는 별도로 선언된 규정 세트와 동일하다. 그의 명칭은 묵시적으로 그를 수용한 TCG의 명칭이다. 상기 타이밍 선언은 규정된 타이밍 파일로부터의 타이밍 객체의 단일한 선언이다. 이하에 테스트 조건 그룹과 함께 파일의 예를 개시한다.

```
#-------------------------------------------------------
```

```
# File myTestConditionGroups.tcg

#-------------------------------------------------------

Version 0.1;

Import CHIPlevels.lvl;

Import edges.spec;

Import timing.tim;

Import timing2.tim;

TestConditionGroup TCG1

{

#이 지역 규정 세트는 사용자 정의 선택자들

#"min", "max" 및 "typ"을 사용한다.

#임의의 사용자 정의 명칭을 가진 임의의

#개수의 선택자들이 허용 가능하다.

#

#이 규정 세트는, 타이밍 및 레벨을 초기화하기 위한

#표현에 사용될 수 있는 변수들을 위한 값을 주는

#테이블을 규정한다. 이하에 설정된 규정 세트는

#이하의 테이블에 따라 변수들을 위한 값들을 정의한다. 즉,

# min max typ

# v_cc 2.9 3.1 3.0

# v_ih vInHigh+ 0.0 vInHigh+ 0.2 vInHigh+ 0.1

# v_il vlnLow+ 0.0 vInLow+ 0.2 vInLow+ 0.1

#...

#"vInHigh"와 같은 참조는 사용자 변수의 블록에서

#미리 정의되어 있어야만 한다.

#

#
```

#그러므로, 만약 상기 "max" 선택자가 기능적

#시험에서 선택되었다면, 상기 값의 "max" 행은

#그 변수에 한정되며 v_cc를 3.1에,

#v_ih를 vInHigh+ 2.0에, 등으로 설정한다.

#

#이것은 지역 규정 세트이며 따라서 명칭이

#없다는 점을 주의하여야 한다.

SpecificationSet (min, max, typ)

{

전위에 대한 최소, 최대 및 전형적 규정들

Voltage vcc = 2.9, 3.1, 3.0;

Voltage v_ih = vInHigh + 0.0,

vInHigh + 0.2,

vInHigh + 0.1;

Voltage v_il = vInLow + 0.0,

vInLow + 0.2,

vInLow + 0.1;

#선두 및 후위 타이밍 에지에 대한

#최소, 최대 및 전형적인 규정들.

#1.0E-6uS라는 기저 값은 1picosecond에 대응되며,

#단위와 함께 숫자에 대한 과학적 표기를

#사용하는 예를 제공함.

Time t_le = 1.0E-6 uS,

1.0E-6 uS + 4.0 * DeltaT,

1.0E-6 uS + 2.0 * DeltaT;

Time t_te = 30ns,

30ns + 4.0 * DeltaT,

30ns + 2.0 * DeltaT;

}

#이전에 인입된 CHIP3Levels를 가리킴.

#그것은 상기 파일로부터 인입된 가능한

#많은 레벨 객체들의 하나임.

Levels CHIP3Levels;

#하나의 타이밍인 Timing1을 수록하는

#timing1.tim 파일을 가리킴.

#만약 파일 명칭에 공백 문자가 포함되어 있다면,

#파일 명칭을 따옴표 처리하여야 함.

Timings Timingl;

}

#다른 테스트 조건 그룹

TestConditionGroup TCG2

{

#ClockAndDataEdgesSpecs는 edges.specs 파일에서

#활용 가능한 규정 세트임.

#이하의 선언을 가지고 있다고 가정함. 즉,

SpecificationSet ClockAndDataEdgesSpecs(min, max, typ)

{

Time clock_le = 10.00 uS, 10.02 uS, 10.01 uS;

Time clock_te = 20.00 uS, 20.02 uS, 20.01 uS;

Time data_le = 10.0 uS, 10.2 uS, 10.1 uS;

Time data_te = 30.0 uS, 30.2 uS, 30.1 uS;

}

이 명명된 세트를 참조하는 규정 세트는 이하와 같음.

SpecificationSet ClockAndDataEdgesSpecs;

#인라인된 레벨 선언. 관련된 규정 세트(상기)가

#VInLow, VInHigh, VOutLow 및 VOutHigh와 같은

#변수들을 가지고 있지 않으므로, 그들은

#디폴트 사용자 변수 모음에서 해석하여야 함.

Levels

{

InPins {VIL = VInLow; VIH = VInHigh + 1.0;}

OutPins {VOL = VOutLow/2.0; VOH = VOutHigh;}

}

#이 타이밍은 "timing2.tim"으로부터 유래함.

#타이밍은 상기 규정 세트에 규정된 바와 같이

#클록을 위한 선두 및 후위 에지 타이밍과 데이터를 필요로 함.

Timings Timing2;

}

상기의 예에서, 상기 테스트 조건 그룹 TCG1은, "min", "typ" 및 "max"로 명명된 세 개의 선택자와 함께 규정 세트를 기술하고 있다. 서로 다른 선택자들은 임의의 개수 존재하여도 좋다. 규정 세트의 본문에서, 변수들 v_il, v_ih, t_le 및 t_te는 상기 선택자들에 대응되는 3중의 값들로 초기화된다. 그러므로, 상기 예에서는, 선택자 "min"을 가진 TCG1의 인스턴스는 변수 v_il을 첫번째 수치 값(vInputLow+ 0.0)으로 한정시킬 것이다. 그것은 규정 세트를 위한 선택자는 사용자가 정의하며, 임의의 개수이어도 좋다는 것을 반복한 것이다. 유일한 요건은 다음과 같다. 즉,

규정 세트의 선택자는 고유한 식별자이다.

상기 규정 세트에 규정된 각 값은, 선택자의 세트와 정확히 같은 개수의 구성요소들의 값들의 배열과 관련된다. i번째 선택자를 취함으로써, 각 값은 값들의 관련된 벡터의 i번째 값으로 한정된다.

상기 TCG의 규정 세트에 이어서, 레벨 선언 또는 타이밍 선언 또는 양자 모두가 있을 수 있다. 레벨 선언은 다양한 핀 파라미터에 대한 레벨들을 선언하는 데에 사용된다. 상기 규정 세트에서 식별되는 변수들은, 상기 TCG를 초기화하는 데에 사용되는 선택자에 기초하여 핀 파라미터들에 대한 서로 다른 실제 값들의 동적 한정을 허용하면서, 이러한 레벨들을 설정하는 데에 사용될 것이다.

이것을 예시화하기 위하여, "min" 선택자를 가동시키는 시험을 생각해 보자. 상기 CHIP3Levels의 규정 세트를 참조하면, InPins 그룹의 핀들에 대한 핀 파라미터 "VIH"는 다음의 선언에 의하여 표현 (v_ih+ 1.0)으로 초기화 될 것이다. 즉,

InPins{VIL=v_il; VIH=v_ih+ 1.0;}

이것은 상기 선택자 "min"이 가동될 때 (VInHigh+ 0.0+ 1.0)으로 해석된다. 이와 유사하게, 타이밍 객체는 규정 세트의 변수들의 선택된 값에 기초하여 초기화될 수 있다. 반드시 타이밍 및 레벨 선언들 모두를 가져야 하는 것은 아니다. 이하의 예에 개시된 바와 같이, 어느 것도 혼자서 또는 어떤 순서로든 모두 제시될 수 있다. 즉,

```
#-------------------------------------------------------------

# File LevelsOnlyAndTimingsOnly.tcg

#-------------------------------------------------------------

Version 0.1;

# 레벨만의 테스트 조건 그룹.

TestConditionGroup LevelsOnlyTCG

{

SpecificationSet(Min, Max, Typ)

{

Voltage v_il= 0.0, 0.2, 0.1;

Voltage v_ih = 3.9, 4.1, 4.0;

}

#인라인된 레벨 선언. 관련된 (상기의) 규정 세트가 VInLow, VInHigh, VOutLow 및 VOutHigh와 같은 변수들을 가지고 있지 않기 때문에, 그들은 디폴트 사용자 변수 모음에서 해석하여야 한다.

Levels

{

InPins {VIL=v_il; VIH=v_ih+ 1.0;}

OutPins {VOL=v_il/2.0; VOH=v_ih;}

}

}

#타이밍만의 테스트 조건 그룹

TestConditionGroup TimingsOnlyTCG

{

SpecificationSet(Min, Max, Typ)

{

Time t_le = 0.9E-3, 1.lE-3, 1.OE-3;

}
```

Timings Timing2;

}

그러나, 테스트 조건 그룹에는 하나 이상의 타이밍 및 레벨이 존재해서는 안된다는 점을 주의하여야 한다. 그러므로, 요컨대, 적어도 하나의 타이밍 또는 레벨이 존재하여야 하며, 많아야 각각 하나씩이다.

테스트 조건(Test Conditions)

테스트 조건 객체는 테스트 조건 그룹을 특정의 선택자에 구속시킨다. 일단 테스트 조건 그룹이 상기한 바와 같이 선언되고 나면, 이하에 개시된 바와 같이 테스트 조건 객체를 선언할 수 있게 된다.

TestCondition TCMin

{

TestConditionGroup=TCG1;

Selector=min;

}

TestCondition TCTyp

{

TestConditionGroup=TCG1;

Selector=typ;

}

TestCondition TCMax

{

TestConditionGroup=TCG1;

Selector=max;

}

이들 테스트 조건들은 다음과 같은 테스트 계획에서 인스턴스화 된다. 즉,

#

세 개의 테스트 조건 그룹의 인스턴스를 가리키는

#기능적 시험인 "MyFunctionalTest"를 선언함.

#

Test FunctionalTest MyFunctionalTest

{

#패턴 리스트를 규정함.

PList=pat1Alist;

#임의의 개수의 테스트 조건이 규정될 수 있음.

TestCondition=TCMin;

TestCondition=TCMax;

TestCondition=TCTyp;

}

테스트 조건 그룹에서의 명칭 해석(Name Resoulution in Test Condition Groups)

테스트 조건 그룹에서의 명칭의 해석은 이전에 기술되었다. 그러나, 이러한 규칙들을 반복하여 제시하면 이하와 같다.

1. 명칭이 인가되면(페이지 참조), 그것은 명명된 사용자 변수 모음에서 해석되어야 한다.

2. 만약 명칭이 인가되지 않으면, 그것이 상기 테스트 조건 그룹에서 선언된 경우 상기 명칭은 지역 규정 세트에서, 또는 그것이 테스트 조건 그룹에서 참조된 것이면 상기 명명된 규정 세트에서 해석된다.

3. 상기의 규칙에 의하여 명칭이 해석되지 않으면, 그것은 상기 디폴트 사용자 변수 모음에서 해석된다.

TCG 실행 시간(TCG Runtime)

테스트 조건 그룹은 이하의 실행 시간 구문을 갖는다:

시험(기능적 시험과 같은)은 인스턴스화 된 테스트 조건을 사용하여 그의 규정 세트로부터의 특정 선택자를 가지고 테스트 조건 그룹(TCG)을 참조할 것이다. 이 선택자는, 상기 규정 세트의 각 변수를 선택된 선택자와 관련된 값으로 한정한다. 이렇게 변수들을 그들의 값으로 한정하는 것은 레벨 및 타이밍을 결정하는 데에 사용될 것이다. 테스트 조건 그룹에서 레벨 파라미터는, 바람직하게, 레벨 블록에 제시된 순서에 따라 순차적으로 설정된다. 그러므로, CHIP3Levels 블록에서는, 레벨 파라미터가 설정되는 순서는 다음과 같다(표기: <리소스-명칭>.<리소스-파라미터>). 즉,

InputPins.VIL,

InputPins.VIH,

OutputPins.VIL,

OutputPins.VIH,

Clock.VOL,

Clock.VOH.

이러한 순차적인 순서는, 테스트 작성자가 전력 공급기의 명시적인 전력 공급 순서를 제어할 수 있게 해준다. 게다가, 레벨 항목이 하나의 핀에 대하여 동일한 핀 파라미터를 명명하면서 두 번 나타나면, 그 핀 파라미터는 두 번 설정된다. 이것은 또한 프로그램적으로도 발생할 수 있다.

만약 파라미터가 다음과 같은 슬루 문장(Slew statement)에 의하여 설정된다면,

VCC=Slew(0.01, 2.0V);

그것은, VCC가 램프 방식으로, 초당 ±0.01 volts의 전위 슬루율로 현재 값으로부터 최종값인 2.0volts에 도달할 것이라는 것을 의미한다.

규정 세트의 변수들은 테스트 조건 그룹의 타이밍 객체로 전달될 수도 있다. 그러면, 상기 타이밍 객체는 선택된 변수들에 기초하여 초기화 될 것이다. 그러한 메커니즘은, 예를 들어 파형의 선두 및 후위 에지를 규정함으로써 타이밍 객체를 개별화하는 데에 사용될 수 있다.

테스트 조건 그룹을 위한 C++

상기한 규칙들을 가지고, 상기 테스트 조건 그룹은 C++ 테스트 조건 그룹에서 선언될 수 있으며, 이하와 같이 초기화 될 수 있다:

상기 테스트 조건 그룹에 대한 규정 세트를 설정하는 상기 테스트 조건 그룹 멤버 함수에 대한 호출이 작성된다. 즉,

Status setSpecificationSet(SpecificationSet *pSpecificationSet);

이것은 지역 규정 세트, 또는 명명된 규정 세트, 또는 널(null)(만약 아무 것도 없다면)의 어느 하나가 될 수 있다.

상기 테스트 조건 그룹에 대한 레벨 객체를 설정하는 테스트 조건 그룹 멤버 함수에 대한 호출이 작성된다. 즉,

Status setLevels(Levels *pLevels);

이것은 지역적으로 선언된 레벨 객체, 외부적으로 선언된 레벨 객체, 또는 널(만약 아무 것도 없다면)의 어느 하나가 될 수 있다.

상기 테스트 조건 그룹에 대한 타이밍 객체를 설정하는 테스트 조건 그룹 멤버 함수에 대한 호출이 작성된다. 즉,

Status setTimings(Timings *pTimings);

이것은 외부적으로 선언된 타이밍 객체, 또는 널(만약 아무 것도 없다면)의 어느 하나가 될 수 있다.

빈 정의(Bin Definitions)

상기 빈 정의 클래스는, 많은 피시험 디바이스를 시험한 결과를 요약한 카운터(counter)의 모음인, 빈(bin)을 정의한다. 피시험 디바이스의 시험 과정에서, 상기 피시험 디바이스는, 예컨대 특정 시험의 결과를 지시하는 임의의 빈으로 설정될 수 있다. 시험이 진행됨에 따라, 상기 피시험 디바이스는 다른 빈으로 설정될 수도 있다. 피시험 디바이스가 최종적으로 설정된 빈은 시험의 종료시에 설정된 마지막의 것이다. 이 최종 빈에 대한 카운터가 이 피시험 디바이스의 시험의 종료시에 증분된다. 빈 정의를 가진 별도의 파일은 .bdefs라는 접미사를 가져야 한다.

빈 정의는 바람직하게 계층적(hierarchical)이다. 예를 들어, 최외곽 레벨에서, 패스(Pass) 및 페일(Fail)이라고 명명된 두 개의 빈을 가진 패스페일 빈(PassFailBins)이 있을 수 있다. 그러면, 복수의 하드 빈(HardBins)이 있어서, 그들 중 일부는 패스 빈으로 매핑되고, 나머지는 페일 빈으로 매핑된다. 하드 빈은 패스페일 빈의 세부(refinement)라고 할 수 있다. 마지막으로, 하드 빈의 세부인 많은 개수의 소프트 빈(SoftBins)이 있을 수 있는데, 그들중 대부분은 동일한 하드 빈으로 매핑된다. 이하는 빈의 계층을 예시하는 예이다. 즉,

#----------

File CHIPbins.bdefs

```
#-------------------------------------------------------------

Version 1.2.3;

BinDefs

{

#하드 빈은 빈의 최외곽 레벨이다.

#그들은 어떠한 다른 빈의 세부도 아니다.

BinGroup HardBins

{

"3GHzPass": "DUTs passing 3GHz";

"2.8GHzPass": "DUTs passing 2.8GHz";

"3GHzFail": "DUTs failing 3GHz";

"2.8GHzFail": "DUTs failing 2.8GHz";

LeakageFail: "DUTs failing leakage";

}

#소프트 빈은 다음 레벨의 세부이다.

#소프트 빈은 하드 빈의 세부이다.

BinGroup SoftBins:HardBins

{

"3GHzAllPass":

"Good DUTs at 3GHz", "3GHzPass";

"3GHzCacheFail":

"Cache Fails at 3GHz", "3GHzFail";

"3GHzSBFTFail":

"SBFT Fails at 3GHz", "3GHzFail";

"3GHzLeakage":

"Leakages at 3GHz", LeakageFail;

"2.8GHzAllPass":
```

"Good DUTs at 2.8GHz", "2.8GHzPass";

"2.8GHzCacheFail":

"Cache Fails at 2.8GHz", "2.8GHzFail";

"2.8GHzSBFTFail":

"SBFT Fails at 2.8GHz", "2.8GHzFail";

"2.8GHzLeakage":

"Leakages at 2.8GHz", LeakageFail;

}

}

상기 예에 의하면, 가장 기초적인 빈은 빈 그룹의 하드 빈이다. 만약 어떤 다른 빈 그룹이 X의 세부라면, 빈 그룹 X는 기초 빈(base bin)의 그룹이라 한다. 그리하여, 빈 그룹인 소프트 빈이 하드 빈의 세부이므로, 빈 그룹 하드 빈은 기초 빈의 그룹이다. 소프트 빈의 빈들은 리프(leaf) 빈이라 불린다. 빈 그룹 Y는, 다른 빈 그룹이 Y의 세부가 아니라면, 리프 빈의 그룹이라 한다.

그 내부의 단일한 빈 그룹 Z를 가진 빈 정의 블록(BinDef block)의 축퇴(degenerate)의 경우는, Z가 리프 빈들의 그룹 뿐만 아니라 대부분의 기초 빈들의 그룹이 되게 한다. 빈 그룹 명칭은 범위에 있어서 전역적이다. 빈 정의 블록은 임의의 개수가 존재할 수 있지만, 선언된 빈 그룹은 독특해야 한다. 하나의 빈 정의 블록으로부터의 빈 그룹은 다른 빈 정의 블록으로부터의 빈 그룹의 세부가 되도록 허용된다. 그러므로, 상기한 예에서, 소프트 빈은 하드 빈으로부터의 별도의 빈 정의 블록 내에 있을 수 있다. 그러나, 가독성을 위하여, 모든 선언된 빈 그룹을 가진 하나의 빈 정의 블록을 가질 것을 추천한다.

상기한 계층은, 다른 빈 그룹을 추가함으로써, 얼마나 많은 피시험 디바이스들이 양품이고 불량품인지를 계수하도록 확장될 수 있다.

#---

File CHIPbins.bdefs

#---

Version 1.2.3;

BinDefs

{

#패스페일 빈은 최외각 레벨의 빈이다.

#그들은 다른 어떠한 빈의 세부도 아니다.

BinGroup PassFailBins

{

Pass:"Count of passing DUTS.";

Fail:"Count of failing DUTS.";

}

#하드 빈은 다음 레벨의 세부이다.

#"HardBins:PassFailBins"로 표기한 바와 같이,

#하드 빈은 패스페일 빈의 세부이다.

BinGroup HardBins:PassFailBins

{

"3GHzPass": "DUTs passing 3GHz", Pass;

"2.8GHzPass": "DUTs passing 2.8GHz", Pass;

"3GHzFail": "DUTs failing 3GHz", Fail;

"2.8GHzFail": "DUTs failing 2.8GHz", Fail;

LeakageFail: "DUTs failing leakage", Fail;

}

#소프트 빈은 그 다음 레벨의 세부이다.

#소프트 빈은 하드 빈의 세부이다.

BinGroup SoftBins:HardBins

{

"3GHzAllPass":

"Good DUTs at 3GHz", "3GHzPass";

"3GHzCacheFail":

"Cache Fails at 3GHz", "3GHzFail";

"3GHzSBFTFail":

"SBFT Fails at 3GHz", "3GHzFail";

"3GHzLeakage":

"Leakages at 3GHz", LeakageFail;

"2.8GHzAllPass":

"Good DUTs at 2.8GHz", "2.8GHzPass";

"2.8GHzCacheFail":

"Cache Fails at 2.8GHz", "2.8GHzFail";

"2. 8GHzSBFTFail":

"SBFT Fails at 2.8GHz", "2.8GHzFail";

"2.8GHzLeakage":

"Leakages at 2.8GHz", LeakageFail;

}

}

이때, 가장 기초적인 빈은 빈 그룹인 패스페일 빈이다. 그들은 전형적으로 어떠한 빈의 세부도 아니다. 빈 그룹 하드 빈은, 패스페일 빈의 세부이며, 그 또한 기초 빈이다. 소프트 빈은 하드 빈의 세부이며, 리프 빈의 그룹이다. 상기한 예는 계층 내의 단 세 개의 빈 그룹만을 가지고 있다. 이하는 보다 복잡한 계층이다. 즉,

BinDefs

{

#가장 기초적인 빈 그룹

BinGroup A {...}

#A의 세부인 기초 빈의 그룹

BinGroup Ax:A {...}

#Ax의 세부인 리프 빈의 그룹

BinGroup Axx:Ax {...}

#A의 세부인 기초 빈의 그룹

BinGroup Ay:A {...}

#Ay의 세부인 리프 빈의 그룹

BinGroup Ayy:Ay {...}

#가장 기초적인 빈 그룹

BinGroup B {...}

#B의 세부인 리프 빈의 그룹

BinGroup Bx:B {...}

}

이 예에서, Ax 및 Ay는 A의 세부이며, Axx는 Ax의 세부이며, Ayy는 Ay의 세부이다. 이 예는 또한, 빈 그룹 B 및 Bx를 제공하는데, 여기서 Bx는 B의 세부이다. 패스페일 빈, 하드 빈, 및 소프트 빈으로 명명된 빈 그룹들과 함께 상기 빈 정의 선언은 이 섹션에서 계속되는 예로서 사용될 것이다.

빈 그룹의 각 빈은 다음을 갖는다. 즉,

1. 식별자 또는 문자열 리터럴의 어느 하나인 명칭

2. 이 빈이 요약하고 있는 것에 관한 기술, 그리고

3. 만약 이 빈이 세부 빈 그룹에 있다면, 기초 빈으로도 알려진, 세부인 빈의 명칭.

패스페일 빈(PassFailBins)의 두 개의 빈은 "패스" 및 "페일"로 명명된다. 하드 빈의 다섯 개의 빈은, "3GHzPass", "2.8GHzPass", "3GHzFail", "2.8GHzFail", "LeakageFail"로 명명된다. 빈의 명칭은 리터럴 문자열 또는 식별자일 수 있다. 빈의 명칭은 빈 그룹 내에서 고유해야만 하지만, 빈 그룹에 걸쳐서는 중첩될 수 있다. 그러나, 빈 그룹의 명칭은 범위에 있어서 전역적이므로 테스트 계획에 걸쳐서 고유해야만 한다.

다섯개의 하드 빈들 중에서, 상기 "3GHzPass" 및 "2.8GHzPass"는 모두 상기 패스페일 빈의 "통과" 빈으로 매핑된다. 하드 빈의 나머지는 상기 패스페일 빈의 "페일" 빈으로 매핑된다.

마지막으로, 여덟 개의 소프트 빈이 존재한다. 소프트 빈 기능적 시험(soft bin functional test; SBFT) 및 캐시(Cache)에 대한 3GHz에서의 두 가지 실패는 "3GHzFail"이라는 하드 빈으로 매핑된다. 유사하게, SBFT 및 캐시(Cache)에 대한 2.8GHz에서의 두 가지 실패는 "2.8GHzFail"이라는 하드 빈으로 매핑된다. 누설(Leakage)에 기인한 두 가지 실패는, 그들이 발생한 속도에 무관하게, 같은 "LeakageFail"이라는 하드 빈으로 매핑된다. 예를 들어, (최외각 레벨에서의) 가장 거친(coarsest) 시험은 피시험 디바이스가 시험을 통과하느냐 실패하느냐의 여부이다. 예를 들어, 세부는, 특정 주파수, 예컨대 3GHz에서, 상기 피시험 디바이스가 시험을 통과하느냐 실패하느냐의 여부이다.

후술하는 바와 같이, 빈들은 테스트 계획 흐름 항목(TestPlanFlowItem)에서 피시험 디바이스에 할당된다. 테스트 계획 흐름 항목은, 테스트 계획이 시험을 실행함으로써 돌아 오는 특정 결과를 획득한 결과로서 발생하는 액션(action) 및 전이(transition)를 기술하는 결과 구절(Result Clause)을 갖는다. 빈 설정 문장(SetBin statement)이 발생하는 것은 이 시점이다. 즉,

#흐름 항목의 결과 구절. 추후 기술됨.

Result 0

{

#시험의 실행으로부터 0을 돌려 받은 경우

#취해질 액션

#상기 피시험 디바이스가 훌륭하다는 것을 표현하는

#SoftBin."3GHZPass"로 빈을 설정함.

SetBin SoftBins."3GHzPass";

}

많은 빈 설정 문장은 피시험 디바이스에 시험을 실행중인 동안에 실행될 수 있다. 시험이 완전히 완결된 때, 실행 시간은 그 피시험 디바이스에 대하여 설정된 최종 빈과 모든 그의 세부를 위한 카운터를 증분시킬 것이다. 그 시험이 진행되는 동안 실행되는 이하의 빈 설정 문장을 가진 피시험 디바이스를 생각해 보자. 즉,

SetBin SoftBins."3GHzSBFTFail";

SetBin SoftBins."2.8GHzAllPass";

이 피시험 디바이스는 3GHz 캐시 및 누설 시험을 통과했지만, SBFT 시험에 실패했고, 그래서 상기 "3GHzSBFTFail" 빈으로 할당되었다. 그리고나서, 2.8GHz에서 시험되었는데, 모든 시험을 통과하였다. 그래서, 상기 최종 빈은 상기 "2.8GHzAllPass" 빈으로 할당되었는데, 그것은 소프트 빈의 세트이다. 이 최종 할당은 다음의 빈들의 카운터를 증분시킬 것이다. 즉,

1. SoftBins."2.8GHzAllPass"

2. HardBins."2.8GHzAllPass"의 세부인 것

3. PassFailBins."Pass"의 세부인 것.

시험이 완료된 때에, 실행 시간은 피시험 디바이스의 최종 빈 할당과, 그의 모든 세부인 빈들에 대한 카운터를 증분시킬 것이다.

빈 설정 문장은 단지 리프 빈 상에서만 허용된다. 기초 빈을 설정하는 것은 규칙에 어긋난다. 상기의 카운터 증분 구문은 다음을 확인시켜준다. 즉,

1. 만약 상기 빈이 리프 빈이라면, 그것은 피시험 디바이스의 시험 종료시에 이 빈에 대하여 빈 설정 문장이 실행된 횟수이다.

2. 만약 상기 빈이 기초 빈이라면, 그것은 그의 세부인 빈의 카운터들의 합이다.

그러므로, 상기한 예에서는, 단지 소프트 빈들만이 빈 설정 문장에서 허용된다. HardBins."LeakageFail"에 대한 카운터는, SoftBins."3GHzLeakageFail" 및 SoftBins."2.8GHzLeakageFail"에 대한 카운터들의 합이다. 이하는 빈 정의에 관한 몇 가지 규칙들이다. 즉,

1. 빈 정의 선언은 복수의 빈 그룹 선언을 포함한다.

2. 각 빈 그룹 선언은 명칭, 선택적인 세부인 빈 그룹의 명칭과 이어서 빈 선언의 블록을 갖는다.

3. 빈 선언은, 명칭, 이어서 기술(description), 이어서 이 빈이 세부인 기초 빈의 명칭을 포함한다.

4. 빈의 명칭은 문자열 리터럴이나 식별자이다. 공백 문자열은 유효한 빈의 명칭이 될 수 없다. 빈의 명칭은 빈 그룹 선언 내의 명칭들 중에서 고유해야만 하지만, 다른 빈 그룹 선언에서는 동일한 명칭이 사용될 수 있다.

5. 만약 빈 그룹 선언 Xxx가 다른 빈 그룹 선언 Yyy의 세부라면, Xxx의 모든 빈 선언은 Yyy로부터의 기초 빈의 명칭을 선언해야 한다. 그러므로, 소프트 빈은 하드 빈의 세부인 것으로 선언되므로, 소프트 빈의 빈 선언들 각각은, 하드 빈의 빈의 세부이다.

6. 패스페일 빈과 같이, 다른 빈 그룹 선언의 세부가 아닌 빈 그룹 선언은, 바람직하게 기초 빈을 선언하지 않는 빈 선언을 가질 것이다.

Bbb 빈은 Bbb가 세부인 빈들 전체의 세트인 기초들(bases)의 세트를 갖는다. 그것은 형식적으로 다음과 같이 정의된다. 즉,

1. 만약 Aaa가 Bbb의 기초 빈이라면, Aaa는 Bbb의 기초들의 세트이다.

2. Aaa의 기초는 어느 것이나 Bbb의 기초들의 세트에 속한다.

빈 그룹의 명칭은 테스트 계획 내에서 전역적이다.

빈 명칭은 빈 그룹에 대하여 지역적이다.

빈 설정 문장은 리프 빈에 대하여만 허용된다.

빈 정의를 위한 C++

상기한 규칙들을 가지고, 빈 정의 선언의 빈 그룹 선언의 각각에 대하여 빈 그룹이라는 객체 형식이 구성될 수 있다. 빈 그룹이라는 클래스는 리프 빈 그룹(LeafBinGroup)이라는 서브클래스(subclass)를 가질 것이다. 이들 두 가지 클래스들의 동작은, BinGroup::incrementBin이 C++ 보호 동작(protected operation)인 반면 LeafBinGroup::incrementBin이 C++ 공개 동작(public operation)이라는 점 이외에는 동일하다.

이하는 다른 어느 빈 그룹의 세부도 아닌 빈 그룹 또는 리프 빈 그룹을 구축하는 디폴트 구축자이다.

구축자:

BinGroup(BinGroup& baseBinGroup);

LeafBinGroup(BinGroup& baseBinGroup);

이것은, 주어진 기초 빈 그룹의 세부인 빈 그룹을 구축한다.

메소드:

Status addBin(constString& binNamen,

const String& description,

const String& baseBinName);

이것은 빈 및 그의 기술을 정의하기 위한 것이다. 만약 그것이 가장 기초적인 빈(most base bin)이라면, 상기 기초 빈 명칭(baseBinName) 파라미커는 공백 문자열이어야만 한다.

빈 카운터를 증분시키는 메소드:

Status incrementBin(const String& binName);

이 동작은 이 빈과, 이 빈의 기초인 다른 모든 빈들에 대한 계수를 증분시킬 것이다. 상기 동작은 빈 그룹이라는 클래스에서 보호되며, 리프 빈 그룹이라는 클래스에서는 공개된다.

빈 카운터를 리셋시키는 메소드:

Status resetBin(const String& binName);

이 동작은 이 빈과 이 빈의 기초인 모든 빈들에 대한 카운터를 리셋시킨다.

빈에 관한 정보를 획득하는 메소드:

Status getBinDescription(const String& binName,

String& description);

Status getBaseBin(const String& binName,

BinGroup* pBaseBinGroup,

String& baseBinName);

Status getBinValue(const String& binName,

unsigned int& value);

현재 정의된 모든 빈 명칭을 획득하기 위하여 반복자(iterator)가 제공될 것이다.

테스트 계획은, 각 빈 그룹 선언을 위한 빈 그룹 멤버의 번호를 포함할 것이다. 상기 빈 정의를 위한 C++은 다음과 같다. 즉,

//테스트 계획 구축자(TestPlan constructor)

TestPlan::TestPlan()

:m_PassFailBins(), // 디폴트 구축자

m_HardBins(&m_PassFailBins),

m_SoftBins(&mHardBins)

{}

//빈 초기화(Bin initializations)

m_PassFailBins.addBin("Pass", "Count of passing DUTS.", "");

m_PassFailBins.addBin("Fail", "Count of failing DUTS.", "");

m_HardBins.addBin("3GHzPass", "Duts passing 3GHz", "Pass");

...

테스트 계획을 위한 상태(state)는, 정의되지 않은 빈 그룹(*undefined* BinGroup)(널(NULL)) 및 정의되지 않은 빈 명칭(공백 문자열)인 m_currentBin으로 초기화된 m_pCurrentBinGroup을 포함한다. 빈 설정 문장이 실행될 때마다, 상기 m_pCurrentBinGroup은, 다음을 호출함으로써, 지시된 명명된 빈 그룹 및 상기 그룹의 명명된 빈에 대한 m_currentBin으로 변경된다. 즉,

//SetBin SoftBins."3GHzAllPass";의 번역

pTestPlan->setBin("SoftBins", "3GHzAllPass");

테스트 계획이 실행을 완료할 때, 그것은, 다음을 호출하여 이 빈과 그의 모든 기초 빈들이 그들의 카운터들을 증분시키게 한다. 즉,

m_pCurrentBinGroup->incrementBin(m_currentBin);

상기 빈 그룹 카운터들은, 테스트 계획이 합성될 때 리셋되지만, 시험이 실행될 때마다 재초기화되지는 않는다. 상기 카운터들은 BinGroup::resetBin의 명시적인 호출에 의하여 리셋될 수 있다.

C. 테스트 계획

상기 테스트 계획은 상기 테스트 프로그램의 주요 구조로서 생각될 수 있다. 상기 테스트 계획은 유사한 구성을 인라인으로 정의할 뿐만 아니라 파일을 인입할 수도 있다. 그러므로, 추가적인 전역 구성요소(global)들을 인라인으로 선언할 뿐만 아니라 특정 전역 구성요소들의 정의가 주어진 파일을 인입할 수도 있다.

C1. 테스트 계획 흐름 및 흐름 항목(Test Plan Flows and Flow Items)

테스트 계획의 결정적인 구성요소들 중 하나는 흐름(Flow)이다. 흐름은 유한 상태 머신(finite state machine)을 캡슐화한다. 그것은, 흐름 가능 인터페이스(IFlowable) 객체를 실행시키고 이어서 다른 흐름 항목으로 전이하는 복수의 흐름 항목(*FlowItems*)을 포함한다. 특정의 흐름 가능 인터페이스의 실행은 'IFlowable' 인터페이스를 구현하는 객체의 실행을 수반한다. 'IFlowable' 인터페이스를 구현하는 전형적인 객체는 테스트 및 흐름(Tests and Flows) 그 자체들이다.

그러므로, 흐름은, 시험 및 다른 흐름을 실행시키는 흐름 항목을 가지며, 이어서 다른 흐름 항목으로 전이한다. 그것은 또한 특정의 흐름 가능 인터페이스의 실행의 결과로부터 야기되는 다양한 리턴(return)에 사용자 주문형 루틴(user custominzed routine)을 호출할 수 있는 기회를 제공한다. 따라서, 전형적으로 흐름은 다음의 형태를 취한다. 즉,

#

#FlowTest1은, MyFunctionalTest1의 Min, Typ 및

#Max 특색(flavor)을 위한 유한 상태 머신을 구현한다.

#성공하면, 그것은 Test1Min, Test1Typ, Test1Max를

#시험하고, 그 호출자에게 성공적인 상태로서 0을 리턴한다.

#실패하면, 실패한 상태로서 1을 리턴한다.

#

#상기 시험 MyFunctionalTest1Min,...이 모두

#결과 0(통과), 1 및 2(페일의 한 쌍의 레벨들)을 리턴한다고 가정하자.

Result 0 Result 1 Result 2

Test1Min Test1Typ return 1 return 1

Test1Typ Test1Max return 1 return 1

Test1Max return 0 return 1 return 1

#

Flow FlowTest1

{

FlowItem FlowTest1_Min MyFunctionalTest1Min

{

```
Result 0
{
Property PassFail="Pass";
IncrementCounters PassCount;
GoTo FlowTest1_Typ;
}
Result 1
{
Property PassFail="Fail";
IncrementCounters FailCount;
Return 1;
}
#만약 MyFunctionalTest1Min이
#2, 5, 6, 7, -6, -5 또는 -4의 어느 하나를 리턴한다면,
#이 결과 블록이 실행된다.
Result 2, 5:7, -6:-4
{
Property PassFail="Fail";
IncrementCounters FailCount;
Return 1;
}
}
FlowItem FlowTest1_Typ {...}
FlowItem FlowTest1_Max {...}
}
```

상기 흐름 FlowTest1의 동작은 다음과 같다. 즉,

1. 흐름 항목 FlowTest1_Min을 실행함으로써 시작한다.

2. FlowTest1_Min은 기능적 시험 MyFunctionalTestMin을 실행한다. 이 시험의 상세는 이하에서 전체 테스트 계획이 제시될 때 설명한다.

3. 이 시험을 실행함으로써 아홉가지의 결과, 즉 0, 1, 2, 5, 6, 7, -6, -5 또는 -4가 예기된다. 처음 두 가지 절과 구절은 각각 0 및 1을 처리하며, 세 번째는 상기 결과값의 나머지 모두를 처리한다.

4. 만약 결과 "0"(통과)가 발생하면, FlowTest1_Min은 통과 카운터(PassCounter)를 증분시킨다. 그리고, 새로운 흐름 항목 FlowTest1_Typ으로 전이한다.

5. 만약 결과 "1" 또는 결과 "2"가 발생하였다면, FlowTest1_Min은 페일 카운터(FailCounter)를 증분시키고 흐름으로부터 리턴한다.

6. FlowTest1_Typ도 동일한 방식으로 동작하고, 성공하면 FlowTest1_Max를 호출한다.

7. FlowTest1_Max도 동일한 방식으로 동작하고, 성공하면 성공적인 결과("0")을 가지고 FlowTest1으로부터 리턴한다.

그러므로, FlowTest1은, 성공적인 실행시에, Test1의 최소의, 전형적인 그리고 최대의 버전들을 통해 장치를 실행시키고, 이어서 리턴한다. FlowTest2도 유사한 방식으로 동작한다.

상기한 흐름은 기본적으로 상태(*States*) 및 전이(*transition*)을 가진 유한 상태 머신을 기술한다. 흐름 항목은 기본적으로 상태이며, 그것은 다음을 수행한다. 즉,

1. 흐름 가능 인터페이스를 실행한다(그것은 이전에 정의된 흐름 또는 시험 또는 상기의 규칙에 따라 C++로 구현될 수 있는 사용자가 정의한 흐름일 수도 있다.).

2. 상기 흐름 가능 인터페이스의 실현은 수치 결과(numeric result)를 리턴한다. 결과에 기초하여, 소정의 액션(*actions*)이 발생하고, 이어서 다음 둘 중의 어느 하나가 발생한다. 즉,

a. 상기 흐름은 수치 결과를 가지고 호출자에게 복귀한다.

b. 상기 흐름은 다른 상태(흐름 항목)로 전이함으로써 지속된다.

그러므로, 흐름 항목은 다음의 컴포넌트를 갖는다. 즉,

흐름 항목(FlowItem)은 명칭을 갖는다.

흐름 항목은 실행될 흐름 가능 인터페이스(IFolwable)를 갖는다.

흐름 항목은 숫자 또는 결과 구절(result clause)을 갖는다.

흐름 항목의 각 결과 구절은 액션을 제공하며, 전이로 끝나며, 하나 또는 그 이상의 결과 값과 관련된다.

이러한 항목들은 흐름 항목에서 다음과 같은 구문이다. 즉,

FlowItem<name><실행될 IFlowable>

{

Result<하나 또는 그 이상의 결과 값>

{

<이 결과 값에 대한 액션>

<이 결과 값에 대한 전이>

}

Result<하나 또는 그 이상의 다른 결과 값>

{

...

}

...

}

상기 *실행될* IFlowable은 시험, 또는 사용자가 정의한 IFlowable, 또는 흐름의 어느 하나일 수 있다. 결과에 대한 액션은 다음 중 어느 하나일 수 있다.

GUI 도구에 의하여 사용되는 문자열 값을 갖는 엔티티(entity)를 속성 결과로 설정하기 위한 *속성 액션*(*Property Action*). 이것은 상기 FlowTest1의 다음 예에서 볼 수 있다. 즉,

Property PassFail="Pass";

속성은 기본적으로, 결과 구절과 관련된 명명된 문자열 또는 정수 값을 갖는 엔티티이다. 그들은 임의의 개수만큼 존재할 수 있으며, 사용자가 이 결과와 관련된 정보를 디스플레이하기 위하여 사용할 GUI와 같은 도구에 의하여 바람직하게 사용된다. 그들은 시험 또는 시험의 흐름의 실제 결과에 영향을 주지 못한다.

소정 개수의 카운터를 증분시키는 *카운터 액션*(*Counter Action*). 이것은 상기 예에서 다음에 의하여 볼 수 있다. 즉,

IncrementCounters PassCount;

임의의 또는 사용자 루틴을 호출하는 *루틴 호출 액션*(*Routine Call Action*). 이에 관하여는 후술한다.

마지막으로, 흐름 항목은, 제어를 다른 흐름 항목으로 전달하는 GoTo 문장 또는 제어를 호출자에게 되돌리는 리턴 문장의 어느 하나일 수 있는 전이(*Transition*)를 갖는다.

미리 정의된 흐름들(Predefined Flows)

흐름 객체의 전형적인 사용은 시험의 시퀀스를 정의하는 것이다. 이어서, 이 시퀀스는, 테스트 계획 서버(Test Plan Server; TPS)에서 발생하는 이벤트(event), 즉 실행 테스트 계획 이벤트의 결과로서 실행된다. 각 사이트 제어기의 테스트 계획 서버는 상기 사용자의 테스트 계획을 실행한다. 그러나, 흐름 객체는 다른 이벤트에도 응답하여 실행된다. 괄호 안의 명칭이 흐름들을 이 이벤트들에 할당하는 데에 사용되는 명칭이다.

1. 시스템 로드 흐름(SysLoadFlow). 이 흐름은, 테스트 계획이 하나 또는 그 이상의 사이트 제어기상으로 로드되었을 때 상기 시스템 제어기상에서 실행된다. 그것은, 어떠한 사이트 제어기상으로도 상기 테스트 계획이 실제로 로드되기 전에 실행된다. 이 흐름은 테스트 계획 개발자가 상기 시스템 제어기로부터 유래하여야만 하는 액션들을 정의할 수 있게 한다. 그러한 액션들은 패턴 파일의 브로드캐스트 로드(broadcast load), 교정 액션 등이 포함된다.

2. 사이트 로드 흐름(SiteLoadFlow). 이 흐름은, 테스트 계획이 상기 사이트로 로드되고 초기화된 후에 상기 사이트 제어기상에서 실행된다. 이것은 임의의 사이트에 특유한 초기화가 발생되도록 한다.

3. 로트 시작/종료 흐름(LotStartFlow/LotEndFlow). 이들 흐름은, 상기 테스트 계획 서버가 새로운 로트(lot)의 시작을 통보 받았을 때 상기 사이트 제어기상에서 실행된다. 이것은 전형적으로 데이터 스트림을 로트 특유적 정보로 주석을 다는 생산 환경에서 사용된다.

4. 피시험 디바이스 변경 흐름(DutChangeFlow). 이 흐름은 사이트 제어기의 피시험 디바이스 정보가 변경되었을 때, 상기 사이트 제어기상에서 실행된다. 또한, 이것은 전형적으로 데이터 스트림을 로트 특유적 정보로 주석을 다는 생산 환경에서 사용된다.

5. 테스트 계획 시작/종료 흐름(TestPlanStartFlow/TestPlanEndFlow). 이들 흐름들은, 테스트 계획 서버가 현재의 시험 흐름을 시작하도록 지시받은 때, 그리고 그 흐름이 그 실행을 끝마친 때에, 상기 사이트 제어기 상에서 실행된다.

6. 시험 시작/종료 흐름(TestStartFlow/TestEndFlow). 이들 흐름들은, 상기 시험 흐름이 새로운 시험을 실행시키기 시작한 때, 그리고 그 시험이 그 실행을 끝마친 때에 상기 사이트 제어기 상에서 실행된다.

7. 시험 흐름(TestFlow). 이 흐름은, 상기 테스트 계획 서버가 "테스트 계획 *실*행(Execute Test Plan)"이라는 메시지를 수신할 때 실행되는 메인 흐름 객체(main Flow object)이다.

만약 사용자가, 시험 흐름 또는 다른 미리 정의된 흐름의 어느 하나가 아닌 사용자의 테스트 계획을 정의한다면, 그것을 실행시키는 바람직한 방법은 이들 미리 정의된 흐름의 어느 하나의 전이 상태(transition state)에 그것을 포함시키는 것이다.

테스트 계획의 예

상기 예에서는, 상기 흐름에 의하여 구현되는 유한 상태 머신을 기술하는 코멘트(comment)와 함께 흐름이 주어졌다. 상기 유한 상태 머신은 전이 행렬로서 주어진다. 상기 행렬의 행들은 흐름 항목들에 대응되며, 열들은 결과들에 대응된다. 상기 행렬의 행의 엔트리(entry)들은, 리턴된 결과가 행에 규정된 값일 때, 상기 행의 흐름 항목으로부터 전이된 흐름 항목을 지시한다.

세 개의 흐름들, 즉 FlowTest1, FlowTest2 및 FlowMain을 가진 테스트 계획이 이하에 개시된다. FlowTest1은 상기한 바와 같이 동작된다. 그것은, "min", "typ" 및 "max"라는 구성들의 각각에서 MyFunctionalTest1이라 명명된 시험을 실행할 것이다. 이와 유사하게, FlowTest2는 이들 구성들의 각각에서 MyFunctionalTest2를 실행할 것이다. 마지막으로 FlowMain은 FlowTest1 및 FlowTest2를 실행할 것이다. 상기 유한 상태 머신 행렬은 이들 흐름들의 각각의 시작에서 코멘트에 제공된다.

```
#-------------------------------------------------------

# File mySimpleTestPlan.tpl

#-------------------------------------------------------

Version 0.1;

Import xxx.pin; #핀들

#제한하는 값들을 제공하는 상수들 및 변수들.

Import limits.usrv;

#테스트 조건 그룹을 인입함.

Import myTestConditionGroups.tcg;

#일부 빈 정의를 인입함.
```

```
Import bins.bdefs;

#-------------------------------------------------------------

# 테스트 계획의 시작

#-------------------------------------------------------------

TestPlan Sample;

#이 블록은, 시험 선언에서 사용된 패턴 리스트의

#파일 인가 명칭들(Pattern Losts file-qualified names) 및

#패턴 리스트의 변수들을 정의한다. 패턴 리스트 변수들은

#개별화가 시험될 때까지 유예된다.

PListDefs

{

#파일 인가 패턴 리스트 명칭들

pl1A.plist:pat1Alist,

pl2A.plist:pat2AList

}

#이 테스트 계획에서의 시험을 위한 소켓

#(이것은 인입되지 않지만, 활성화 시에 해석된다.):

SocketDef=mytest.soc;

#인라인으로 특정 사용자 변수를 선언함.

UserVars

{

#현재의 시험을 위한 문자열 명칭

String CurrentTest="MyTest";

}

TestCondition TC1Min

{

TestConditionGroup=TCG1;
```

```
Selector=min;
}
TestCondition TC1Typ
{
TestConditionGroup=TCG1;
Selector=typ;
}
TestCondition TC1Max
{
TestConditionGroup=TCG1;
Selector=max;
}
#TC2Min, TC2Typ, TC2Max 등에 대하여도 유사함.
#
#기능적 시험을 선언함. "FunctionalTest"는 시험을
#실행하는 C++ 테스트 클래스를 가리키며,
#결과로서 0, 1 또는 2를 리턴함. 상기 테스트 조건 그룹 TCG1은
#테스트 조건인 TC1Min을 가리킴으로써 "min" 선택자로 선택됨.
#
Test FunctionalTest MyFunctionalTest1Min
{
PListParam=patlAList;
TestConditionParam=TC1Min;
}
#"typ"으로 TCG1을 선택하는 다른 기능적 시험.
Test FunctionalTest MyFunctionalTest1Typ
{
```

```
PListParam=patlAList;
TestConditionParam=TC1Typ;
}
#"max"로 TCG1을 선택하는 다른 기능적 시험
Test FunctionalTest MyFunctionalTest1Max
{
PListParam=pat1AList;
TestConditionParam=TC1Max;
#이제, "min"으로 TCG2를 선택함.
Test FunctionalTest MyFunctionalTest2Min
{
PListParam=pat2AList;
TestConditionParam=TC2Min;
}
#"typ"로 TCG2를 선택하는 경우와, "max"로 TCG2를 선택하는 경우도 유사함.
Test FunctionalTest MyFunctionalTest2Typ
{
PListParam=patlAList;
TestConditionParam=TC2Typ;
}
Test FunctionalTest MyFunctionalTest2Max
{
PListParam=pat1AList;
TestConditionParam = TC2Max;
}
#
#여기서, 다음의 테스트 객체가 정의됨.
```

MyFunctionalTest1Min

MyFunctionalTest1Typ

MyFunctionalTest2Max

MyFunctionalTest2Min

MyFunctionalTest2Typ

MyFunctionalTest2Max

#

#

#카운터들은 시험의 실행 동안 증분되는 변수들임.

#그들은 영(0)으로 초기화되는 무부호 정수임.

#

Counters {PassCount, FailCount}

#

#이제 흐름이 나타남. 흐름은, 본질적으로

#"흐름 가능(Flowables)"을 실행할 수 있으며,

#흐름 가능을 실행함으로써 리턴되는 결과에

#기초하여 다른 흐름 가능으로 전이할 수 있는

#유한 상태 머신을 나타내는 객체이다.

#흐름은 다른 흐름을 호출할 수 있다.

#

#

#하나의 흐름은 복수의 흐름 항목과 그들 사이의

#전이들로 구성된다. 흐름 항목은, 그를 에워싼

#흐름 내에서 고유한 명칭을 가지며, "흐름 가능"

#객체를 실행하며, 이어서 그를 에워싼 동일한

#흐름 내에서 다른 흐름 항목으로 전이한다.

#

#흐름 가능 객체들은 시험들 및 다른 흐름들을 포함한다.

#흐름 가능 객체가 실행될 때, 상기 흐름 항목이

#다른 흐름 항목으로 전이하기 위하여 상기 흐름 항목에

#의하여 사용되는 수치 결과를 리턴한다. 이 결과로서,

#시험 및 흐름 모두는 수치 결과 값을 리턴함으로써 종료된다.

#

#FlowTest1은 MyFunctionalTest1dml Min, Typ 및 Max 특색에 대한

#유한 상태 머신을 구현한다. 성공하면, 그것은 Test1Min,

#Test1Typ, Test1Max를 시험하고, 그의 호출자에게

#성공적인 결과로서 0을 리턴한다. 실패하면 실패의 결과로서

#1을 리턴한다.

#

#MyFunctionalTest1Min,... 등은 모두 0(통과),

#1 및 2(페일의 한 쌍의 레벨에 대하여)의 결과를

#리턴한다고 가정하자. FlowTest1에 의하여 구현되는

#유한 상태 머신의 전이 행렬은 다음과 같다. 즉,

#--

Result 0 Result 1 Result 2

#____

FlowTest1_Min FlowTest1_Typ return 1 return 1

FlowTest1_Typ FlowTest1_Max return 1 return 1

FlowTest1_Max return 0 return 1 return 1

#

#여기서, 각 흐름 항목에 의하여 실행되는 흐름 가능 인터페이스는

#다음과 같다. 즉,

흐름 항목(FlowItem) 실행되는 흐름 가능 인터페이스(IFlowable)

FlowTest1_Min MyFunctionalTest1Min

```
# FlowTest1_Typ MyFunctionalTest1Typ

# FlowTest1_Max MyFunctionalTest1Max

#

Flow FlowTest1

{

FlowItem FlowTest1_Min MyFunctionalTest1Min

{

Result 0

{

Property PassFail="Pass";

IncrementCounters PassCount;

GoTo FlowTest1_Typ;

}

Result 1,2

{

Property PassFail="Fail";

IncrementCounters FailCount;

Return 1;

}

}

FlowItem FlowTest1_Typ MyFunctionalTest1Typ

{

Result 0

{

Property PassFail="Pass";

IncrementCounters PassCount;

GoTo FlowTest1_Max;
```

```
}
Result 1,2
{
Property PassFail="Fail";
IncrementCounters FailCount;
Return 1;
}
#FlowTest1_Max에 대하여도 유사함.
FlowItem FlowTest1_Max MyFunctionalTest1_Max
{
Result 0
{
Property PassFail ="Pass";
IncrementCounters PassCount;
Return 0;
}
Result 1,2
{
Property PassFail="Fail";
IncrementCounters FailCount;
Return 1;
}
}
}
#
#FlowTest2는 FlowTest1과 유사하다. 그것은 MyFunctionalTest2의
#Min, Typ 및 Max 특색에 대한 유한 상태 머신을 구현한다.
```

#성공하면, 그것은 Test2Min, Test2Typ, Test2Max을 시험하고,

#이어서 그의 호출자에게 성공적인 결과로서 0을 리턴한다.

#실패하면, 실패의 결과로서 1을 리턴한다.

#MyFunctionalTest2Min,... 등은 모두 0(통과),

#1 및 2(페일의 한 쌍의 레벨에 대하여)의 결과를

#리턴한다고 가정하자. FlowTest2에 의하여 구현되는

#유한 상태 머신의 전이 행렬은 다음과 같다. 즉,

#--

Result 0 Result 1 Result 2

#_____

FlowTest2_Min FlowTest2_Typ return 1 return 1

FlowTest2_Typ FlowTest2_Max return 1 return 1

FlowTest2_Max return 0 return 1 return 1

#

#여기서, 각 흐름 항목에 의하여 실행되는 흐름 가능 인터페이스는

#다음과 같다. 즉,

흐름 항목(FlowItem) 실행되는 흐름 가능 인터페이스(IFlowable)

FlowTest2_Min MyFunctionalTest2Min

FlowTest2_Typ MyFunctionalTest2Typ

FlowTest2_Max MyFunctionalTest2Max

#

Flow FlowTest2

{

#...

}

#

이제, 주요 시험 흐름인 FlowMain이 나타난다.

#그것은, 이하와 같이, FlowTest1 및 FlowTest2를

#호출하는 유한 상태 머신을 구현한다.

#----------

Result 0 Result 1

#----------

FlowMain_1 FlowMain_2 return 1

FlowMain_2 return 0 return 1

#

#여기서 각 흐름 항목에 의하여 실행되는 흐름 가능 인터페이스는

#다음과 같다. 즉,

흐름 항목(FlowItem) 실행되는 흐름 가능 인터페이스(IFlowable)

FlowMain_1 FlowTestl

FlowMain_2 FlowTest2

Flow FlowMain

{

#첫번째로 선언된 흐름은 실행될 최초의 흐름(initial flow)이다.

#그것은 성공하면 FlowMain2로 진행하고, 실패하면 1을 리턴한다.

FlowItem FlowMain_1 FlowTest1

{

Result 0

{

Property PassFail="Pass";

IncrementCounters PassCount;

GoTo FlowMain2;

}

Result 1

{

```
#유감입니다. FlowTest1은 실패했습니다.
Property PassFail ="Fail";
IncrementCounters FailCount;
#우측의 소프트 빈에 추가함.
SetBin SoftBins."3GHzSBFTFail";
Return 1;
}
}
FlowItem FlowMain_2 FlowTest2
{
Result 0
{
#모두 통과!
Property PassFail="Pass";
IncrementCounters PassCount;
#우측의 소프트 빈에 추가함.
SetBin SoftBins."3GHzAllPass";
Return 0;
}
Result 1
{
#FlowTest1은 패스, 그러나, FlowTest2는 페일
Property PassFail="Fail";
IncrementCounters FailCount;
#우측의 소프트 빈에 추가함
SetBin SoftBins."3GHzCacheFail";
Return 1;
```

}

}

}

TestFlow=FlowMain;

상기 테스트 계획은 우선 순위에 따라 다음과 같이 구성된다.

1. 먼저, 버전 번호가 제공된다. 이 번호는 컴파일러의 버전과의 호환성을 보장하기 위하여 사용된다.

2. 이어서, 다수의 인입들이 선언된다. 이들은 상기 테스트 계획에 사용된 명칭들을 해석하기 위하여 필요한 선언들을 가진 다양한 파일들이다.

3. 다음으로, 상기 테스트 계획의 명칭이 선언되며, 그 다음으로 테스트 계획의 인라인 선언이 온다.

4. 다음으로, 한 세트의 *PListDefs*이 선언된다. 이들은, 명명된 파일들로부터의 GlobalPLists를 명명하는 파일 인가 명칭들을 포함한다. 그들은 또한 패턴 리스트 변수들을 포함한다. 패턴 리스트 변수들은 실행시에 주문형 흐름 가능들에서 초기화될 수 있는 변수들이다. 그들은 시험들을 실제의 패턴 리스트에 한정하는 것을 실행 시간까지 연기하는 수단을 제공한다.

5. 다음으로, 한 세트의 사용자 변수가 선언된다. 이들은 문자열을 포함한다.

6. 이어서, 통과되고 실패한 시험들의 개수를 판단하기 위한 특정의 카운터들이 선언된다. 카운터들은 단순히 영으로 초기화되고 카운터 증분(IncrementCounter) 문장에서 증분되는 변수들이다. 그들은, 피시험 디바이스의 시험의 종료시에 현재 설정된 빈만이 증분되는 의미를 갖는 상기한 빈(Bin)들과는 서로 다르다.

7. 다음으로, 일련의 테스트 조건이 선언된다. 이들의 각각은 테스트 조건 그룹과 선택자를 규정한다. 이 예에서, 상기 테스트 조건 그룹들은 mytestconditionsgroups.tcg로부터 유래한다. 그러나, 그들은 상기 테스트 계획 내에 인라인될 수도 있다.

8. 다음으로, 일련의 흐름 가능 또는 시험이 선언된다. 이 각각은, 패턴 리스트 및 테스트 조건을 선택하는 이미 알려진 기능적 시험이다. 그러므로, 예를 들어, MyFunctionalTest1Max는 테스트 조건 TC1Max 및 패턴 리스트를 선택한다.

9. 그 다음으로, 세 개의 흐름이 선언되는데, 그들은 FlowTest1, FlowTest2 및 FlowMain이다. 흐름(Flow)은 흐름 가능(Flowable)을 실행시킨다. 흐름 가능은 시험(MyFucntionalTest1Max와 같은) 및 다른 흐름들(FlowTest1 및 FlowTest2와 같은)을 포함한다. FlowTest1 및 FlowTest2의 각각은 Test1 및 Test2의 최소, 전형적 및 최대 버전 각각을 거쳐 실행된다. FlowMain 흐름은 이전에 선언된 흐름들, 즉 FlowTes1 및 이어서 FlowTest2를 호출한다.

10. 마지막으로, 상기 시험 흐름(TestFlow) 이벤트는 상기 FlowMain 흐름에 할당된다. 그러므로, 상기 FlowMain 흐름은, 사용자가 이 계획을 실행하도록 선택하였을 때 이 테스트 계획에 의하여 실행되는 것이다.

흐름을 위한 C++(C++ for Flows)

상기의 규칙을 가지고, 흐름들 그 자신들만을 제외하고는 대부분의 구성요소들에 대하여 C++ 구현이 가능하다.

흐름 항목을 위한 C++(C++ for FlowItem)

흐름 항목을 나타내는 C++ 클래스는 다음의 인터페이스를 가질 수 있다. 즉,

이 흐름 항목을 위하여 실행될 흐름 가능 인터페이스(IFlowable)를 설정하는 동작:

Status setFlowable(IFlowable* pIFlowable);

일단 흐름 항목이 이 흐름 가능 인터페이스(IFlowable)를 실행시키는 데에 필요한 호출의 세트로부터 복귀하면, 그것은 그 결과값에 따라 카운터의 리스트를 증분시킬 필요가 있을 것이다. 이 목적을 위하여, 상기 흐름 항목은 증분되어야 할 카운터들의 벡터를 가질 필요가 있다. 이것은 다음의 호출에 의하여 초기화된다. 즉,

Status setCounterRefs(unsigned int result,

CounterRefList counterRefs);

이것을 호출함으로써, 카운터에 대한 참조의 벡터가 흐름 항목 내에 셋업되고, 그리하여 그것은, 일단 상기 흐름 가능 인터페이스가 실행을 완료하면 그들을 증분시킨다. 예를 들어, 다음 문장

IncrementCounters A, B, C;

는 다음과 같이 상기한 호출을 바람직하게 이용할 것이다. 즉,

//어딘가에 먼저

CounterRefList counters;

...

//흐름 객체(flowObject)의

//결과 구절을 위한 코드

// Result 2,3 {...}

counters.reset();

counters.add(&A);

counters.add(&B);

counters.add(&C);

flowObject.setCounterRefs(2, counters);

flowObject.setCounterRefs(3, counters);

카운터라 명명된 임시적인 CounterRefList 객체가 사용된다. 먼저 counters.reset()이 호출되고, 이어서, 복수의 counters.add() 호출이 카운터의 리스트를 셋업하기 위하여 호출된다. 이어서, 이것은 카운터 어드레스의 벡터를 결과값 2 및 3에 대하여 갱신되도록 설정하는 데에 사용된다.

이어서, 흐름 항목이 특정 결과에 대하여 다른 흐름 항목으로 전이할 필요가 있다. 즉,

Status setTransition(unsigned int result, FlowItem* pFlowItem);

그러한 복수의 호출이, 소정이 결과 구문이 많은 결과값들을 다루는 경우에 당연히 작성될 필요가 있다.

상기 흐름 항목은 결과를 리턴할 필요가 있다. 이것은 다음에 의하여 수행된다. 즉,

Status setReturnResult(unsigned int result,

unsigned int returnResult);

예를 들어, 상기 예에서의 흐름 항목 FirstFlowItem에 대하여, "result"에 대하여 값 "2"로, 그리고 "returnResult"에 대하여 "1"로, 상기한 것들이 호출될 것이다.

마지막으로, 상기 흐름 항목은 실행할 동작이 필요하다. 즉,

Status execute(unsigned int& result, FlowItem* pNextFlowItem);

이 동작은 상기 실행 가능 항목을 실행할 것이고, 이어서 지시된 카운터를 갱신하며, 이어서 결과 또는 다음의 흐름 항목에 대한 포인터를 리턴할 것이다. 만약 이 포인터가 널(NULL)이라면, 그 결과는 리턴된 값이다.

흐름 항목 FlowMain_1에 대하여 생성된 코드는 다음과 같다. 즉,

FlowItem FlowMain_1;

FlowItem FlowMain_2;

CounterRefList counters;

FlowMain_l.setFlowable(FlowTestl);

//Result 0

counters.reset0;

counters.add(&PassCount);

FlowMain_1.setCounterRefs(0,counters);

FlowMain_1.setTransition(0,&FlowMain_2);

//Result 1

counters.reset();

counters.add(&FailCount);

FlowMain_1.setCounterRefs(l,counters);

//ITestPlan으로부터의 이하의 호출은

//현재의 빈 그룹과 빈 명칭을 설정할 것이다.

pTestPlan->setBin("SoftBins","3GHzSBFTFail");

FlowMain_1.setRetumResult(1, 1);

상기에서 생성된 코드는 흐름 가능 인터페이스인 "FlowTest1"을 실행시키기 위하여 FlowMain_1을 셋업하고, 이어서 각 결과에 대하여 적합한 카운터들의 리스트를 증분시키기 위하여 그것을 셋업하며, 마지막으로 필요한 액션을 취한다. 결과가 "0"인 경우의 필요한 액션은 FlowMain_1으로의 전이이며, 결과가 "1"인 경우에는 리턴이다.

C2. 테스트 계획에서의 카운터 지원(Counter Support in a Test Plan)

카운터들은 영으로 초기화되고 시험이 실행 중인 동안 다양한 시점에서 IncrementCounter 문장에 의하여 증분될 수 있는 변수들이다. 그들은 빈(Bin)들과는 서로 다른데, 빈들은 시험의 끝에서만 증분된다. 게다가, 빈들은 계층적이지만, 카운터들은 단순한 변수들이다. 그러므로, 카운터들은 빈에 비하면 훨씬 단순하고 보다 제한적인 설비들이다.

카운터들은, 무부호 정수인 명명된 카운터들의 세트를 유지하는 카운터 클래스의 멤버를 통해 테스트 계획에서 지원될 수 있다. 객체들은 카운터 선언을 통해 이 클래스에서 정의될 수 있다. 카운터들은 시험이 시작되면 자동적으로 리셋되지 못하므로, 테스트 계획으로 하여금 많은 피시험 디바이스를 거쳐 계수를 모으도록 할 수 있다. 카운터의 값의 리셋, 증분 및 질의하기 위한 메소드들이 제공된다. 이것은, 시험을 실행한 결과로서 계수를 결정하기 위하여 빈(bin)을 사용하는 것의 대안이 될 수 있다.

상기 테스트 계획은 바람직하게, 멤버 변수(member variable)인 m_modifiedCounters를 포함하는데, 그것은 피시험 디바이스에 시험을 실행함으로써 수정되는 카운터의 세트이다. 이 세트는 시험의 시작시에 공백 세트로 초기화된다. incrementCounter 호출이 만들어지는 곳마다, 상기 m_modifiedCounters에 명명된 카운터를 추가하는 코드가 생성될 것이다. 그러므로, 이 맴버는 피시험 디바이스에 시험이 실행되는 동안 수정된 모든 카운터를 함께 모은다.

흐름 객체에 대한 C++(C++ for the Flow Object)

일단 모든 흐름 항목들이 창출되면, 상기 흐름 객체가 이하에 개시된 바와 같이 창출될 수 있다. 즉,

흐름 항목을 추가하는 동작:

```
Status addFlowItem(FlowItem* pFlowItem, bool isInitialFlowItem);
```

이것은 지시된 흐름 항목을 상기 흐름에 추가한다. 만약 이것이 상기 흐름의 최초의 흐름 항목이라면, 상기 진리값(boolean)은 진실(True)로 설정된다.

흐름을 실행시키는 동작:

```
Status executeFlow(unsigned int& result);
```

이것은 상기 흐름이 복귀될 때 상기 흐름의 실행 결과와 함께 리턴된다. 이것은 액션은 최초의 흐름 항목과 함께 상기 흐름의 실행을 시작하는 것이다. 그것은 현재의 흐름 항목이 실행될 다음의 흐름 항목을 리턴하는 한 흐름 항목을 계속하여 실행시킨다. 현재의 흐름 항목이 결과를 리턴한 경우, 이 동작은 그 결과와 함께 완료된다.

따라서, 상기 흐름에 대하여 생성된 C++ 코드는, 상기 흐름에 흐름 항목을 추가시키기 위한 'addFlowItem()'에 대한 복수의 반복된 호출을 갖는다. 상기 'executeFlow()' 동작은 상기 테스트 계획에서의 이 흐름이 실행을 위해 선택되었을 때에 발생한다.

C3. 테스트 클래스들(Test Classes)

일반적으로 프로그램 코드의 대부분은 장치의 시험을 위한 데이터이며, 그 나머지는 테스트 프로그램의 코드인데, 그것은 시험의 방법론을 실현시킨다. 이 데이터는 피시험 디바이스에 의존적이다(예를 들어, 전력 공급 조건, 신호 전위 조건, 타이밍 조건 등). 상기 테스트 코드는 규정된 장치 조건을 ATE 하드웨어에 로드하기 위한 메소드들과 사용자가 지정한 객체들(데이터 로깅(datalogging) 등과 같은)을 실현하기 위하여 필요한 것들로 구성된다.

상기에서 예시된 바와 같이, 테스트 코드의 재사용 가능성을 향상시키기 위하여는, 그러한 코드들은 임의의 장치 특유적 데이터(예를 들어, 핀 명칭, 자극 데이터 등) 또는 장치-시험 특유적 데이터(예를 들어, DC 유닛, 측정 핀들, 목표 핀들의 개수, 패턴 파일의 명칭, 패턴 프로그램의 주소 등)로부터도 독립적이어야만 한다. 시험을 위한 코드가 이러한 형식의 데이터들과 함께 컴파일되면, 이러한 테스트 코드의 재사용 가능성은 감소된다. 그러므로, 어떠한 장치 특유적 데이터 또는 장치-시험 특유적 데이터도, 코드가 실행 중인 동안의 입력으로서, 상기 테스트 코드에 대하여 외부적으로 활용 가능하게 만들어져야 한다.

개방형 구조의 테스트 시스템에 있어서, 'ITest' 인터페이스의 구현인 테스트 클래스는 특정한 형식의 시험에 대하여 테스트 데이터와 코드의 구분을 (그리고, 따라서 코드의 재사용 가능성을) 실현시킨다. 그러한 테스트 클래스는 그의 별도의 인스턴스에 대한 "템플리트(template)"로 여겨지는데, 그것은 단지 장치 특유적 및/또는 장치-시험 특유적 데이터의 기초로 하여서만 서로 간에 다르다. 테스트 클래스들은 테스트 계획 파일에 규정된다. 각 테스트 클래스는 전형적으로 특정 형식의 장치 시험 또는 장치 시험을 위한 셋업을 구현한다. 예를 들어, 기능적, AC 및 DC 파라미터 시험들은 별도의 테스트 클래스들에 의하여 바람직하게 구현된다. 그러나, 주문형 테스트 클래스들이 테스트 계획에 사용될 수도 있다.

테스트 클래스들은 그 시험의 특정 인스턴스에 대한 선택 사항을 지정하는 데에 사용되는 파라미터들을 제공함으로써 상기 사용자로 하여금 클래스의 행동을 구성할 수 있게 한다. 예를 들어, 기능적 시험은, 실행될 패턴 리스트과 시험을 각각 규정하기 위한 'PList' 및 테스트 조건(TestCondition) 파라미터를 취할 것이다. 이들 파라미터에 대한 서로 다른 값들을 (테스트 계획 기술 파일의 서로 다른 "시험" 블록을 이용함으로써) 규정함으로써, 상기 사용자로 하여금 기능적 시험의 서로 다른 인스턴스를 창출할 수 있게 한다. 도 5는 하나의 테스트 클래스 504로부터 어떻게 서로 다른 테스트 인스턴스들 502가 도출되는지를 도시한다.

이들 클래스들은, 상기 컴파일러 400이 상기 테스트 계획 파일로부터 상기 시험의 기술(description of the test) 및 그들의 파라미터를 취하고 올바른 C++ 코드를 생성할 수 있도록 허용하도록 설계되어야 하는데, 상기 C++ 코드는 테스트 프로그램을 생성하도록 컴파일되고 링크될 수 있다. 상기 테스트 클래스 인스턴스들은, 장치 시험의 복잡한 실행 시퀀스를 창출하도록 시험 흐름을 기술하는 객체에 추가될 수 있다.

C4. 테스트 인터페이스(ITest) 및 흐름 가능 인터페이스(IFlowable)로부터의 유도(Derivation from ITest and IFlowable)

상기한 바와 같이, 테스트 클래스는 테스트 인터페이스(ITest)로부터 도출된다. 상기한 규칙들을 가지고, 이들은 상기 'ITest' 인터페이스를 구현하는 C++ 클래스들에서 구현될 수 있다. 'ITest' 인터페이스를 위하여 규정된 메소드들에 더하여, 이들 클래스들은, 장치 시험의 특정 클래스들을 수행하기 위하여 요구되는 시험 특유적 지능(Test-specific intelligence) 및 로직(logic)을 제공한다. 테스트 클래스들은 또한, 'IFlowable' 인터페이스를 구현한다. 이 결과로서, 테스트 클래스의 인스턴스들은 시험을 실행하는 흐름 항목에 사용될 수 있다.

개별화(Customization)

개별화 메커니즘은 사용자로 하여금 C 함수들을 호출할 수 있게 하며, 상기 'ITest' 및 'IFlowable' 인터페이스를 구현하는 그들 자신의 클래스들을 개발할 수 있게 한다.

프리헤더(Pre-Header)

TPL은 테스트 시스템의 테스트 클래스를 정의하는 포괄적인 메커니즘을 제공한다. 테스트 엔지니어는 추가적인 설비없이 테스트 시스템에 의해 사용되고 로딩될 수 있는 커스텀 테스트 클래스를 생성하기 위해 이러한 메커니즘을 사용한다. 테스트 엔지니어는 또한, 상기의 ITest 및 IFlowable 인터페이스를 구현하는 그들 자신의 테스트 클래스를 개발할 수 있다. 프리헤더는 C++ 언어에 자체 검사 능력을 제공함으로써 커스텀 테스트 클래스의 속성을 지원한다.

테스트 클래스의 한 객체를 그의 메소드와 서명(signature)에 관하여 조사할 수 있다면, 적합한 파라미터가 생성된 소스 코드에 삽입되도록 활용 가능하다는 것이 확인될 수 있을 것이다. 그러한 기능은 번역 단계(translation phase) 동안 오류 점검 및 유효화를 위하여 매우 유용하다. 만약 시험 엔지니어가 파라미터의 명칭 또는 이들 파라미터에 대한 인자(argument)의 개수(또는 어쩌면 형식)에 관하여 실수를 저질렀다면, 이러한 실수는 번역 단계에서 파악될 수 있고, C++ 컴파일러로부터의 컴파일시 오류 메시지를 기다리는 대신에, 번역시에 의미있는 오류 메시지가 제공될 수 있을 것이다.

자체 검사는, 객체에게 스스로의 내부를 들여다 보고 그의 속성(attribute)들 및 메소드들에 관한 정보를 리턴하도록 하는 능력을 가리킨다. 자바(Java)와 같은 일부 언어는, 그 언어의 일부로서 이 능력을 제공한다. 비주얼 베이직(Visual Basic)과 같은 다른 언어들은 함께 사용되도록 의도된 객체에 그러한 요건을 강요한다. C++은 이 특성에 대한 준비가 없다.

일 실시예에서, 프리헤더의 자체 검사 능력은 또한 선택적인 파라미터들의 표시뿐만 아니라 디폴트 파라미터 값을 제공하는 데에도 유용하다. 또한, 테스트 클래스들의 구현의 일부로서 이 능력이 제공된다면, 그래픽 사용자 인터페이스(graphical user interface:GUI) 응용 프로그램들은 이 정보를, 테스트 엔지니어들이 이들 클래스들을 효과적으로 사용하

도록 돕기 위한 대화(dialog) 및 다른 사용자 인터페이스 구성요소들을 동적으로 구축하는 데에 사용할 수 있다. 다른 실시예에서, 테스트 클래스 개발자는 (하나의 테스트 클래스마다) 하나의 텍스트 기반 소스 파일에서, 상기 개발자가 상기 클래스를 파라미터화하기 위하여 요구되는 것들로서 지정한 테스트 클래스의 대중적(public) 메소드/속성들을 특정할 수 있다.

하나의 소스가 선호된다는 점에 유의한다. 다시 말해, 하나의 파일에 테스트 클래스의 파라미터화 인터페이스의 기술을 가지며, 다른 독립된 (헤더) 파일에 C++ 인터페이스 기술을 가지며, 상기 두 가지 소스들을 계속하여 동기시킬 필요로 부담을 가질 것을 원하지는 않는다. 이 목적을 위하여, 상기 "텍스트 기반" 기술이 테스트 클래스를 위한 프리 헤더 파일에 내장되며, 그것은 컴파일러에 의하여 상기 테스트 클래스를 위한 C++ 헤더를 생성하기 위하여 뿐만 아니라 자체 검사를 목적으로 사용된다. 생성된 C++ 헤더 파일은 상기 테스트 클래스 C++ 코드를 최종적으로 컴파일하는 데에 사용되는 것이다.

C++에서의 헤더의 사용은 잘 알려져 있다. 그러나, C++ 언어는 분석하고 해석하기가 어렵기 때문에, 본 발명의 바람직한 실시예는, 테스트 프로그램을 생성하기 위해 C++ 컴파일러에 의해 사용되는 C++ 헤더 파일을 생성하기 위해, TPL 컴파일러에 의해 컴파일되는 프리헤더를 정의한다. 이 실시예에 의하면, 상기 테스트 클래스 개발자는, 프리 헤더 파일을 작성하여 대응되는 테스트 클래스 또는 다른 테스트 엔티티에 투명성을 허용한다.

이하의 예는 본 발명의 일 실시예에 의한 테스트 클래스를 위한 프리 헤더 파일의 개념을 개시한다. 시험 FunctTest1을 갖는 소스 파일로부터의 다음 발췌를 보자. 즉,

...

TestCondition TC1

{

TestConditionGroup=TCG1; #레벨들에 대하여 앞서 정의된 TCG

Selector=min;

}

TestCondition TC2

{

TestConditionGroup=TCG2; #타이밍에 대하여 앞서 정의된 TCG

Selector=min;

}

...

Test FunctionalTest FuncTestl

{

PListParam=patList1; #앞서 정의된 패턴 리스트

TestConditionParam=TC1;

TestConditionParam=TC2;

}

상기 C++ 컴파일러는, 상기 FuncTest1이 선언이 규정에 맞는지를 판단하기 위하여, 기능성 시험이 남기는 것을 알고 있을 필요가 있다. C++ 컴파일러로의 기능성 시험의 지식에 구축하는 대신에, 기능성 시험이 남기는 것의 정의를 상기 프리 헤더에서 규정할 수 있다.

기능성 시험이, 기초적 클래스들 Test1 및 Test2를 가지며, 파라미터 리스트('PList')인 멤버와 테스트 조건(TestConditions)의 배열을 가지는 C++ 클래스라 가정한다. 상기 C++ 컴파일러는, FuncTest1의 선언이 규정에 맞는지를 인지하기 위하여 상기 기능적 시험의 멤버들의 형식에 관하여 알고 있어야 한다.

게다가, 상기 FuncTest1에 대한 C++ 객체의 선언을 생성하기 위하여, 상기 기능성 테스트 클래스에 대한 C++ 헤더가 구축될 필요가 있다. 이것도 또한, 상기 C++ 컴파일러로 하여금 상기 기능성 테스트 클래스의 기초적 클래스, 그의 멤버들의 명칭 및 기타 그러한 정보를 알고 있을 것을 요구한다.

본 발명의 일 실시예에 의한 프리 헤더는 C++ 컴파일러에게 선언들의 규칙 적합성을 인지하고 선언에 대응되는 C++ 헤더와 객체 선언들을 생성하는 데에 모두 필요한 정보를 제공한다.

기능적 시험은 (파라미터화에 관한 한) 간단한 형식이 될 수 있고, 따라서 파라미터화를 위한 꽤 간단한 기술을 사용한다는 점을 주의하여야 한다. 따라서, 테스트 개발자는 다음과 같이 상기한 파라미터화를 지원하는 프리 헤더, 즉 FunctionalTest.ph를 작성할 수 있을 것이다(기초적 테스트 클래스들인 Test1 및 Test2에 대하여 프리 헤더가 활용될 수 있다고 가정한다.). 즉,

```
1 Version 1.0;
2 #
3 #FunctionalTest를 위한 프리 헤더의 파라미터화 규정
4 #
5 Import Testl.ph; #기초적 클래스 Test1에 대한 것
6 Import Test2.ph; #기초적 클래스 Test2에 대한 것
7 TestClass=FunctionalTest; #이 테스트 클래스의 명칭
8 PublicBases=Testl, Test2; #대중적 기초적 클래스들의 리스트
9 #파라미터 리스트 또는 "파라미터 블록":
10 Parameters
11 {
12 #이하의 선언은 기능적 시험이 다음을 가짐을 규정한다.
13 # -PList 형식의 파라미터
14 # -[C++ 형식 Tester::PattemTree에 의하여 나타난]
15 # -m_pPatList로 명명된 멤버에 저장되며
16 # -그것을 설정하기 위한 setPatternTree로 명명된 함수.
17 # -도구 팁으로서 사용하기 위한 GUI를 위한 파라미터 기술
```

```
18 PList PListParam
19 {
20 Cardinality=1;
21 Attribute=m_pPatList;
22 SetFunetion=setPatternTree;
23 Description="기능적 시험을 위한 PList 파라미터";
24 }
25 #
26 #이하의 선언은 기능적 시험이 다음을 가짐을 규정한다.
27 # -TestCondition 형식의 하나 또는 그 이상의 파라미터
28 # - [C++ 형식 Tester::TestCondition에 의하여 나타내어진]
29 # -mtestCondnsArray로 명명된 멤버에 저장된
30 # -그것을 설정할 addTestCondition이라고 명명된 함수.
31 # -도구 팀으로서 사용될 GUI에 대한 파라미터 기술
32 #[Implement] 구절은 이 함수의 디폴트 구현을 생성하기 위하여
33 #전이 단계를 야기한다.
34 #
35 TestCondition TestConditionParam
36 {
37 Cardinality=1-n;
38 Attribute=m_testCondnsArray;
39 SetFunction=addTestCondition[Implement];
40 Description="기능적 시험을 위한 테스트 조건 파라미터";
41 }
42 }
43 #
44 #이하의 섹션은 C++ 코드로의 탈출인 프리 헤더의 일부이다.
```

```
45 #이것은 "템플리트 블록(template block)"이라 불릴 것이다.
46 #
47 #이 세션의 모든 것들은, "$Class", "$Inc", "$ParamAryTypes",
48 #"$ParamAttrs", "$ParamFns" 및 "$ParamImpls"을 제외하고는
49 #생성된 헤더 파일에서 축어적으로 재생성된다.
50 #
51 #이하의 섹션에서는 '#' 문자로 시작되는 코멘트가 제공되지 않음을
52 #주의하여야 한다.
53 #
54 CPlusPlusBegin
55 $Inc
56 namespace
57 {
58 class $Class
59 {
60 //파라미터 스토리지를 위한 배열 형식:
61 $ParamAryTypes
62 public:
63 virtual void preExec();
64 virtual void exec();
65 virtual void postExec0;
66 $ParamFns
67 ...
68 private:
69 double m_someVar;
70 $ParamAttrs
71 ...
```

72 };

73 ...

74 **$ParamImpls**

75 }//명칭 공간 끝

76 CPlusPlusEnd

파라미터화된 테스트 클래스를 위한 C++(C++ For Parameterized Test Classes)

컴파일러가 프리 헤더 파일을 처리함에 따라, 그것은 **$Inc, $Class,** $ParamAryTypes 및 기타와 같은 컴파일러 변수들의 값들을 구축한다. 이렇게 함으로써, 축어적으로 상기의 C++ 코드를 생성함으로써 C++ 헤더를 생성할 수 있게 되며 상기 컴파일러 변수들 **$Inc, $Class,** 등의 값을 지시된 위치로 확장할 수 있다. 'FunctionalTest.ph'에 대하여, 상기 기능적 테스트 클래스에 대한 다음의 C++ 헤더 파일, 'FunctionalTest.h'를 생성한다.

1 #line 7 "./FunctionalTest.ph"

2 #include <ITest.h>

3 #line 5 "./FunctionalTest.ph"

4 #include <Test1.h>

5 #line 6 "./FunctionalTest.ph"

6 #include <Test2.h>

7 #line 5 5"./FunctionalTest.ph"

8 #include <vector>

9 #line 55 "./FunctionalTest.ph"

10 #include <Levels.h>

11 #line 55 "./FunctionalTest.ph"

12 #include <TestCondnGrp.h>

13 ...

14 #line 56 "./FunctionalTest.ph"

15 namespace

16 {

17 #line 7 "./FunctionalTest.ph"

18 class FunctionalTest: public ITest,

```
19 #line 8 "./FunctionalTest.ph"
20 public Test1,
21 #line 8 "./FunctionalTest.ph"
22 public Test2
23 #line 59 "./FunctionalTest.ph"
24 {
25 //파라미터 스토리지를 위한 배열:
26 #line 61 "./FunctionalTest.ph"
27 public:
28 #line 37 "./FunctionalTest.ph"
29 typedef std::vector<Tester::TestCondition *>
TestConditionPtrsAry_t;
30 #line 62 "./FunctionalTest.ph"
31 public:
32 virtual void preExec()
33 virtual void exec();
34 virtual void postExec();
35 public:
36 #line 7 "./FunctionalTest.ph"
37 void setName(OFCString &name); #모든 시험에 대하여 자동
38 #line 22 "./FunctionalTest.ph"
39 void setPatternTree(PatternTree *);
40 #line 23 "./FunctionalTest.ph"
41 String getPListParamDescription() const;
42 #line 39 "./FunctionalTest.ph"
43 void addTestCondition(TestCondition *);
44 #line 40 "./FunctionalTest.ph"
```

```
45 void getTestConditionParamDescription () const;
46 #line 67 "./FunctionalTest.ph"
47 ...
48 private:
49 double m_someVar;
50 #line 70 "./FunctionalTest.ph"
51 private:
52 #line 7 "./FunctionalTest.ph"
53 OFCString m_name; # 모든 시험에 대하여 자동
54 #line 21 "./FunctionalTest.ph"
55 Tester::PatternTree *m_pPatList;
56 #line 38 "./FunctionalTest.ph"
57 TestConditionPtrsAry_t_m_testCondnsArray;
58 #line 71 "./FunctionalTest.ph"
59 ...
60 };
61 ...
62 #line 7 "./FunctionalTest.ph"
63 inline void
64 #line 7 "./FunctionalTest.ph"
65 FunetionalTest::setSame(OFC$tring &name)
66 #line 74 "./FunctionalTest.h"
67 {
68 m_name=name;
69 return;
70 }
71 #line 39 "./FunctionalTest.ph"
```

```
72 inline void
73 #line 39 "./FunctionalTest.ph"
74 FunctionalTest::addTestCondition(TestCondition *arg)
75 #line 74 "./FunctionalTest.ph"
76 {
77 m_testCondnsArray.push_back(arg);
78 return;
79 }
80 #line 23 "./FunctionalTest.ph"
81 inline void
82 Tester::String FunctionalTest::getPListParamDescription()
83 {
84 return "기능적 시험에 대한 PList 파라미터";
85 }
86 #line 40 "./FunctionalTest.ph"
87 inline void
88 Tester::String FunctionalTest::getTestConditionParamDescription()
89 {
90 return "기능적 시험에 대한 테스트 조건 파라미터";
91 }
92 #line 75 "./FunctionalTest.ph"
93 }//End namespace
```

상기한 바와 같이, 이 프리 헤더 파일은, 컴파일러로 하여금, FunctionalTest 테스트 클래스에 대한 선언의 유효성을 점검하고, 그에 대한 소스 코드를 생성하며, 테스트 클래스를 위한 C++ 헤더 파일을 생성하게 한다.

예를 들어, 편의를 위하여 이하에 다시 기재한 상기의 기능적 시험 선언을 생각해 보자. 즉

```
Test FunctionalTest FuncTest1
{
```

PListParam=patList1; #앞서 정의된 패턴 리스트

TestConditionParam=TC1;

TestConditionParam=TC2;

}

상기 컴파일러는, C++ 헤더 파일의 선언에 따라 상기 FunctionalTest 구축을 위해 다음의 소스 코드를 생성한다:

FunctionalTestFuncTest1;

FuncTest1.setName("FuncTest1");

FuncTest1.setPatternTree(&patList1);

FuncTest1.addTestCondition(&TC1);

FuncTest1.addTestCondition(&TC2);

상기 기술 함수에 대하여 생성된 명칭에도 또한 주의하여야 한다. Xxx로 명명된 각 파라미터는 다음의 멤버 함수와 관련된다. 즉,

Status getXxxDescription() const;

이것은 GUI가 사용할 수 있는 도구 팁(tool tip)에 대한 기술을 가진 문자열을 리턴한다.

다른 프리 헤더 특성(Other Pre-Header Features)

상기 프리 헤더는, 추가적인 형식으로서, 몇몇 다른 사용자 정의 에뉴머레이션(enumeration)을 지원한다. 이것은, GUI로 하여금, 특정 파라미터의 값을 설정하는 데에 사용될 수도 있는 가능한 선택들의 드롭 다운(drop down) 리스트를 제공하게 한다. 게다가, 상기 프리 헤더는 테이블로서 생각될 수 있는 다수의 파라미터들을 관련시키는 특성을 제공한다. 예를 들어, 명칭에 대한 문자열의 배열 및 값들에 대한 정수들의 배열의 관련된 세트로서 "속성들(properties)"의 배열을 구현하는 것이 편리할 수 있다. 일 실시예에서, 이 특성은 커스텀 형식의 배열을 사용하는 것에 의해 구현된다. 이러한 특성은 이하의 예에서 개시하였다:

#--

File FooBarTest.ph

#

주문형 테스트 클래스 FoobarTest에 대한

파라미터화 규정 프리 헤더(Parameterization specification pre-header)

#--

Version 1. 0;

Import Testl.ph; #기초 클래스 Test1에 대한 것.

TestClass=FoobarTest; #이 테스트 클래스의 명칭

PublicBases=Test1; #대중적 기초 클래스들의 리스트

#파라미터들의 리스트:

Parameters

{

#에뉴머레이션 형식

Enum FBEnum=FB1, FB2, FB3;

#FBEnum 파라미터를 정의함.

FBEnum FBE

{

Cardinality=1;

Attribute=m_fb;

SetFunction=setFb;

Description="Foobar 시험을 위한 FBE 파라미터";

}

#이 클래스는 상기 클래스 내에서 해석되는

#명칭-번호 쌍의 배열을 갖는다.

ParamGroup

{

Cardinality=0-n;

#이 배열에서의 명칭 필드는:

-문자열 형식이며,

-[C++ 형식의 Tester::String에 의하여 표현되며]

-m_NameArray로 명명된 멤버에 저장되며,

-그것을 설정하는 함수는 addName으로 명명된다.

-도구 팁으로서 사용하기 위한 GUI에 대한 파라미터 기술

String Name

{

Attribute=m_NameArray;

SetFunction=addName;

Description="값을 가진 명칭";

}

#이 배열에서의 번호 필드는:

-정수 형식을 가지며,

-[C++ 형식인 int로 표현되며]

-m_NumberArray로 명명된 멤버에 저장되며,

-그것을 설정하기 위한 함수는 addNumber로 명명됨.

-도구 팁으로서 사용하기 위한 GUI에 대한 파라미터 기술

Integer Number

{

Attribute=m_NumberArray;

SetFunction=addNumber;

Description="명칭의 값";

}

}

#이하의 선언은, 기능적 시험이 다음을 가지고 있음을 규정한다. 즉,

-PList 형식의 파라미터

-[C++ 형식 Tester::PatternTree에 의하여 표현되며]

-m_pPatList로 명명된 멤버에 저장되며,

-그것을 설정하기 위한 함수는 setPattemTree로 명명됨.

-도구 팁으로서 사용하기 위한 GUI에 대한 파라미터 기술

PList PListParam

{

Cardinality=1;

Attribute=m_pPatList;

SetFunction=setPatternTree;

Description="기능적 시험을 위한 PList 파라미터";

}

#

#이하의 선언은 기능적 시험이 다음을 가지고 있음을 규정한다. 즉,

-하나 또는 그 이상의 TestCondition 형식

-[C++ 형식 Tester::TestCondition에 의하여 표현되며]

-m_testCondnsArray로 명명된 멤버에 저장되며,

-그것을 설정하기 위한 함수는 addTestCondition로 명명됨.

#[implement] 구절은 이 함수의 디폴트 구현을 생성하기

#위한 번역 단계를 야기한다.

#

TestCondition TestConditionParam

{

Cardinality=1-n;

Attribute=m_testCondnsArray;

SetFunction=addTestCondition[Implement];

Description="기능적 시험을 위한 테스트 조건 파라미터";

}

}

CPlusPlusBegin

$Inc

namespace

{

class $Class

{

//파라미터 스토리지에 대한 어레이 형식:

$ParamAryTypes

public:

virtual void preExec();

virtual void exec();

virtual void postExec();

$ParamFns

//...

private:

double m_someVar;

$ParamAttrs

//...

};

//...

$ParamImpls

} // End namespace

CPlusPlusEnd

커스텀 형식 명칭-번호 쌍이 선언될 수 있음에 유의한다. 그 커스텀 형식의 배열 파라미터는 상기 파라미터의 ParamGroup과 동일한 효과를 성취하도록 사용될 수 있다. 상기에서 제시된 테크닉은 커스텀 형식을 선언할 필요성을 회피하기 위한 방법이다.

도 13은 본 발명의 일 실시예에 따라 테스트 프로그램을 개발하는 방법을 도시한다. 상기 방법은 테스트 계획 파일 1302를 테스트 프로그램 언어(TPL)로 기술하고, 상기 테스트 계획 파일은 상기 테스트 프로그램의 하나 이상의 테스트를 기술한다. 상기 방법은 또한, 시스템 프로그램 언어(SPL)로 테스트 클래스 파일 1306을 기술하고 상기 TPL로 상기 테스트 클래스 파일의 대응되는 프리헤더(pre-header) 파일 1304를 기술하며, 상기 테스트 클래스 파일은 상기 테스트 프로그램의 하나 이상의 테스트의 구현을 기술한다. 이러한 구현의 예에서, 상기 SPL은 C/C++ 프로그램 언어이다. 관련 기술 분야에서 숙달된 자는 C++ 프로그램 언어 대신에 다른 시스템 프로그램 언어가 사용될 수 있음을 인식할 것이다. 테스트 계획 파일을 TPL로 기술하는 이점들 중의 하나는 문제 도메인에 더 가까와진다는 것이라는 점에 유의한다. TPL은 시험 계획 개발자로부터 시스템 프로그래밍의 부담과 복잡성을 분리한다.

본 발명의 일 실시예에 따라, 프리헤더 파일은 다음의 선언을 갖는다:

- 다른 TPL 파일에서와 같은, 선택적 인입.

- 테스트 클래스의 명칭.

- 이 클래스의 구현을 위한 DLL을 특정하는 선택적인 TestClass DLL 키워드.

- 공개(public) 기반의 선택적 리스트.

- 이 테스트 클래스의 파라미터를 특정하는 파라미터 섹션.

- 상기 테스트 클래스를 위한 인라인된(in-lined) 템플리트를 제공하는 템플리트 섹션.

프리헤더 파일의 파라미터 섹션은 테스트 클래스 파일 1306을 위한 파라미터를 특정하기 위해 테스트 클래스 개발자에 의해 사용되고, 이렇게 해서 사용자 정의된 테스트를 생성한다. 이들은 사용자가 인스턴스로서 특정할 수 있는 테스트 클래스 파일의 멤버이고, 이렇게 해서 사용자에게 시험 동작의 더 정밀한 제어를 제공한다. 예를 들어, 테스트 클래스 개발자는, 시험이 시행될 때 '장황한(verbose)' 또는 '간결한(terse)'과 같은 추적 레벨을 제어하기 위해 사용자에 의해 정의될 수 있는, 사용자 정의된 파라미터 TraceLevel을 선언할 수 있다.

테스트 클래스 파일의 파라미터 섹션 내에서 선언될 수 있는 세 가지 형식의 엔티티가 있다:

- Enum 키워드를 사용하는 에뉴머레이션(enumeration). Enum 키워드는 테스트 클래스 에뉴머레이트된(enumerated) 형식을 정의하기 위해 사용된다는 점에 유의한다. 테스트 클래스 파일은 이러한 에뉴머레이트된 형식의 파라미터를 가질 수 있다. 이는 또한 파라미터에 허용된 값들의 드롭다운(drop-down) 리스트를 구축하기 위해 GUI에 의해 사용될 수 있다.

- 파라미터 형식(여기서, 형식은 유효한 TPL 형식)을 특정하는 Integer, PList, 등과 같은 키워드를 사용하는 파라미터.

- ParamGroup 키워드를 사용하는 파라미터의 그룹. ParamGrup 키워드는, C++ 클래스와 유사하게, 다수의 서로 다른 파라미터를 그룹으로 수집하기 위해 사용된다.

테스트 클래스 파일의 개별 파라미터는 파라미터 섹션 내에서 선언될 수 있다. 파라미터 선언은 상기 파라미터의 형식, 그 명칭, 및 다양한 속성을 선언하는 구문을 갖는다. 다음은 한 세트의 파라미터의 일 예이다.

- 카디낼리티(Cardinality): 이것은 지원되는 이러한 형식의 파라미터의 수를 지시한다.

- 속성(Attribute): 이러한 형식의 파라미터 값(들)의 저장으로서 사용하기 위한 C++ 변수의 명칭.

- SetFunction: 이러한 파라미터를 위한 값의 설정을 위해 사용하는 함수의 속성. 상기 함수를 따르는 선택적 키워드 "[implement]"는 ($ParamFunctionImplementations 에 의해 지시된 포인트에서 삽입된) 상기 클래스 헤더에서 인라인 메소드로서 사용가능하게 된다. 그렇지 않으면, 사용자는 상기 함수의 구현을 제공할 책임이 있다.

- 기술(Description): 문자열 리터럴(string literal)은 실행시간 중에 파라미터의 변경을 돕기 위해 GUI에 의한 도구 팁(tool tip)으로서 사용될 수 있다. *Foo*라는 명칭의 파라미터를 위한 커스텀 클래스에서 생성된 C++ 멤버 함수는 "*OFC: StringgetFooDescription() const*"이다. 이 함수는 특정의 문자열을 리턴한다.

- GUIType: 이 선택적인 키워드는, 이 테스트 클래스의 인스턴스를 위해 XML 기술이 생성된 때, GUI로 통과될 문자열을 특정한다. 그것은 GUI가 이 파라미터 값을 특별한 방법으로 디스플레이하도록 한다.

프리헤더의 템플리트 섹션은 헤더 파일 1312로 삽입될 소스 코드를 특정한다. 그것은 테스트 클래스 개발자에게 이 테스트 클래스 파일을 위해 생성된 C++ 헤더에 대한 직접적인 제어를 준다. 상기 템플리트 섹션은 키워드 CPlusPlusBegin과 CPlusPlusEnd 사이에 배치된다. 그것은, 다양한 선언을 위한 배치를 특정하는 ($ImportDeclarations, $ClassName, $ParamArrayTypes, $ParamFunctionDeclarations, $ParamAttributes, $ParamFunctionImplementations, 등과 같은) 다양한 대체 지시자(indicator)를 사용한다.

상기 방법은 테스트 계획 파일 1302와 프리헤더 파일 1304를 컴파일하기 위해 TPL 컴파일러 1308를 적용한다. 상기 TPL 컴파일러는 C++ 언어로 기술된 유도된 시험 계획 파일 1310을 형성하기 위해 테스트 계획 파일 1302를 컴파일한다. 상기 TPL 컴파일러는 테스트 계획 파일을 컴파일하는 동안 적어도 두 개의 중요 기능을 수행한다. 첫째, 그것은 테스트 계획 파일을 TPL 기술로부터 SPL 기술로 번역한다. 둘째, 그것은 프리헤더 파일을 사용하여 테스트 계획 파일을 유효화한다. 구체적으로, 그것은 테스트 계획 파일의 의미론적(semantic) 오류를 검사한다. 또한, 그것은 테스트 엔티티의 속

성을 확인한다. 그것은 상기 테스트 엔티티의 속성이 테스트 계획 파일 또는 프리헤더 파일에 선언되고 존재한다는 것을 보장한다. 또한, 그것은 함수 호출과 그에 대응하는 테스트 계획의 파라미터를 확인한다. 그것은 테스트 계획 파일의 함수 호출이 테스트 클래스 구현에 존재하고 함수 호출의 파라미터가 정의된다는 것을 보장한다.

유사하게, TPL 컴파일러 1308은 C++ 언어로 기술된 헤더 파일 1312를 형성하기 위해 프리헤더 파일 1304를 컴파일한다. TPL 컴파일러 1308은 헤더 파일 1312에 프리헤더 파일 1304의 파라미터 섹션에 기술된 파라미터를 삽입한다. 또한, TPL 컴파일러 1308은 헤더 파일 1312에 프리헤더 파일 1304의 템플리트 섹션에 참조된 소스 코드를 삽입한다. 헤더 파일 1312는 두 개의 용도를 갖는다. 첫째, 테스트 클래스 객체 파일 1316을 생성하기 위해 SPL 컴파일러 1314에 의해 사용된다. 둘째, 테스트 계획 객체 파일 1318을 생성하기 위해 SPL 컴파일러 1314에 의해 참조된다.

마지막으로, 테스트 클래스 객체 파일 1316과 테스트 계획 객체 파일은, 테스트 프로그램 1322의 실행가능한 테스트 계획 DLL 파일(TPL.DLL)과 실행가능한 테스트 클래스 DLL 파일(TCL.DLL)을 생성하기 위해 링커(linker) 1320에 의해 링크된다.

C5. 커스텀 함수 선언들(Custom Function Declarations)

이것은, 흐름 전이가 발생할 때, 사용자가 커스텀 함수를 호출할 수 있게 한다. 커스텀 함수는 이하와 같이 프리 헤더를 통하여 선언된다. 즉,

```
#-------------------------------------------------------------

# File MyFunctions.ph

#

#MyFunctions에 대한 파라미터화 규정 프리 헤더

#-------------------------------------------------------------

Version 1.0;

Functions=MyFunctions; #이 함수들의 그룹의 명칭

#두 값의 최소한을 결정하기 위하여 MyFunctions 명칭 공간에서

#이하의 C++ 함수를 선언함.

# //x, y의 최소한을 리턴함.

# double MyRoutines::Min

# (ITestPlan* pITestPlan, int& x, int& y);

Integer Min(Integer x, Integer y);

#UserRoutines 명칭 공간에서

#배열의 평균을 리턴하기 위하여

#이하의 C++ 함수를 선언함.

# //배열의 평균을 리턴함

# double MyRoutines::Avg
```

(ITestPlan* pITestPlan, double* a, const int a_size);

#상기 C++ 함수가 a 및 a'Length로 호출됨.

Double Avg(Double a[]);

#피시험 디바이스 식별자와 메시지를 인쇄하기 위하여

#UserRoutines 명칭 공간(namespace)에 이하의

#C++ 함수를 선언함.

//배열의 평균을 리턴함.

double MyRoutines::Print

(ITestPlan* pITestPlan, String* msg, unsigned int& dutId);

#상기 C++ 함수는 a 및 a'Length로 호출된다.

Void Print(String msg, UnsignedInteger dutId);

컴파일러가 상기한 선언들을 표준적인 방식으로 확장할 것이므로, C++ 섹션은 전형적으로 상기한 선언들을 제공받아야 한다. 물론, 사용자는 이들 함수들의 C++ 구현에 책임을 진다. 상기 함수들의 모두가 묵시적 첫번째 파라미터로서 'ITestPlan' 포인터를 가정적으로 취한다는 점을 주의하여야 한다. 이 포인터는, 상기 함수 작성자에게 테스트 계획의 '스테이트S(stateS)'로의 억세스를 제공한다. 예를 들어, 상기 함수 작성자는 상기 현재의 흐름, 상기 흐름의 현재의 흐름 항목, 현재의 결과 구절, 사용자 변수의 값들 및 기타 그러한 정보들로 억세스하기 위한 ITestPlan 인터페이스를 사용할 수 있다. 특정의 테스터에 의하여 정의된 함수들이 Functions.ph 파일에서 사용되도록 활용가능하다. 즉,

Version 1.2.3;

#

File Functions.ph

#

Functions=Functions; #이 함수들의 그룹의 명칭

#Functions 명칭 공간에서 다음의 C++ 함수를 선언함.

#호출자에 의하여 현재 시험되고 있는

#피시험 디바이스의 식별자를 리턴함.

UnsignedInteger GetDUTID();

커스텀 함수 선언을 위한 C++(C++ for Custom Function Declarations)

상기 MyFunctions에 대하여 컴파일러에 의하여 생성될 C++ 코드는, 상기 MyFunctions 명칭 공간에서 몇개의 함수들만을 선언하는 것이다. 즉,

namespace MyFunctions

{

double Min(ITestPlan* pITestPlan, int& x, int& y);

double Avg(ITestPlan* pITestPlan, double* a, const int a_size);

void Print(ITestPlan* pITestPlan, char* Msg, unsigned int dutID);

}

이들 함수들은 흐름으로부터 호출될 수 있다.

C6. 커스텀 흐름 가능(Custom Flowables)

상기 프리 헤더를 사용하여 C++의 'IFlowable' 인터페이스를 구현하는 프리 헤더를 창출할 수도 있다. 이것은, 사용자가 흐름 항목에서 실행될 수 있는 커스텀 흐름 가능들을 정의할 수 있게 한다. 이하에 개시된 내용은, 사용자 정의 흐름 가능인 MyFlowable에 대한 프리 헤더이다. 즉,

#---

File MyFlowable.ph

MyFlowable에 대한 파라미터화 규정 프리 헤더

#---

Version 1.2.4;

FlowableClass=MyFlowable; #이 커스텀 클래스의 명칭

#파라미터 리스트:

Parameters

{

#이하의 선언은 MyFlowable이 다음을 가짐을 규정한다. 즉,

-하나의 선택적인 정수형 파라미터 Int1

-[C++ 형식인 int에 의하여 표현되고]

-m_int1Val이라 명명된 멤버에 저장되며,

-그것을 설정할 함수는 setInt1Val이라 명명됨.

Integer Int1

{

Cardinality=0-1;

Attribute=m_int1Val;

SetFunction=setInt1Val;

}

#이하의 선언은 MyFlowable이 다음을 가짐을 규정한다. 즉,

-하나의 정수형의 의무적 파라미터 Int2

-[C++ 형식인 int로 표현되며]

-m_int2Val로 명명된 멤버에 저장되며

-그것을 설정할 함수는 setInt2Val로 명명됨.

Integer Int2

{

Cardinality=1,

Attribute=m_int2Val;

SetFunetion=setInt2Val;

}

#이하의 선언은 MyFlowable이 다음을 가짐을 규정한다. 즉,

-문자열 형식의 하나 또는 그 이상의 파라미터

-[C++ 형식 Tester::String에 의하여 표현되며]

-m_stringArrVal로 명명된 멤버에 저장되며,

-그것을 설정할 함수는 addStringVal로 명명됨.

String StringItem

{

Cardinality=1-n;

Attribute=m_stringArrVal;

SetFunction=addStringVal;

}

#이하의 선언은 MyFlowable이 다음을 가짐을 규정함. 즉,

-하나의 PList 파라미터

-[C++ 형식 Tester::PList에 의하여 표현되며]

-m_plist로 명명된 멤버에 저장되며,

-그것을 설정할 함수는 setPListParam로 명명됨.

PList PListParam

{

Cardinality=1;

Attribute=m_plist;

SetFunction=setPListParam;

}

}

#

#이하의 섹션은 C++ 코드로의 탈출인 프리 헤더의 일부이다.

#

#"$Class", "$Ihc", "$ParamAryTypes", "$ParamAttrs", "$ParamFns" 및

#"$ParamImpls"를 제외하고,

#이 섹션의 모든 것들은 생성된 헤더 파일에서 축어적으로 재생성될 것이다.

#

#문자 '#'로 시작되는 코멘트가 이하의 섹션에서는

#지원되지 않는다는 점을 주의하여야 한다.

#

CPlusPlusBegin

$Inc

namespace

{

class $Class

{

//파라미터 스토리지에 대한 어레이 형식:

$ParamAryTypes

public:

virtual void preExec();

virtual void exec();

virtual void postExec();

$ParamFns

//...

private:

double m_someVar;

$ParamAttrs

//...

};

//...

$ParamImgls

}//End namespace

CPlusPlusEnd

'IFlowable' 인터페이스를 구현하는 몇몇 클래스들이 존재한다. 이들은 다음을 포함한다. 즉,

1. 테스트 계획이 현재의 테스터 구성 내에서 실행될 수 있는지의 여부를 점검하는 프로그램 로딩을 위한 흐름.

2. 특정 패턴 및 패턴 리스트를 로드하는 패턴 로딩을 위한 흐름.

3. 하드웨어 및 소프트웨어를 이미 알려진 상태로 놓으며, 전역 변수들을 로드하며, 그리고 기타 초기화 및 유효화 기능을 수행하는 초기화를 위한 흐름.

4. 기타 일반적으로 유용한 시험 흐름들.

C7. 커스텀 형식들(Custom Types)

상기한 테스트 클래스 파라미터화에 관한 논의에서는, 테스트 클래스 파라미터들이, 'PLists' 및 'TestConditions'와 같은 이미 알려진 형식들, 즉 기초적 형식들 및 테스터가 정의한 형식들만이 허용되었다. 사용자를 위한 유연성을 위하여, 형식 확장성(type extensibility)이 제공되는 것이 중요하며, 그리하여 (컴파일러에게는 선험적으로 알려지지 않은) 형식들도 생성되고 사용될 수 있다. 커스텀 형식(custom types;CTs)이 *Custom Types*에서 정의될 수 있다. 이들은, 함수 서명(function signature)을 위한 C-언어의 'typedefs'에 대응되는 형식들에 대하여 뿐만 아니라, C-언어 구조에 대응되는 형식들(C++에서 그들과 명칭이 같은 것들과 상당히 다른 평범한 오래된 데이터(Plain Old Data; POD)로도 알려져 있는) 형식을 정의하는 데에 사용될 수 있다. 사용자 형식을 가진 별도의 파일은 .ctyp라는 확장자를 가질 것이다. 이하에 본 발명의 바람직한 실시예에 의한 사용자 형식 선언을 개시한다. 즉,

#----------

```
# File MyCustomTypes.ctyp

#-------------------------------------------------------------

Version 1.0.0;

CustomTypes

{

#구조화된 Plain-Old-Data 형식

Pod Foo

{

String S1; #문자열은 표준 형식임.

Integer I1; #정수도 같음.

String S2;

}

#'Foo'를 사용하는 또 다른 구조화된 형식

Pod Bar

{

Foo Fool;

String S1;

Foo Foo2;

}

#

#함수로의 포인터

# 리턴 형식: 정수

# 파라미터들: 정수, 정수

#

Routine BinaryOp(Integer, Integer) Returns Integer;

#

#함수로의 또 다른 포인터
```

리턴 형식: Void

파라미터: 정수

#

Routine UnaryOp(Integer) Returns Void;

#

#파라미터를 취하지 않으며 값을 리턴하지 않는 함수로의 포인터

#

Routine NullaryOp() Returns Void;

}

커스텀 형식을 위한 C++(C++ for Custom Types)

상기에 제시한 커스텀 형식 선언은 컴파일러에 의하여 이하의 C++ 코드로 번역될 것이다. 즉,

```
namespace CustomTypes
{
struct Foo
{
Tester::String S1;
int I1;
Tester::String S2
};
struct Bar
{
Foo Foo1;
Tester::String S1;
Foo Foo2;
};
typedef int(*BinaryOp)(int&, int&);
typedef void(*UnaryOp)(int);
```

typedef void(*NullaryOp)();

}

이러한 형식들의 객체들은 파라미터로서 다음에 개시된 바와 같이 테스트 클래스들에게 전달된다.

테스트 클래스 파라미터로서 커스텀 형식의 사용(Using Custom Types as Test Class Parameters)

사용자가, 커스텀 형식들을 수록한 파일(즉, .ctyp 파일) 내에서 정의된 임의의(즉, 사용자가 정의한) 객체들 뿐만 아니라, (패턴 리스트 및 테스트 조건에 더하여) 다른 클래스 객체들로 초기화될 필요가 있는 시험으로의 확장을 가지는 경우를 생각해 보자. 예를 들어, 사용자가 상기 파일 MyTestCTs.ctyp에 정의된 CTs를 이용하고 싶어한다고 하자.

#File MyTesetCTs.ctyp

Version 1.0;

CustomTypes

{

Pod Foo

{

String name;

PList patternList;

}

Pod Bar

{

Foo someFoo;

Double dVal;

}

Routine Binary0p(Integer, Integer) return Integer;

}

상기 형식들을 이용하기 위하여 사용자가 하여야 할 모든 것들은, 그의 테스트 클래스 프리 헤더에 상기 파일을 인입하는 것이다. 상기 컴파일러가 그렇게 정의된 CTs를 해석하므로, 그것이 상기 테스트 클래스 프리 헤더를 처리할 때에 'Foo' 및 'Bar'에 대한 정의들은 그것에 활용 가능하게 된다. 게다가, 상기 컴파일러는, 그 정의가 상기 파일 myTestCTs.h에 위치하는 상기의 'Foo' 및 'Bar' 형식들에 각각 대응하는 두 개의 C-언어 구조, 즉 'struct Foo' 및 'struct Bar'를 정의한다. myTestCTs.ctt에 대한 상기 인입 문장은 생성된 테스트 클래스 C++ 헤더에 상기 파일 myTestCTs.h가 포함되도록 한다. 이하의 예는 이 프로세스를 예시한다. 먼저, 상기 테스트 계획에서의 시험에 대한 선언을 생각해 보자(패턴 리스트과 테스트 조건에 대한 선언은 명확성을 위하여 생략되었다.). 즉,

...

```
Import MyFunctions.ph;

Import MyCustomTypes.ctyp;

...

#상기 'CustomVars' 블록은 앞서 정의된

#커스텀 형식의 변수들을 정의한다.

CustomVars

{

...

Bar bar1=

{

{"이것은 'Foo'이다.", somePatList}, #someFoo

3. 14159 #dVal

}

#

#이진 연산자인 함수.

#할당의 오른쪽의 명칭은 MyFunctions에서

#선언된 루틴이며, 물론 그에 대하여

#사용자는 구현을 제공하여야 한다.

#

BinaryOp bop1=MyFunctions.Min;

}

...

Test MyFancyTest MyTest1

{

...

BarParam=bar1;

BinaryOpParam=bop1;
```

}

...

상기 예에서, 'CustomVars' 블록이 테스트 계획에 포함된다. 별도의 개별화된 변수를 가진 파일이 .cvar 확장자를 가질 것이다. 사용자는 다음과 같이, 상기 파라미터화를 지원하는 MyFancyTest에 대한 헤더를 작성한다(패턴 리스트 및 테스트 조건에 대한 파라미터화 선언은 명확성을 위하여 생략되었다.). 즉,

#---

File MyFancyTest.ph

#

MyFancyTest에 대한 파라미터화 규정 프리 헤더

#---

Version 1.0.2;

Import MyCustomTypes.ctyp; #MyFancyTest에 사용된 CTs에 대한 것

Import FunctionalTest.ph; #기본적 클래스 'FunctionalTest'에 대한 것

TestClass=MyFancyTest; #이 테스트 클래스의 명칭

PublicBases=FunctionalTest; #대중적(public) 기본 클래스들의 리스트

#파라미터의 리스트:

Parameters

{

#이하의 선언은 MyFancyTest가 다음을 가짐을 규정한다. 즉,

-커스텀 형식 Bar의 파라미터의 선택적 배열

-[C++ 형식 CustomTypes::Bar에 의하여 표현되며]

-m_barsArray로 명명된 멤버에 저장되며,

-그것을 설정하는 함수는 addBarParam로 명명됨.

#'addBarParam'에 대한 구현이 생성됨.

Bar BarParam

{

Cardinality=0-n;

Attribute=m_barsArray;

SetFunction=addBarParam[Implement];

}

#이하의 선언은 MyFancyTest가 다음을 가짐을 규정한다.

-커스텀 형식 BinaryOp의 선택적 파라미터

-[C++ 형식 CustomTypes::BinaryOp로 표현되며]

-m_binaryOp로 명명된 멤버에 저장되며,

-그것을 설정하는 함수는 setBinaryOpParam로 명명됨.

#'setBinaryOpParam'에 대한 구현이 생성됨.

BinaryOp BinaryOpParam

{

Cardinality=0-1;

Attribute=m_binaryOp;

SetFunction=setBinaryOpParam[Implement];

}

}

CPlusPlusBegin

$Inc

namespace

{

class $Class

{

$ParamAryTypes

public:

virtual void preExec();

virtual void exec();

virtual void postExec();

$ParamFns

//...

private:

double m_someVar;

$ParamAttrs

//...

};

//...

$ParamImpls

}//End namespace

CPlusPlusEnd

커스텀 형식을 사용하는 커스텀 테스트 클래스에 대한 C++(C++ for Custom Test Classes using Custom Types)

마지막으로 상기 컴파일러가 일단 이 프리 헤더를 처리하였으면, 그것은 상기 'MyFancyTest' 클래스에 대하여 다음과 같은 C++ 헤더 파일 MyFancyTest.h를 창출할 것이다. 즉,

#include <MyCustomTypes.h>

#include <ITest.h>

#include <FunctionalTest.h>

...

namespace

{

class MyFancyTest:public ITest,

public FunctionalTest

{

public:

typedef std::veetor<CustomTypes::Bar*> BarAry_t;

public:

virtual void preExec();

virtual void exec();

```
virtual void postExec();

public:

void setName(OFCString &name); #모든 시험에 대하여 자동

void setPatternTree(PatternTree *);

void addTestCondition(TestCondition *);

void addBarParam(CustomTypes::Bar *);

void setBinaryOpParam(CustomTypes::BinaryOp *);

...

private:

double m_someVar;

private:

OFCString m_name; #모든 시험에 대하여 자동

PatternTree *m_pPatList;

TestConditionPtrsAry_t m_testCondnsArray;

BarAry_t m_barsArray;

BinaryOp *m_binaryOp;

...

}; //End class MyFancyTest

...

inline void

MyFancyTest::addBarParam(CustomTypes::Bar *arg)

{

m_barsArray.push_back(arg);

return;

}

inline void

MyFancyTest::seBinaryOpParam(CustomTypes::BinaryOp *arg)
```

{

m_binaryOp=arg;

return;

}

}//End namespace

C8. 파라미터화(Parameterization)

상기한 바와 같이, 테스트 클래스에 대한 프리 헤더, 커스텀 흐름 가능 클래스, 또는 커스텀 함수 정의들은 파라미터화 규정 섹션을 통해 상기 클래스/함수로의 제한적인 자체 검사를 제공한다. 상기 컴파일러는 이 섹션을 상기 클래스/함수에 대한 파라미터화 인터페이스를 생성하도록 사용한다. 테스트 클래스들 및 흐름 가능 클래스들에 있어서, 그것들은 또한 계속해서 이 섹션을, 그 클래스의 인스턴스를 초기화하기 위하여 상기 테스트 계획 코드의 호출을 생성하도록 사용한다. 프리 헤더 및 대응되는 선언에 관하여 이하의 점들을 주의하여야 한다. 즉,

1. 모든 시험 또는 커스텀 흐름 가능 클래스 선언은 바람직하게 프리 헤더에 규정된다. 상기 프리 헤더의 파라미터 블록은 바람직하게 그러한 클래스가 규정될 수 있는 유일한 장소이다. (그러므로, 필연적 결과로서, 패턴 리스트 및 테스트 조건 규정과 같은 시험을 위한 "표준" 파라미터들은 또한 상기 프리 헤더의 파라미터 블록에 포함될 필요가 있다. 즉, 이렇게 함으로써, 모든 파라미터들, 즉 표준 및 CTs들이 균등하게 다루어질 수 있게 된다.)

2. 시험 또는 흐름 가능 클래스에 대한 프리 헤더에서 비 선택적인(즉, 영이 아닌 카디낼리티(cardinality)를 갖는) 것으로 정의된 모든 파라미터는 그 클래스의 인스턴스를 위한 상기 시험 블록 또는 흐름 가능 블록 선언에서 초기화되어야 한다.

3. 상기 시험/흐름 가능 블록의 파라미터의 초기화를 위하여 사용되는 객체들은 앞서 정의되어 있어야 한다.

4. 대체 지시자(replacement indicators)인 $Class, $Inc, $ParamAryTypes, $ParamFns, $ParamAttrs 및 $ ParamImpls 등은, 상기 사용자가 생성된 클래스 헤더 파일에서의 대응되는 생성된 코드가 삽입되도록 의도한 상기 프리 헤더의 사용자 코드 섹션 내의 정확한 위치에 나타나야만 한다. 특정의 코드가 각각에 대하여 생성되므로, 이들은 정확히 한 번만 나타나야 한다.

5. 상기 프리 헤더의 파라미터 블록의 파라미터 규정의 명칭(상기 예에서의 PListParam, TestConditionParam 또는 BarParam과 같은)은, 그 클래스의 인스턴스의 선언에서 사용되는 파라미터의 명칭이다.

6. 이하는 파라미터 규정에서 사용되는 기술자들의 구문이다. 즉,

a. 카디낼리티(cardinality): 이것은 지원될 이 형식의 파라미터들의 개수를 지시한다. 이하는 일 실시예에 의한 가능한 값들이다. 즉,

i. 1: 이 파라미터는 필수적이며, 단 한 번만 규정되어야 한다. 이 파라미터는 상기 파라미터의 형식의 객체로의 포인터로서 유지된다.

ii. 0-1: 이 파라미터는 선택적이다. 만약 규정되면, 그것은 단 한 번만 규정되어야 한다. 이 파라미터는 파라미터의 형식의 객체로의 포인터로서 유지된다.

iii. 1-n: 이 파라미터는 필수적이다. 게다가 이것에 대하여 다수의 값들이 지정될 수 있다. 그 값들은 규정의 순서에 따라 저장된다.

iv. 0-n: 이 파라미터는 선택적이다. 다수의 값들이 이것에 대하여 지정될 수 있다. 그 값들은 규정의 순서에 따라 저장된다.

상기에 있어서, 모든 규정된 값들은 상기 파라미터의 형식으로의 포인터에서 템플리트화된 STL 벡터 <>에 저장된다. 이 벡터의 형식은 $ParamAryTypes에 의하여 지시된 위치에서 정의되고 삽입될 것이다. 이 형식 선언들에 대한 억세스 레벨은 항상 대중적(public)이다.

b. 속성(Attribute): 이 형식의 파라미터 값(들)에 대한 저장소로서 사용하기 위한 C++ 변수의 명칭. 상기 명칭은 C++ 클래스의 사적 데이터 멤버(private data member)로서 축어적으로 재생성되며, C++ 식별자에 대한 요건에 부합되어야 한다. 이 속성의 형식은 다음 사항을 주의하여야 한다. 즉,

i. 이 속성의 형식은, 만약 단지 단일의 값들이 허용된다면, 파라미터의 형식으로의 포인터이다.

ii. 이 속성의 형식은, 만약 다수의 값들이 허용된다면(상기 참조), 파라미터의 형식으로의 포인터에서 템플리트화된 STL 벡터<>이다.

상기 속성은 테스트 계획에 의하여 창출되고 채워지는(populated) 객체들로의 참조를 보유하며, 이들 객체들을 소유하지 않는다. 상기 객체들의 수명은 상기 테스트 계획 자신에 의하여 관리된다.

c. 설정 함수(setFunction): 이 파라미터의 값을 설정하기 위하여 사용되는 함수의 명칭. 이하의 점들에 주의하여야 한다. 즉,

i. 상기 명칭은 축어적으로 재생성되며, 따라서 C++ 언어 요건에 부합되어야 한다.

ii. 이 함수의 억세스 레벨은 항상 대중적이다.

iii. 리턴 형식은 항상 'void'이다.

iv. 상기 함수는, 항상 포인터-대-파라미터 형식의 단일 인자만을 취한다.

값은 언제나 개별적으로 정해진다는 점을 주의하여야 한다. 즉, 여러개 값들의 규정을 허용하는 파라미터들에 대하여, 테스트 계획 내에서 생성된 코드는 이러한 기능을 반복적으로, 특정된 모든 값들의 각각에 대하여 한번씩 호출할 것이며, 그들 각각은 (위에서 설명한) STL 벡터에 부가될 것이다.

상기 함수 명칭에 뒤따르는 "[implement]"라는 선택적 키워드는, 이 함수에 대한 평범한 구현이, ($ParamImpls에 의하여 지시된 위치에서 삽입되어) 상기 클래스 헤더의 인라인 메소드로서 활용 가능하게 될 것이라는 점을 지시한다. 그렇지 않으면, 사용자는 상기 함수의 구현을 제공할 책임을 진다.

d. 기술(Description): 이 파라미터의 실행 시간 수정을 돕기 위하여 GUI 도구에 의하여 사용될 도구 팁인 문자열 리터럴(string literal). Xxx로 명명된 파라미터에 대한 커스텀 클래스에서 생성된 C++ 멤버 함수는:

String getXxxDescription() const;

가 될 것이다.

상기 함수는 규정된 문자열을 리턴할 것이다.

개별화에 의한 테스트 계획 사례(Test Plan Example with Customization)

이하에 개시된 내용은 소정의 개별화로 꾸며진 테스트 계획 사례이다. 즉,

```
#------------------------------------------------------------

# File MyCustomizedTestPlan.tpl

#------------------------------------------------------------
```

Version 0.1;

#

#이전과 같이 인입함...

#이하의 인입은 묵시적이지만,

#명시적으로 제공될 수도 있다.

Import FunctionalTest.ph;

#MyFlowables, MyFunctions 및 Functions에 대하여 인입함.

Import MyFlowables.ph;

Import MyFunctions.ph;

Import Functions.ph;

#--

테스트 계획의 시작

#--

TestPlan Sample;

#이 블록은 시험 선언들에 사용되는 패턴 리스트 파일 인가 명칭들

#및 패턴 리스트 변수들을 정의한다. 상기 파일 인가 명칭들은

#명명된 파일에서 패턴 리스트를 가리킨다. 상기 변수들은

#실행 시간에 상기 패턴 리스트 명칭들을 보유할 문자열

#변수들을 가리킨다. 사용자 정의 흐름 가능 객체는 APL을 통해

#이들 변수들의 값들을 설정할 수 있다.

PListDefs

{

#파일 인가 패턴 리스트 명칭들

p11A.plist:pat1Alist,

p12A.plist:pat2AList,

#패턴 리스트 변수들

plistXxx,

```
plistYyy,

plistZzz

}

#이전과 같이 소켓 정의, 사용자 변수 선언

#이전과 같이 테스트 조건들 TClMin, TClTyp, TClMax,

#TC2Min, TC2Typ, TC2Max의 선언

#

#기능적 시험 선언. "FunctionalTest"는 시험을 실행하고

#결과로서 0, 1 or 2를 리턴하는 C++ 테스트 클래스를 가리킨다.

#상기 테스트 조건 그룹 TCG1은 TC1Min이라는 테스트 조건을

#가리킴으로써 "min" 선택자로 선택된다.

#

#상기 컴파일러는 상기 인입된 FunctionalTest.ph 파일

#때문에 이것을 컴파일할 수 있음을 주의하여야 한다.

#

Test FunctionalTest MyFunctionalTest1Min

{

PListParam=patlAList;

TestConditionParam=TC1Min;

}

#

#이전과 같이 다음에 대한 추가적인 기능적 시험 선언

# MyFunctionalTest1Typ

# MyFunctionalTest1Max

*# MyFunctionalTest2Min

# MyFunctionalTest2Typ

# MyFunctionalTest2Max
```

```
#
#'MyFlowable'의 선언이 있음. 그것은 여기에서 사용되기
#전에 상기 흐름 가능에 의하여 초기화되는 'plistXxx'라는
#패턴 리스트 변수를 사용함.
#
#컴파일러는 인입된 MyFlowables.ph 파일 때문에 이것을 컴파일할 수 있음.
Flowable MyFlowable MyFlowable1
{
Int1=10;
Int2=20;
StringItem="Hello World";
PListParam=plistXxx;
}
#이전과 같이 'PassCount' 및 'FailCount'라는 카운터들...
#이전과 같이 흐름들. 'FlowTest1' 및 'FlowTest2'라는 흐름들은
#이전 예로부터 변경되지 않았음.
Flow FlowTest1
{
#...
}
Flow FlowTest2
{
#...
}
#
#이제, 주요 흐름인 'FlowMain'이 나타날 수 있음.
#그것은 이하와 같이 FlowTest1 및 FlowTest2를
```

#호출할 수 있는 유한 상태 머신을 구현함.

#-----------------------------------

Result 0 Result 1

#-----------------------------------

FlowMain_1 FlowMain_2 return 1

FlowMain_2 FlowMain_3 return 1

FlowMain_3 FlowMain_4 return 1

FlowMain_4 FlowMain_5 return 1

FlowMain_5 return 0 return 1

#

#여기서, 각 흐름 항목에 의하여 실행되는

#흐름 가능 인터페이스는 다음과 같다. 즉,

#---

흐름 가능(FlowItem) 실행되는 흐름 가능 인터페이스(IFlowable)

#---

FlowMain_1 MyFlowable1

FlowMain_2 DatalogStartFlow

FlowMain_3 FlowTest1

FlowMain_4 FlowTest2

FlowMain_5 DatalogStopFlow

#

Flow FlowMain

{

#

#첫번째로 선언된 흐름은 실행될 최초의 흐름이다.

#그것은 성공하면 FlowMain_IitializationFlow로 진행하고,

#실패하면 1을 리턴한다.

```
#
FlowItem FlowMain_1 MyFlowable1
{
Result 0
{
Property PassFail="Pass";
IncrementCounters PassCount;
#사용자 함수 호출
MyFunctions.Print("MyFlowable1 통과",
Functions.GetDUTID());
GoTo FlowMain_2;
}
Result 1
{
Property PassFail="Fail";
IncrementCounters FailCount;
#사용자 함수 호출
MyFunctions.Print("MyFlowable1 페일",
Functions.GetDUTID());
SetBin SoftBins."3GHzLeakage";
Return 1;
}
}
#
#성공하면 FlowMain_3로 진행하고,
#실패하면 1을 리턴함.
FlowItem FlowMain_2 DatalogStartFlow
```

```
{
Result 0
{
Property PassFail="Pass";
IncrementCounters PassCount;
#사용자 함수 호출
MyFunctions.Print("DatalogStartFlow 통과",
Functions.GetDUTID());
GoTo FlowMain_3;
}
Result 1
{
Property PassFail="Fail";
IncrementCounters FailCount;
MyFunctions.Print("DatalogStartFlow 페일",
Functions.GetDUTID());
Return 1;
}
}
#이 흐름 항목은 이전에 정의된 FlowTest1을 호출함.
FlowItem FlowMain_3 FlowTest1
{
Result 0
{
Property PassFail="Pass";
IncrementCounters PassCount;
#사용자 함수 호출
```

```
MyFunctions.Print("FlowTest1 통과",
Functions.GetDUTID());
GoTo FlowMain_4;
}
Result 1
{
Property PassFail ="Fail";
IncrementCounters FailCount;
#사용자 정의 함수 호출
MyFunctions.Print("FlowTest1 페일",
Functions.GetDUTID());
SetBin SoftBins."3GHzCacheFail";
Return 1;
}
}
#이 흐름 항목은 이전에 정의된 'FlowTest2'를 호출함.
FlowItem FlowMain_4 FlowTest2
{
Result 0
{
Property PassFail ="Pass";
IncrementCounters PassCount;
#사용자 함수 호출
MyFunctions.Print("FlowTest2 통과",
Functions.GetDUTID());
GoTo FlowMain_5;
}
```

```
Result 1
{
#FlowTest1은 패스, 그러나 FlowTest2 페일.
Property PassFail="Fail";
IncrementCounters FailCount;
#사용자 함수 호출
MyFunctions.Print("FlowTest2 페일",
Functions.GetDUTID());
SetBin SoftBins."3GHzSBFTFail";
Return 1;
}
}
FlowItem FlowMain_5 DatalogStopFlow
{
Result 0
{
#모두 통과!
Property PassFail="Pass";
IncrementCounters PassCount;
#사용자 함수 호출
MyFunctions.Print("모두 통과!",
Functions.GetDUTID());
SetBin SoftBins."3GHzAllPass";
Return 0;
}
Result 1
{
```

```
#FlowTest1 및 FlowTest2는 통과,

#그러나, DatalogStopFlow는 페일.

Property PassFail="Fail";

IncrementCounters FailCount;

#사용자 함수 호출

MyFunctions.Print("DatalogStopFlow 페일",

Functions.GetDUTID());

Return 1;

}

}

}
```

상기 코드에 관하여 이하의 점에서 주의하여야 한다. 즉,

1. 상기 PListDefs 섹션은 여기에서 몇몇 PList 명칭들과 또한 몇몇 PList 변수들을 가진다. 상기 PList 명칭은 시험에서 직접 사용될 수 있는 명칭들이다. 상기 PList 변수들은 시험에서 사용될 수 있는 변수들이며, 또한 그들의 값들은 개별화된 흐름 가능에서 코드에 의하여 실제의 PList로 실행 시간에 한정된다.

2. 상기 PListDefs 섹션은 선택적이다. 만약 나타나지 않는다면, 그 내용은 컴파일러에 의하여 다양한 시험 선언들로부터 추론될 것이다. 만약 나타난다면, 그것은, 더 선언하게 되더라도, 시험의 모든 사용되는 PList 파라미터를 선언하여야만 한다.

3. 실행 시간 API는 PList 변수들에게 값을 할당하는 데에 활용 가능할 것이다. 상기 테스트 계획 클래스는 다음의 함수를 갖는다. 즉,

```
Status SetPListVariable(const Tester::String& varName,

const Tester::String& fileQualifiedPListName);
```

상기 흐름 가능은, PListVariable을 특정한 PList에 한정시키는 데에 상기 함수를 사용할 수 있다.

4. 사용자 함수들 및 함수들은 전이 직전에 FlowItems에서 호출될 수 있으며, 그것은 제어를 다른 FlowItem으로 이전하거나 리턴하는 것의 어느 하나일 수 있다.

사용자 함수 호출을 위한 C++(C++ for User Function Calls)

흐름에서 커스텀 호출을 수반하는 예외를 가지고, 컴파일러에 의하여 생성되는 C++ 코드가 앞서 제시된 다양한 개별화 기술에 관하여 개시되었다. 흐름 항목(FlowItem)에서의 사용자 함수 호출은 각 흐름의 'IUserCalls' 멤버에 의하여 바람직하게 처리된다. 각 흐름은 바람직하게, 다음과 같은 단일한 가상의 멤버 함수를 이출(移出)하는 IUserCalls 인터페이스의 멤버를 갖는다. 즉,

```
class IUserCalls
```

```
{
public:
virtual void exec(const String& flowItemName,
unsigned int result)=0;
};
```

사용자 함수 호출을 가진 흐름에 마주치게 되면, 상기 흐름은 상기 인터페이스를 구현하는 클래스의 인스턴스로 채워지게 된다. 예를 들어, 상기 예에서의 FlowMain에서, 상기 흐름은 다음의 클래스의 인스턴스로 채워지게 될 것이다. 즉,

```
class FlowMain_UserCalls:public IUserCalls
{
public:
virtual void exec(const String& flowItemName,
unsigned int result)
{
if(flowItemName=="FlowMain_1")
{
//...
} else if(flowItemName=="FlowMain_2")
{
//...
} else if(flowItemName=="FlowMain_3")
{
switch(result)
{
case 0:
MyFunctions::Print("FlowTest1 통과",
Functions::GetDUTID());
return;
```

```
case 1:
MyFunctions::Print("FlowTest1 페일",
Functions::GetDUTID());
return;
default:
return;
}
}
else if(flowItemName=="FlowMain_4")
{
//...
}
else if(flowItemName=="FlowMain_5")
{
//...
};
}
};
```

상기 FlowItem::execute() 동작은 상기 흐름 항목의 명칭을 알고 있다. 그것이 포인터를 다음 흐름으로 리턴하기 전에, 그것은 그 자신의 흐름 항목 명칭 및 현재의 결과 값을 전달하면서 그 에워싼 흐름에 대하여 IUserCalls::exec()를 호출할 것이다. 이것은 필요로 하는 사용자 정의 함수를 수반하면서 상기한 코드가 실행되도록 한다.

C9. 테스트 프로그램의 컴파일(Test Program Compilation)

상기에서 설명한 바와 같이, 상기 테스트 계획 기술 파일은, 테스트 계획에서 사용되는 객체와 서로에 대한 그들의 관계를 규정한다. 본 발명의 일 실시예에 의하면, 이 파일은, 표준 인터페이스인 ITestPlan의 구현의 형태로 상기 사이트 제어기 상에서 실행될 수 있는 C++ 코드로 번역된다. 이 코드는, 상기 사이트 제어기 상으로 로드될 수 있는 윈도우즈 동적 링크 라이브러리(DLL)로 패키지화 될 수 있다. 상기 테스트 프로그램 DLL은, 상기 사이트 제어기 소프트웨어가 그것이 수록하는 상기 테스트 계획(TestPlan) 객체를 생성하고 리턴하는 데에 사용할 수 있는, 이미 알려진 표준 엔트리 포인트(standard known entry point)를 가지도록 생성될 수 있다.

테스트 계획 기술로부터의 구축(Construction from a Test Plan Description)

테스트 계획 기술로부터 ITestPlan의 구현으로의 변환 프로세스는 상기 테스트 프로그램 컴파일러 400에 의하여 달성된다. 이 프로세스는 전이 및 컴파일의 두 단계로 발생한다.

전이 단계 402에서는, 상기 컴파일러 400이, 상기 테스트 계획에 사용된 모든 시험 형식에 대한 상기 프리 헤더와 함께, 테스트 계획 파일(또한 그것이 인입하는 다양한 다른 파일들)을 처리한다. 이 단계에서는, 그것은, MSVC++(마이크로 비주얼 C++) 작업 공간 및 프로젝트 파일들, DLL "상용구(boilerplate)" 코드 등과 같은 다른 모든 지원 파일들과 함께, 상기 테스트 계획 객체에 대한 C++ 코드와, 마주치게 되는 시험 형식에 대한 C++ 헤더들을 창출한다. 상기 컴파일러 400은, 컴파일시의 오류 메시지가 생성된 코드 내부를 가리키는 대신에 기술 파일(description file)의 적절한 위치를 가리키도록 보증하기 위하여, 상기 생성된 코드 내에 파일 및 라인 지령(line directives)을 포함시킨다.

상기 컴파일러가 필요한 파일들을 생성한 다음에 발생하는 컴파일 단계에서는, MSVC++ 컴파일러와 같은 표준 컴파일러 404가 상기 파일들을 컴파일하고 DLL로 링크하기 위하여 발동된다.

상기 컴파일러는 입력으로서 유효한 테스트 계획 파일(그리고 관련된 모든 파일들)을 취하며, 필요에 따라, 테스트 계획 파일 및 상기 테스트 계획의 "인입" 지령에 의하여 표시된 모든 다른 파일들을 생성한다. 또한, 그것은 상기 테스트 계획 DLL을 구축하기 위한 MSVC++ "솔루션"을 생성한다. 예를 들어, 만약 상기 메인 파일(MyTestPlan.tpl)이 타이밍 정보를 통합하기 위하여 Timing1.tim을 포함한다면, 상기 컴파일러는 (다른 것들 중에서) 다음의 파일을 창출할 것이다. 즉,

MyTestPlan.h

MyTestPlan.cpp

Timing1.cpp

MyTestPlan.sln(MSVC++의 "솔루션" 파일)

MyTestPlan.vsproj(MSVC++의 "프로젝트" 파일)

모든 파일들이 창출(또는 갱신)된 후에, 상기 컴파일러는 그것이 창출한 상기 "솔루션"을 지정하면서 MSVC++ 응용프로그램을 발동시키고 DLL을 구축한다. 모든 오류 및/또는 경고들이 사용자에게 제시된다.

만약 상기 테스트 계획을 구축한 후에 상기 사용자가 Timing1.tim에 변경을 가하였다면, 상기 사용자는 그것을 MyTestPlan.tp1으로 전달하면서 상기 컴파일러를 발동시킬 것이다. 상기 컴파일러는 (타임 스탬프(timestamp) 정보에 의하여) 상기 메인 테스트 계획 파일이 변경되지 않았다는 것을 인식할 것이며, 그리하여 MyTestPlan.h/.cpp가 재창출되지 않을 것이다. 그러나, 상기 메인 테스트 계획 파일을 처리한는 동안, 그것은 상기 Timing.tim 파일이 변경되었음을 알게 될 것이다. 그러므로, 그것은 상기 Timing1.cpp 파일을 재창출할 것이고, MSVC++ 응용 프로그램을 발동시켜 상기 DLL을 재구축하게 할 것이다. 이것은 MyTestPlan.cpp를 재컴파일할 필요가 없게 하며, 상기 Timing1.cpp를 컴파일하고 상기 DLL에 재링크시키기만 하면 된다. 이러한 접근 방식은, 컴파일하는데에 상당한 시간을 요하는 대규모의 테스트 계획에 대한 재컴파일 및 재링크 시간을 감축시키는 데에 특별히 유용하다.

D. 테스트 프로그램의 실행(Running the Test Program)

상기 사이트 제어기 소프트웨어는 상기 테스트 프로그램 DLL을 그의 프로세스 공간(process space)로 로드하고, 상기 테스트 계획 객체의 인스턴스를 창출하기 위하여 상기 DLL 내에서 "공장(factory)" 함수를 호출한다. 일단 상기 테스트 계획 객체가 창출되면, 상기 사이트 제어기 소프트웨어는 상기 테스트 계획을 실행시키거나 필요한 임의의 다른 방식으로 그것과 상호 작용할 수 있다.

비상호작용적 구축(Non-Interactive Builds)

윈도우즈 환경에서 대부분의 C++ 소프트웨어 개발자들에게, 응용 프로그램(또는 DLL, 또는 라이브러리, 등)의 구축은, 개발 환경(MS 비주얼 C++, 볼랜드 C++, 또는 이와 유사한 것)의 개선, 코드의 편집, 및 (종종) 그 제품을 구축하는 단추를 누르는 것을 의미한다.

본 발명의 일 실시예에 의하면, 이와 유사한 한 세트의 활동들이 제공된다. 테스트 계획 개발자들은 코드를 편집하고 그들의 테스트 계획을 구축할 수 있어야 한다. 그러나, 테스터는 테스트 계획 개발자들이 결과인 테스트 계획 DLL을 생산하기 위하여 C++ 개발 환경을 개선해 나갈 것을 요구한다.

본 발명은 이것을 해결하기 위하여, 비상호작용적 구축(non-interactive build)라는 개념을 채용하였다. 비상호작용적 구축은 비상호작용적 모드에서 MS 비주얼 C++을 사용하는 구축으로 정의된다. 이것은 여전히 그러한 구축을 관리하기 위하여 다른 도구들을 상호작용적으로 사용할 수 있도록 허용한다. 단 하나의 암시는 비주얼 C++이 비상호작용적으로 사용된다는 것이다.

가정된 환경(Assumed Environment)

사용자 환경에 대한 소정의 가정을 가한다. 그 가정은 다음과 같다. 즉,

1. 상기 테스트 계획 개발자는 상기한 메소드들과 규칙들에 따라 그의 테스트 계획을 개발할 것이다.

2. 상기 테스트 계획 개발자는 C++에 관한 전문적 지식을 갖고 있지 않다.

3. 상기 테스트 계획 개발자는 파일들을 테스트 계획 DLL로 변환하기 위한 명령 라인 도구들(command line tools) 또는 GUI 도구들에 억세스할 수 있다.

단추 없는 응용 프로그램의 구축(Building Applications Without Buttons)

MS 비주얼 스튜디오(Visual Studio)와 비상호작용적으로 작업하기 위하여는 다음의 두 가지 접근 방식이 요구된다. 그 첫번째(그리고 가장 간단한) 방법은, 명령 라인 인터페이스(command-line interface)를 사용하는 것이다. 그 두번째(그리고 보다 복잡한) 방법은 자동화 인터페이스(Automation interface)를 사용하는 것이다. 이 섹션은 두 가지 접근 방식 모두에 관하여 기술한다.

프로젝트의 창출(Creating the Project)

비주얼 스튜디오를 비상호작용적으로 사용하기 위하여, 하나 또는 그 이상의 유효한 프로젝트를 수록한 작업 솔루션을 가지고 시작하여야 한다. 불행히도, 이것은, 명령 라인 또는 자동화 접근 방식의 어느 것으로부터도 달성될 수 없는 작업이다. 어떠한 방법도 프로젝트의 창출을 위한 메커니즘을 제공하지는 않는다. 그러나, 비주얼 스튜디오를 위한 프로젝트와 솔루션은 템프리트로부터 창출될 수 있다. 그러므로, 시작할 프로젝트 명칭과 템플리트가 주어지면, 비주얼 스튜디오를 위한 솔루션/프로젝트를 창출할 수 있다.

프로젝트를 채움(Populating the Project)

생산된 프로젝트에 새로운 파일을 추가하는 것은, 명령 라인은 이것을 지원하지 않기 때문에, 비주얼 스튜디오의 자동화 모델을 사용한다. 우리는 새로운 또한 기존의 파일을 프로젝트에 추가하는 두 가지 비주얼 스튜디오 매크로(macro)를 제공한다. 유사한 코드가, 동일한 과제를 수행하기 위하여, 액티브 스크립트 엔진(VBScript, JScript, ActivePerl, ActivePython 등과 같은)을 사용하는 외부 스크립트에 의하여 사용될 수도 있다. 그러므로, 우리의 코드 생성 도구는 새로운 파일들을 창출하고, 자동화 모델을 이용하여 그들을 기존의 비주얼 스튜디오 프로젝트에 추가할 수 있다. 파일들이 창출된 후에, 그들은 필요에 따라 도구들에 의하여 갱신될 수 있다.

프로젝트의 구축(Building the Project)

일단 솔루션과 프로젝트가 마련되면, 비주얼 스튜디오를 상기 테스트 계획을 구축하는 데에 비상호작용적으로 사용하는 데에는 몇 가지 선택 사항이 존재한다. 가장 간단한 선택 사항은 명령 라인에서 그것을 발동시키는 것이다. 그러한 명령 라인은 다음과 같이 보일 것이다. 즉,

devenv *solutionFile /buld solutionCfg*

여기서 solutionFile은 비주얼 스튜디오 솔루션 파일이며, solutionCfg는 이 솔루션 내의 프로젝트에 응용 가능한 특정의 구성이다. 다른 솔루션은, 자동화를 위하여 비주얼 스튜디오 객체 모델을 사용하는 것이다. 이에 의하여, 구축 및 구성 프로세스를 보다 섬세하게 제어할 수 있게 된다. 상기한 바와 같이, 그것은 명령 라인으로부터 프로젝트를 구축하기 위한 펄

(Perl) 스크립트의 리스트를 포함한다. 이 프로그램은, 구축을 위하여 프로젝트와 구성을 규정하는 구성 파일(그 프로젝트에 관한 다른 정보뿐만 아니라)을 읽으며, 상기 자동화 모델을 사용하여 그들 모두를 구축한다. 스크립트에서 자동화 객체를 사용하는 방법의 예로서 이 스크립트의 $msdev 객체의 사용을 보라.

디버거 지원(Debugger Support)

테스트 클래스의 개발자들이 그들의 작업을 확인하고 디버그(debug)하기 위하여, 그들은 그들에게 상기 사이트 제어기로 뚫고 들어가 그들의 코드들을 밟아 나갈 수 있게 해 주는 디버거로의 억세스를 필요로 한다. 상기 컴파일러에 의하여 생성된 코드는 MSVC++에 의하여 컴파일된 C++이므로, 우리는 상기 MSVC++ 디버거를 테스트 클래스 구현의 디버그에 사용한다. 이 특성은 테스트 클래스 개발자들이나 다른 C++에서 직접 작업하는 사람들만을 위한 것임을 주의하여야 한다. 다른 메커니즘이, 생성된 C++ 코드로의 직접적인 언급 없이 테스트 프로그램의 동작을 디버그하거나 밟아나가고자 하는 시험 엔지니어들에게 제공된다.

시스템 소프트웨어 환경(System Software Environment)

이 섹션은 상기 테스터의 일반적인 소프트웨어 환경, 즉 사용자 테스트 계획에 의하여 요구되는 파일을 위한 위치들(locations), 그러한 파일들을 위한 대체 위치들을 규정하기 위한 메커니즘들, 및 상기 테스트 계획들 및 모듈 제어 소프트웨어의 위치들을 규정하는 메소드들에 관하여 기술한다.

테스트 계획에 의하여 요구되는 환경(Environment Reqruied by Test Plan)

테스트 계획에 의하여 요구되는 이하의 것들에 대한 검색 경로(search path)의 실행 시간 구성과 함께 시스템 표준 위치들(system standard locations)은, 환경 구성 파일(environment configuation files)에 의하여 규정되는 "환경" 변수들에 의하여 구성될 수 있다. 즉,

1. 패턴 리스트들,

2. 패턴들,

3. 타이밍 데이터 및

4. 테스트 클래스 DLL들

이들은 텍스트 파일(text file)들이며, 다음과 같은 간단한 구문을 갖는다. 즉,

Tester_PATOBJ_PATH="pattern₩data₩;D:₩projects₩SC23₩patterns₩data"

본래의 OS가 지원하는 환경 변수를 통하는 대신에 텍스트 파일에서 정의된 그러한 "환경들"을 구비하는 이점은, 최대 문자열 길이 등과 같은 OS가 지원하는 환경 변수들이 갖는 공통된 제한에 의하여 그 구현이 제한받지 않는다는 것이다. 다음의 "환경" (설정) 변수들이 상기의 리스트에 나열된 엔티티들을 위하여 사용될 것이다. 즉,

패턴 리스트들: Tester_PATLIST_PATH.

패턴 객체 파일들: Tester_PATOBJ_PATH.

패턴 소스 파일들: Tester_PATSRC_PATH(이것은 선택적이다. 참고할 것).

타이밍 데이터 파일들: Tester_TIMING_PATH.

테스트 클래스 DLLs: Tester_TEST_CLASS_LIBPATH.

특별한 경우들을 지원하기 위하여, 유용한 디폴트 행위를 유지하면서, 세가지 구성의 레벨들을 제공한다. 이들은 우선 순위가 증가하는 순서대로 기술된다. 즉,

첫번째로, 시스템 환경 셋업 파일, 즉 $Tester_INSTALLATION_ROOT₩cfg₩setups₩Setup.env는 "환경" 변수들의 디폴트 값을 규정할 것이다. 만약 다른 구성 메커니즘이 활용 가능하지 않다면, 이 파일이 요구될 것이다. 일반적으로, 그것은 상기 시스템에서 실행되는 모든 테스트 계획에 대하여 활용 가능하다. 이 파일은, 상기한 세 가지 변수들에 대한 디폴트 값들을 할당하기 위하여 상기 설치자(installer)로부터의 입력을 가지고, 설치 도중에 상기 설치 및 구성 운영(ICM) 시스템에 의하여 창출된다. (상기 세 가지 변수들에 대한 시스템 디폴트와 별도로, 이 파일은 이하의 서브 섹션에서 기술되는 바와 같이, 소정의 다른 테스터 "환경" 변수들에 대한 시스템 디폴트도 포함하고 있다는 점을 주의하여야 한다.)

두번째로, 환경 셋업 파일은 상기 테스트 계획에 대한 실행 시간 인수로서 상기 사용자에 의하여 규정될 수 있다. 실행 시간 구성의 상기 변수들은 디폴트 정의에 대하여 우선권을 가질 것이다.

마지막으로, 테스트 계획은 그 실행에서 사용될 환경 변수들을 규정하는 특별한 블록을 사용할 수도 있다. 상기 테스트 계획에서 정의된 변수들은 시스템 디폴트 파일이나 상기 사용자 정의 파일에 대하여 우선 순위를 가질 것이다.

일반적으로 모든 필요한 변수들은 상기한 메커니즘들 중의 어느 하나를 통하여 정의되어야 한다. 만약 변수가 정의되지 않으면, 실행 시간 오류가 발생할 것이다.

다른 환경 설정들(Other Environment Setups)

사용자 테스트 계획에 의하여 요구되는 "환경" 변수들 이외에도, 이하의 두 가지 "환경" 변수들이 테스트 환경에 의하여 요구된다. 즉,

1. Tester_TEST_PLAN_LIBPATH: 이것은 상기 시스템 제어기가 로드되어야만 하는 사용자 테스트 계획 DLL을 발견하기 위하여 사용할 검색 경로를 규정한다. 사용자 핀 기술 및 소켓 파일들을 발견하기 위하여 동일한 검색 경로가 사용될 것이라는 점을 주의하여야 한다. 상기 ICM으로의 설치시에 규정되는 이 변수에 대한 디폴트 값은 상기 ICM에 의하여 $Tester_INSTALLATION_ROOT₩cfg₩setups₩Setup.env의 파일에 저장된다.

2. Tester_MODULE_LIBPATH: 이것은, 시스템이 벤더 제공 하드웨어 모듈 제어 소프트웨어 DLL들을 로드하기 위하여 사용할 검색 경로를 지정한다. 상기 구성 관리 데이터베이스(CMD)로부터 추출되는 이 정보는 역시 상기 ICM에 의하여 $Tester_INSTALLATION_ROOT₩cfg₩setups₩Setup.env의 파일에 저장된다.

사용자는 상기 Tester_TEST_PLAN_LIBPATH 변수에 대하여 상기 Setup.env 파일에 주어진 값을 무시할 수 있지만, 만약 사용자가 상기 벤더 제공 하드웨어 모듈 제어 소프트웨어 DLL들에 대한 검색 경로를 명시적으로 바꾸기를 원하지 않는 한, 상기 Tester_TEST_PLAN_LIBPATH에 대한 상기 Setup.env 파일에서 주어진 값은 사용자에 의하여 변경되어서는 안된다.

검색 경로 규정 의미론(Search Path Specification Semantics)

검색 경로를 규정하는 상기 "환경" 변수에 관하여는 이하의 점들에 주의하여야 한다. 즉,

1. 각각은, 시스템이 특정 형식의 참조된 파일을 발견하기 위하여 검색할 세미콜론(";")으로 구분된 디렉토리 명칭의 리스트가어야 한다.

2. 그러한 "환경" 변수의 값에 대한 최초의 시스템 열람 이후에, 그 값에 대하여 (예를 들어, 환경 구성 파일을 편집함으로써) 사용자가 가한 어떠한 변경도 상기 사용자가 상기 시스템에 명시적으로 "통지"한 경우에만 상기 시스템에 의하여 등록될 것이다.

3. 상기 테스터가 동작하는 것과 같은 분산형 환경"에서는 현재 작업 디렉토리(currently working directory; CWD)라는 표기가 사용자가 직관적으로 원했던 것이 아닐 수도 있기 때문에, 현재 작업 디렉토리에 관련된 경로는 애매한 결과를 가져올 수 있음에 따라, 상기 검색 경로의 상대적 경로명들은, 관련된 환경 변수(루트(root)를 정의하는 기능성을 제공하는)의 특정 설정에 관련된 것으로 해석될 것이다. 상기 검색 경로의 모든 상대적 경로명이 관련된 것으로 가정하는 루트를 지정하는 이 관련된 환경 변수는, 상기 "Tester_INSTALLATION_ROOT" 변수로서, 그것은 사용자 시스템에서의 상기 테스터가 설치된 가장 높은 레벨(즉, "루트")의 디렉토리의 위치를 준다.

4. 디렉토리 엔트리들은 문자 세트 [₩/:*?"<>|;]의 문자를 포함할 수 없다. 세미콜론(";")이외에는 이 세트의 모든 다른 문자들은 윈도우즈 파일 명칭으로서 규칙에 맞지 않는다는 점을 주의하여야 한다. 상기 세미콜론(";")은 검색 경로에서 엔트리들을 구별하는데 사용되므로 검색 경로 엔트리에는 사용되어서는 안된다. 경로명들은 내부에 공백을 가질 수 있으나, 경로명의 직전 또는 직후에(즉, 경로명에서 공백이 아닌 첫번째 문자 앞과 그 마지막 문자 뒤에) 나타나는 모든 공백들은 경로명의 일부로서 고려되지 않을 것이며 무시될 것이다.

5. 검색 경로 디렉토리들은 정의에서 마주치게 되는 순서대로 검색될 것이다. 파일의 첫번째 발생은 선택된 것이 될 것이다.

E. 패턴들(Patterns)

매우 큰 규모의 테스트 패턴 파일의 세트를 효율적으로 관리하고, 처리하며 로딩하는 것은 본 발명의 일 실시예에 의한 프레임워크의 중요한 구조적 국면이다. 구조적 패턴 리스트들이라는 아이디어는, 사용자에게 시스템의 다루기 쉬운 개념화 및 사용의 편의를 제공하는 효과적인 도구로 여겨진다.

피시험 디바이스로의 자극은 테스트 벡터를 통하여 테스트 시스템에 대하여 활용 가능하게 된다. 벡터들은 순차적(또는 선형), 스캔(scan) 또는 알고리즘 패턴 생성기 도출적(Algorithmic Pattern Generator(APG)-derived)인 것으로 카테고리화 된다. 본 발명의 일 실시예에 의한 시스템에 있어서, 테스트 벡터들은 시험시에 상기 피시험 디바이스에 인가되는 패턴에 의하여 조직화된다. 패턴은 상기 사용자의 테스트 프로그램의 패턴 객체로 표현된다. 시스템에서, 패턴들은 패턴 리스트 객체들로서 프로그램적으로 표현되는 패턴 리스트들로 조직화된다. 패턴 리스트 객체는 패턴들 또는 다른 리스트들의 정리된 리스트를 나타낸다. 정리는 리스트 컴포넌트들의 선언의 순서에서 묵시적이다. 만약 하나의 패턴이 필요하다면, 그것은 혼자서 리스트으로 캡슐화되어야 한다는 점을 주의하여야 한다.

상기 사용자의 테스트 프로그램의 패턴 리스트 객체는 디스크 상의 패턴 리스트 파일과 관련되는데, 상기 패턴 리스트 파일은 상기 패턴 리스트의 실제 정의를 수록한다. 그러므로, 패턴 리스트의 내용은 관련된 디스크 파일의 내용에 의하여 동적으로 결정된다(이에 대한 보다 상세한 내용은 후술한다.).

패턴 리스트의 정의는 상기 패턴 리스트에 대한 명시적 명칭을 제공하며, 관련된 파일 명칭을 통해 패턴들 및/또는 다른 패턴 리스트들의 정리된 리스트를 식별한다. 그것은 또한 실행 선택 사항의 규정을 제공하며, 상기 선택 사항은 패턴 리스트들 및 패턴들에 모두 적용될 수 있으므로, 상기 실행 선택 사항은 패턴 객체가 기술된 후에 상세히 기술된다. 상기 패턴 리스트는 다음의 규칙을 따라야만 한다. 즉,

file-contents:

version-info global-pattern-list-definitions

version-info:

Version *version-identifier;*

global-pattern-list-definitions:

global-pattern-list-defznition

global-pattern-list-definitions global-pattern-list-definition

global-pattern-list-definition:

global-pattern-list-declaration {list-block}

global-pattern-list-declaration:

GlobalPList *pattern-list-name options*$_{opt}$

list-block:

list-entry

list-block list-entry

list-entry:

pattern-entry;

pattern-list-entry;

global-pattern-list-definition;

local-pattern-list-definition;

pattern-entry:

Pat *pattern-name options*$_{opt}$

pattern-list-entry:

PList *pattern-list-reference options*$_{opt}$

pattern-list-reference:

pattern-list-qualified-name

file-name':'pattenr-list-qualified-name

pattenr-list-qualified-name:

pattern-list-name

pattern-list-qualified-name'.'pattern-list-name

local-pattern-list-definition:

local-pattern-list-declaration {list-block}

local-pattern-list-declaration:

LocalPList *pattern-list-name options*$_{opt}$

options:

option

options option

option:

[option-definition]

option-definition:

option-name option-parameters$_{opt}$

option-parameters:

option-parameter

option-parameters ',' option-parameter

이하는 상기에 사용된 정의되지 않는 비종단자(non-terminals)의 기술이다. 즉,

1. 버전-식별자(*version-identifier*): 세트 [0-9]로부터의 하나 또는 그 이상의 문자들의 시퀀스로서, 첫번째 문자는 숫자이어야 함.

2. 명칭(*name*): 세트 [a-zA-Z_0-9]로부터의 하나 또는 그 이상의 문자들의 시퀀스로서, 첫번째 문자는 세트 [a-zA-Z_]로부터의 문자이어야 함.

3. 패턴-리스트-명칭(*pattern-list-name*): 세트 [a-zA-Z_0-9]로부터의 하나 또는 그 이상의 문자들의 시퀀스로서, 첫번째 문자는 세트 [a-zA-Z_]로부터의 문자이어야 함.

4. 파일-명칭(*file-name*): 유효한 윈도우즈 파일 명칭(파일 명칭 내에 공백이 포함되어 있다면, 반드시 큰 따옴표로 에워싸여야 함.). 이것은 단순한 파일 명칭, 즉 디렉토리 컴포넌트를 가지지 말아야 한다는 점을 주의하여야 함. 패턴-리스트-참조는 그 파일 내의 패턴 리스트를 가리키는 내적(internal)이거나, 다른 파일의 것을 가리키는 외적(external)의 어느 하나이어야 함. 외적 참조는 파일-명칭에 의하여 인가되어야 할 필요가 있음.

5. 선택 사항-명칭(*option-name*): 세트 [a-zA-Z_0-9]로부터의 하나 또는 그 이상의 문자들의 시퀀스로서, 첫번째 문자는 세트 [a-zA-Z_]로부터의 문자이어야 함.

6. 선택 사항-파라미터(*option-parameter*): 세트 [a-zA-Z_0-9]로부터의 하나 또는 그 이상의 문자들의 시퀀스.

패턴 리스트 파일은 코멘트를 지원하는데, 그것은 패턴 리스트 파일 분석기에 의하여 무시되도록 되어 있다. 코멘트들은 '# ' 문자로 시작되며 그 줄의 끝까지 연장된다.

E1. 패턴 리스트를 위한 규칙(Rules for Pattern List)

패턴 리스트를 위한 정적 또는 컴파일시 규칙은 명칭들의 선언 및 해석을 지배한다. 패턴 리스트 언어의 명칭들은 'global-pattern-list-definitions' 및 'local-pattern-list-definitions'에 의하여 선언된다. 그들은 'pattern-list-references'에 의하여 참조된다. 이하는 이들 선언들 및 참조들을 지배하는 소정의 규칙들이다. 즉,

1. 'global-pattern-list-definitions' 또는 'local-pattern-list-definitions'은 패턴 리스트의 명칭을 선언한다. 'pattern-list-references'는 선언된 패턴 리스트의 명칭을 참조한다. 전역 패턴 리스트의 명칭들은 전역적으로 알려진다. 지역 패턴 리스트의 명칭들은 그들이 선언된 리스트 블록 내에서만 알려진다. 그들은 그 리스트 블록에서 직접 인가 없이 지칭될 수 있다. 보다 깊이 함유된(nested) 선언에서는, 지역 패턴 리스트는 인가된 명칭에 의하여 지칭될 필요가 있다.

2. 지역 패턴 리스트의 명칭들은 에워싸는 패턴 리스트의 범위 내에서 알려지며, 전역 패턴 리스트의 명칭들은 상기 시스템의 범위 내에서 알려진다. 예를 들어,

GlobalPList G1

{

LocalPList L1

{

LocalPList L2

{

...

}

GlobalPList G2

{

...

}

PList L2; #OK. 명칭 L2는 이 범위 내에서 알려짐

PList G2 #OK. 명칭 G2는 전역적임.

}

PList L2; #오류. 명칭 L2는 여기에서는 알려져 있지 않음.

PList L1.L2; #OK. 명칭 L1은 여기에서 알려져 있음. L2는

인가에 의하여 알려짐.

PList Gl.L1.L2; #OK. G1에 의한 인가는 필요하지 않으며 허용됨.

PList G2; #OK. 명칭 G2는 전역적임.

}

3. 전역적 패턴 리스트는 패턴 리스트 파일의 최외각 레벨에서 정의될 수 있으며, 또는 에워싸는 패턴 리스트 내에서 함유된 것으로서 정의될 수 있다. 그러나, 함유(nesting)은 단지 편의를 위한 것이다. 그들은 개념적으로 파일의 최외각 레벨에서 전역적 패턴 리스트으로서 정의된다. 함유된 전역적 패턴 리스트는 동일한 명칭의 최외각(함유되지 않은) 전역적 패턴 리스트과 의미론적으로 등가이다. 그러므로, 예를 들어,

GlobalPList G1

{

GlobalPList G2...

}

는 의미론적으로 다음과 등가이다. 즉,

GlobalPList G1

{

PList G2; #G2를 참조함

}

GlobalPList G2...

4. 모든 전역적 패턴 리스트들은 고유하게 명명됨.

GlobalPList G1

{

#이것은 그것으로의 참조를 바로 여기에서 가짐으로써,

#최외각 레벨에서 선언된 것과 마찬가지이다.

GlobalPList G2

{

...

}

}

#이 선언은, 명칭 G2가 이미 잡혀 있으므로, 여기 또는

#다른 어느 파일에서도 오류가 될 것이다.

GlobalPList G2 #오류. 전역적 명칭 G2는 잡혀있음.

{

...

}

5. 지역 패턴 리스트들은 언제나, 또한 상기 지역 패턴 리스트의 명칭의 범위를 정하는, 에워싸는 패턴 리스트 내에 그들의 정의를 함유되도록 한다. 지역 패턴 리스트들은 그들의 에워싸는 패턴 리스트 내에서 고유하게 명명된다. 지역 패턴 리스트들은 의미론적으로 패턴 리스트 파일의 최외각 레벨에서 발생되지 못한다.

GlobalPList G1

{

LocalPList L1

{

}

LocalPList L2

{

LocalPList L1 #OK. 지역 명칭 L1은, L2에 의하여

#정의된 에워싸는 범위 내에서

#직접 선언되지 않는다.

{

}

PList L1; #OK. L2에 선언된 L1을 가리킴.

PList G1.L1; #OK. G1에 선언된 L1을 가리킴.

}

#오류. G1에 의하여 정의된 에워싸는 범위가 이미 그 내에서

#선언된 L1을 가지고 있을 때, L1의 명칭을 재 선언함.

LocalPList L1;

{

}

}

6. 각 패턴 리스트 파일은 하나 또는 그 이상의 전역 패턴 리스트들에 대한 정의를 수록한다. 이것은 상기 구문을 곧바로 뒤따른다. 상기 최외각 레벨은 전역-패턴-리스트-정의(global-pattern-list-definition)이며, 적어도 하나의 그것들이 존재하여야 한다.

7.상기 패턴-명칭(pattern-name)은 패턴으로의 참조이며 Pat 키워드를 뒤따른다. 그것은, 상기 패턴 명칭에 뒤이어 .pat를 첨부함으로써 획득되는 명칭을 갖는 패턴 파일 내의 패턴을 참조한다. 이 파일은 패턴에 대하여 정의된 검색 경로를 따라 획득되는 파일을 표시한다.

8. 패턴-리스트-참조(pattern-list-reference)는 상기 PList 키워드를 뒤따르는 패턴 리스트으로의 참조이다. 상기 참조는, 점으로 구분된 명칭들의 리스트인 인가된 패턴 리스트 명칭이 뒤따르는 선택적인 파일 명칭으로 구성된다. 그러므로, 예를 들어, 다음의 내용이 패턴-리스트-참조가 될 수 있다. 즉,

PList foo.plist:G1.L1.L2.L3;

이것은, 파일 foo.plist의 전역 패턴 리스트 G1에 함유된 L1에 함유된 L2에 함유된 지역 패턴 리스트 L3를 참조한다. 상기 명칭에서 가장 왼쪽의 명칭 부분(leftmost name segment)은 G1이다.

상기 가장 왼쪽의 명칭 부분은 전역 패턴 리스트, 또는 참조 지점으로부터 가시적인 지역 패턴 리스트의 어느 하나로 해석되어야 한다.

패턴-리스트-참조의 명칭 해석은 다음과 같이 진행된다. 즉,

1. 각 명칭 부분은 그 앞의 접두어의 문맥에서 선언된 명칭으로 해석되어야 한다.

2. 만약 파일 인가가 존재한다면, 상기 가장 왼쪽의 명칭 부분은 상기 명명된 파일에서 선언된 전역 패턴 리스트으로 해석되어야 한다.

3. 만약 파일 인가가 존재하지 않는다면, 상기 가장 왼쪽의 명칭은 그 에워사는 범위 내에서의 지역 패턴 리스트으로, 그리고 그것이 실패하면, 그 다음의 에워싸는 범위로 해석되는 식으로 하여, 에워싸는 전역 범위까지로 해석될 수 있다.

4. 범위의 검색을 가장 가까운 에워싸는 전역 범위로 제한하는 것은, 마치 그들이 상기 패턴 리스트 파일의 최외각 레벨에서 선언된 것처럼, 상기 전역 범위의 의미론을 보존하기 위하여 필요하다. 만약 상기 함유된 전역 범위가 상기 최외각 레벨에서 문자 그대로 (등가적으로) 선언되었다면, 상기 명칭 해석 검색은 그 범위를 조사한 후에 종료될 것이다.

5. 만약 상기 참조가 이전 단계에서 해석되지 않았다면, 상기 가장 왼쪽의 명칭 부분은, 같은 이 파일 내에서 전역 패턴 리스트으로 해석될 수 있다.

6. 만약 상기 참조가 이전 단계에서 해석되지 않았다면, 상기 가장 왼쪽의 명칭 부분은, 상기 가장 왼쪽의 명칭 부분에 .plist라는 접미사를 추가함으로서 상기 파일의 전역 패턴 리스트으로 해석될 수 있다.

7. 만약 상기 참조가 이전 단계에서 해석되지 않았다면, 상기 참조는 오류이다.

상기한 바와 같이, 상기 규칙은, 상기 가장 왼쪽의 명칭 부분이 참조 지점으로부터 가시적인 지역 패턴 리스트가나 또는 전역 패턴 리스트의 어느 하나로 해석된다는 것을 알려준다.

이하는 이러한 아이디어의 예를 개시한다. 즉,

```
GlobalPlist G1

{

PList G3; #OK. 이 파일의 후반의 패턴 리스트를 가리킴.

PList G4; #OK. 파일 "G4.plist"의 패턴 리스트를 가리킴.

#OK. "my_plists.plist" 파일의 G1을 가리킴.

PList my_plists.plist:G1;

#OK. "my_plists.plist" 파일의 패턴 리스트를 가리킴.

#인가된 명칭은, G1으로 명명된 전역 패턴 리스트의

#범위에서 선언된

#L1으로 명명된 지역 패턴 리스트의 범위에서 선언된 L2로
```

```
#명명된 지역 패턴 리스트를 가리킴.
PList my_plists.plist:Gl.Ll.L2;
LocalPList L1
{
LocalPList L2
{
}
}
PList L1; #OK. G1이 에워싸는 범위에서
#선언된 L1을 가리킴.
}
GlobalPlist G2
{
LocalPList L2
{
}
GlobalPList G3
{
LocalPList L3
{
}
PList L1; #오류. 여기 또는 다른 에워싸는
#범위에서 L1은 선언되지 않음.
#오류. 명칭 L2는 이 범위에서 선언되지 않았음.
#또한, L2가 에워싸는 범위에서 선언되었다 하더라도,
#이 범위는 전역적이며, 따라서 더이상의 에워싸는
#범위는 검사되지 않음.
```

#

#이하의 LocalPList L3의 명칭 L2와 대조.

PList L2;

PList G1.L1; #OK. G1내의 L1을 가리킴.

#오류. G3는 실제로 G1 내에서 함유되지 않았음.

#G3는 전역적이기 때문에, 그것은 실제로 최외각

#레벨에서 선언되며, G1도 같음. 따라서, G1.G3는 무의미함.

PList G2.G3.L3;

}

LocalPList L3

{

#OK. G2.L2를 가리킴. 에워싸는 전역 범위는 G2이며, 명칭 L2는 #G2내에서 선언됨.

PList L2;

}

}

모든 패턴 리스트 파일의 명칭 및 패턴 파일의 명칭은 그들을 사용하는 테스트 계획에 걸쳐 고유할 것이 요구된다.

패턴 리스트 참조는 같은 파일 내에서의 참조의 이전 또는 이후에 정의된 패턴 리스트를 가리킬 수 있다.

재귀적인(recursive) 또는 상호간에 재귀적인(mutually recursive) 리스트 정의는 허용되지 않는다. 상기 패턴 리스트 파일의 구문에는 사용자가 그러한 정의를 창출하지 못하게 하는 어더한 것도 존재하지 않는 한편, 상기 분석기(parser)는 그러한 조건을 검출한 경우 오류를 알릴 것이다. 그러한 조건의 검출에는 소정의 비용이 요구된다는 것을 주의하여야 한다. 사용자는, 상기 입력 공간이 상호간에 재귀적인 정의로부터 자유롭다는 것을 보장할 책임을 가정할 수 있다면, 그 점검을 꺼둘 수도 있다.

GlobalPList G1

{

LocalPList L2

{

LocalPList L3

{

#오류. L2는, L2를 실행하는 L3를 실행함.

#이것은 L2로의 재귀적 참조임.

PList L2;

PList G2;

}

}

}

GlobalPList G2

{

#오류. G1.L2는 G1.L2을 실행하는 G2를 실행하는 L3를 실행함.

#G1.L2에 대한 상호 재귀적 참조임.

PList G1.L2;

}

패턴 및 패턴 리스트의 구문적 기술은, 선택 사항이 그들 상에서 규정될 수 있게 한다. 일반적으로, 선택 사항은 벤더에 특유하다. 상기 구문은, 임의의 패턴 또는 패턴 리스트가 다수의 파라미터를 각각 가진 다수의 선택 사항을 규정할 수 있게 한다. 대부분의 벤더들에 의해 인식될 몇 개의 지원된 선택 사항들이 기술된다.

패턴 실행 시퀀스를 정의한 후에, 패턴 트리(pattern tree)의 동적(즉, 실행) 의미론에 관하여 기술한다.

E2. 패턴들(Patterns)

도 6은 본 발명의 일 실시예에 의한 패턴 컴파일러 602 및 패턴 로더(loader) 604를 도시한다. 상기 패턴의 사용자 정의 내용은 패턴 소스 파일 606에서 활용 가능한데, 그것은 평범한 텍스트 파일이다. 패턴 컴파일러는, 소스 파일을 상기 테스터 하드웨어 상으로 로드하기에 적합한 모듈 특유적 포맷으로 컴파일하는 책임을 가지는데, 이 후자의 파일은 패턴 객체 파일이라 불린다. 이하는 그의 일반적인 속성들이다. 즉,

1. 패턴 객체는 사용자에 의하여 창출되지 않는다. 오히려, 사용자는 항상 패턴 리스트를 다루며, 다른 패턴 리스트들 및/또는 패턴들의 모음이다. 패턴 리스트 객체는 그에 수록된 패턴 객체들을 창출하고, 소유하며 유지하는 한편, 필요한 경우 사용자가 억세스할 수 있게 한다.

2. 패턴은 하나의 테스트 계획 내에서 고유하게 명명된다. 즉, 하나의 테스트 계획 내에 두 개의 패턴이 동일한 명칭을 가질 수 없다. 패턴의 명칭은 그것을 포함하는 파일의 명칭으로부터 구별되어야 한다. 상기 패턴 파일 명칭은 패턴을 가리키기 위하여 패턴 리스트 파일에서 사용되는 것인데, 상기 패턴의 실제 명칭은 상기 패턴 파일내에서 정의된다.

본 발명의 일 실시예에 의하면, 일반적으로 하나의 피시험 디바이스는 서로 다른 벤더로부터의 테스터 모듈에 접속될 수 있다. 이것은 전체적인 패턴의 컴파일-로드-실행의 체인(chain)에 대한 암시를 갖는다. 주요한 것들을 이 섹션에서 기술한다.

E3. 패턴 컴파일(Pattern Compilation)

그러므로, 패턴 컴파일러 602는 (사용되는 벤더 특유적 디지털 모듈의 관점에서) 특정의 사이트 구성을 목표로 할 필요가 있다. 이 논의의 나머지 부분에 대하여, "모듈"이라는 용어는 예를 들어 디지털 모듈을 가리키는 데에 사용된다. 서로 다른 벤더들로부터의 모듈들 608을 상기 시스템으로 통합하기 위하여는, 이하의 절차들이 우선된다. 즉,

1. 각 모듈 벤더는 그 자신의 모듈 특유적 패턴 컴파일러 601을 동적으로 로드할 수 있는 라이브러리나 별도의 실행 가능한 형태로 제공할 책임을 질 것이다. 이 컴파일러 라이브러리/실행 가능은 최소한 잘 알려진 다음을 인수로 갖는 compile() 함수를 제공할 것이다. 즉,

a. (하나 또는 그 이상의) 패턴 소스 파일 경로명들의 배열,

b. 핀 기술 파일 명칭,

c. 소켓 파일 명칭,

d. 컴파일된 객체의 목적지를 규정하는 선택적인 디렉토리 경로명,

e. 임의의 벤더 특유적 파라미터들(다른 벤더에 의하여 무시될 수 있는 것)을 허용하는 문자열 명칭/값의 쌍들의 선택적인 배열

2. 상기 패턴 소스 파일은 두 가지의 서로 다른 형식의 섹션들을 공급해준다. 즉,

a. 모든 컴파일러들에 의하여 억세스될 수 있는 (그러나 반드시 사용되어야 하는 것은 아닌) 정보를 수록한 "공통" 섹션, 및

b. 각각 고유의 벤더 코드에 의하여 식별되는, 하나 또는 그 이상의 벤더 특유적 섹션으로서, 특정 벤더의 컴파일러에 의하여 사용될 수 있는 정보에 대한 것.

3. 벤더의 컴파일러는 패턴 객체 파일을 직접 창출하지 않을 것이다. 대신에, 상기 테스터는 객체 파일 관리자(OFM) 614에 의하여 관리되는 패턴 객체 "메타 파일(metafile)" 612를 제공할 것이며, 상기 객체 파일 관리자는 패턴 컴파일러의 일부이다. 상기 패턴 컴파일러는, 시스템 제어기로서 행동하는 컴퓨터 상에, 또는 예를 들어 상기 시스템 제어기가 접속된 네트워크 상에서, 오프라인으로, 위치할 수 있다. 지금까지 추상적인 용어로서 언급된 상기 "패턴 객체 파일"은 실제로는 이 객체 메타 파일이다. 상기 객체 메타 파일은, 소스 파일의 확장자가 객체 파일 확장자로 대체된 패턴 소스 파일과 같은 명칭으로 명명될 것이다. 상기 OFM은 이 파일을 읽고 쓰기 위하여 응용 프로그램 프로그래밍 인터페이스(application programming interface; APIs)를 제공할 것이다. 상기 객체 메타 파일은 다음을 저장할 저장고(provisions)를 제공할 것이다. 즉,

a. 공통의 헤더 정보,

b. 대응되는 모듈을 식별하는 정보 및 상기 모듈에 대한 패턴 데이터의 위치를 포함하는 모듈 특유적 헤더 정보,

c. 모듈 벤더에 의하여 요구되는 것으로서 조직화되고 상기 모듈 벤더에 의하여 해석될 수 있는 모듈 특유적 패턴 데이터.

상기 OFM APIs는 모듈 벤더의 컴파일러가 모듈 특유적 헤더 정보 및 데이터를 상기 객체 메타 파일에 기록할 수 있게 한다. 상기 객체 메타 파일의 이 레이아웃(layout)은, 목표 사이트의 두 개 또는 그 이상의 모듈들이 동일한 경우에도, 패턴 데이터가 개별 모듈 기반으로 조직화 되게 한다는 점을 주의하여야 한다.

추가적인 벤더 제공 구성 정보가, 직접 메모리 억세스(DMA)와 같은 효율적인 데이터 통신을 이용할 수 있는 모듈 특유적 하드웨어 로딩 정보의 생성을 가능하게 하기 위하여, 패턴 컴파일러에 의하여 요구될 수도 있다는 점에 유의한다.

E4. 모듈을 위한 패턴 로드(Pattern Load for a Module)

각 모듈 벤더는 일반적인 절차를 따라 그 자신의 패턴 로딩 메커니즘 615를 제공할 책임을 질 것이다. 특정 모듈 608의 상기 패턴 객체 메타 파일 612는 서로 다른 섹션 616에 모듈 특유적 데이터를 저장한다. 상기 벤더 구현은, 상기 패턴 객체

메타 파일로부터 관련된 모듈 특유적 섹션으로 억세스하기 위하여 상기 OFM APIs를 이용할 것이다. 상기 테스터 프레임 워크는, 상기 메타 파일의 적당한 섹션으로부터 상기 모듈 특유적 데이터를 모듈로 차례로 로드하기 위하여 각 모듈의 로드 메소드(load method)를 호출할 책임을 질 것이다.

E5. 패턴 파일들(Pattern Files)

각 컴파일러 벤더가 패턴을 위한 완전히 서로 다른 평범한 텍스트 포맷을 규정하게 할 수 있으며, 그것은 실제로는 대부분의 경우 필요할 것이다. 그러나, 일반적으로, 모든 벡터에 대하여 모듈들에 걸쳐 일관되고 동일한 의미론이 필요한 주기 기반(cycle-based)의 테스트 환경에 있어서, 패턴 파일에 대한 공유되고 일반적인 구문은 단순히 희망 사항이 아니며 필요한 것일 수 있다. 이 공유된 구문은 상기 패턴 소스 파일에서 "공통" 섹션에 대하여 규정된 것이다. 사실상, 대부분의 경우에, 상기 "공통" 섹션이 상기 패턴 파일에서 요구되는 (헤더 정보 이외에) 유일한 섹션이며, 모든 벤더의 컴파일러가 단지 그 섹션과만 작동하는 것을 상상할 수 있다. 이 섹션은 모든 컴파일러가 해석할 수 있어야만 하는 패턴 파일에 대한 규칙들을 제시한다. 상기 패턴 파일은 다음과 같이 조직화된다. 즉,

file_contents :

version_info pattern_definitions

version_info :

Version *version-identifier';'*

pattern_definitions :

pattern_definition

pattern_definitions pattern_definition

pattern_definition :

main_header'{'main_section'}'

main_header'{'main_section vendor_sections'}'

subr_header'{'subr_section'}'

subr_header'{'subr_section vendor_sections'}'

main_header :

MainPattern *identifier*

main_section :

CommonSection *'{'common_contents*

main_section_domains'}'

common_contents :

timing_reference timing_map_reference

timing_reference :

Timing *file-name';'*

timing_map_reference :

TimingMap *file-name';'*

main_section_domains :

main_section_domains main_section_domain

main_section_domain

main_section_domain :

Domain *domain_name'{'main_section_contents'}'*

domain_name :

identifier

main_section_contents :

main_section_contents main_section_content

main_section_content

main_section_content :

label_spec main_pattern_spec

main_pattern_spec

label_spec :

label';'

label :

identifier

main_pattern_spec :

main_operation capture_mask_flag'{'

vector_and_waveforms'{'

main_operation :/ empty */*

common_operation

jal_op

jsr_op

jsrc_op

jsc_op

exit_op

common_operation :

idxi_op

idxin_op

jec_op

jech_op

jff_op

jffi_op

jni_op

ldin_op

nop_op

pause_op

sndc_op

sndt_op

stfi_op

sti_op

stps_op

wait_op

*/**

** Instructions specific to the MAIN Patterns*

**/*

jsr_op :

JSR *identifier*

jsrc_op :

JSRC *identifier*

jsc_op :

JSC *identifier*

jal_op :

JAL *identifier*

exit_op :

EXIT

/

** Instructions common to the MAIN and SUBR Patterns*

**/*

idxi_op :

IDXI *24-bit number*

idxin_op :

IDXIn *index-register*

jec_op :

JEC *identifier*

jech_op :

JECH *identifier*

jff_op :

JFF *identifier*

jffi_op :

JFFI *identifier*

jni_op :

JNI *identifier*

ldin_op :

LDIN *index-register*

nop_op :

NOP

pause_op :

PAUSE

sndc_op :

SNDC *8-bit number*

sndt_op :

SNDT *8-bit number*

stfi_op :

STFI *24-bit number*

sti_op :

STI *24-bit number*

stps_op :

STPS

wait_op :

WAIT

capture_mask_flag : / empty */*

capture_mask_flag CTV

capture_mask_flag MTV

capture_mask_flag MATCH

vector_and_waveforms : / empty */*

vector_and_waveforms vector

vector_and_waveforms waveform

vector :

vector_declaration'{'vector_data'}'

vector_declaration :

Vector

V

vector_data :

vector_datum

vector_data vector_datum

vector_datum :

pin_name'='vector-value';'

pin_name'='identifier';'

waveform :

waveform_declaration'{'waveform_data'}'

waveform_declaration :

Waveform

W

waveform_data :

waveform_datum

waveform_data waveform_datum

waveform_datum :

waveform-table-pin-group-name'='identifier';'

pin_name :

identifier

vendor_sections :

vendor_sections vendor_section {}

vendor_section {}

vendor_section :

VendorSection *'{'vendor_section_contents'}'*

subr_header :

SubrPattern

subr_section :

CommonSection *'{'common_contents*

source_selection_table subr_section_domains'}'

CommonSection *'{'common_contents*

subr_section_domains'}'

subr_section_domains :

subr_section_domains subr_section_domain

subr_section_domain

subr_section_domain :

Domain *domain_name'{'subr_section_contents'}'*

source_selection_table :

SourceSelectionTable *'{'source_selector_definitions'}'*

source_selector_definitions:

source_selector_definitions source_selector_definition

source_selector_definition

source_selector_definition :

SourceSelector *source_selector_name'{'*

source_mappings'}'

source_selector_name :

identifier

source_mappings :

source_mappings source_mapping

source_mapping

source_mapping :

pin_name'='source';'

source :

MAIN

INVERT_MAIN

SUBR

INVERT_SUBR

subr_section_contents :

subr_section_contents subr_section_content

subr_section_content

subr_section_content :

label_spec subr_pattern_spec

subr_pattern_spec

subr_pattern_spec :

subr_operation capture_mask_flag'{'

vectors_and_waveforms'}'

*subr_operation :/*empty*/*

common_operation

rtn_op

stss_op

*/**

** SUBR 패턴들에 특유한 지시들*

**/*

rtn_op :

RTN

stss_op :

STSS *identifier*

이하는 상기에 사용된 정의되지 않는 비종단자(non-terminals)의 기술이다. 즉,

1. 버전-식별자(*version-identifier*): 세트 [0-9]로부터의 하나 또는 그 이상의 문자들의 시퀀스로서, 첫번째 문자는 숫자이어야 함.

2. 식별자(*identifier*): 세트 [a-zA-Z_0-9]로부터의 하나 또는 그 이상의 문자들의 시퀀스로서, 첫번째 문자는 세트 [a-zA-Z_]로부터의 문자이어야 함.

3. 벤더-섹션-콘텐츠(*vendor-section-contents*): 벤더 특유적 컴파일러에만 의미를 갖는 임의의 텍스트.

4. 파일-명칭(*file-name*): 유효한 윈도우즈 파일 명칭(파일 명칭 내에 공백이 포함되어 있다면, 반드시 큰 따옴표로 에워싸여야 함.). 이것은 단순한 파일 명칭, 즉 디렉토리 컴포넌트를 가지지 말아야 한다는 점을 주의하여야 함.

5. 파형-테이블-핀-그룹-명칭(*waveform-table-pin-group-name*): 세트 [a-zA-Z_0-9]로부터의 하나 또는 그 이상의 문자들의 시퀀스로서, 첫번째 문자는 세트 [a-zA-Z_]로부터의 문자이어야 함. 이 변수는 다른 곳에서 선언되어, 핀들의 그룹에 공통인 파형-테이블(waveform-table)의 명칭을 보유한다.

6. 24 비트 숫자(*24-bit number*): 최대 16777215까지의 유효 십진수.

7. 8 비트 숫자(*8-bit number*): 최대 256까지의 유효 십진수.

8. 인덱스-레지스터(*index-register*): 유효 십진수. 모듈의 일 실시예에 의 하면, 이는 [1-8]의 값을 가질 수 있다.

9. 벡터(*vector*) : 이것은 STIL의 Vector 문장과 유사하다. 이것은, 상기 컴파일러가 상기 핀 기술 파일로 억세스할 필요가 있도록 만들면서, 신호 명칭 및 신호 그룹의 명칭을 가리킨다.

10. 파형-시간-참조(waveform-time-reference) : 세트 [a-zA-Z_0-9]로부터의 하나 또는 그 이상의 문자들의 시퀀스로서, 첫번째 문자는 세트 [a-zA-Z_]로부터의 문자이어야 함.

패턴 파일들은 코멘트를 지원하는데, 그들은 패턴 파일 컴파일러에 의하여 무시되게 된다. 코멘트는 '#' 문자로 시작되며, 그 줄의 끝까지 연장된다.

상기 패턴 파일의 헤더 및 "공통" 섹션의 구성으로의 참조에 관하여 이하의 점들을 주의하여야 한다.

1. 상기 패턴 명칭 항목은 상기 패턴 파일이 그를 위한 데이터를 수록하는 패턴 객체와 관련된 명칭을 규정한다. 이것은 대응되는 패턴 객체 메타 파일에서 헤더로 넘겨진다.

2. 상기 파형-시간-참조는, 상기 패턴 파일에 대하여 외부적으로 상기 타이밍 파일에서 정의될 특정 파형-및-타이밍 정의에 대한 명칭이다. 상기 패턴 파일의 파형-시간-참조의 규정은, 다른 파형-시간-참조와 마주칠 때까지, (파형-및-타이밍에 대한) 그 특정의 명칭을 모든 이후의 벡터들로 한정시킨다.

3. 서브루틴 호출에 대한 피연산자(operand)는, 같은 패턴 파일에서 이전에 마주쳤던 패턴-스펙 라벨(pattern-spec label)이거나 또는 외부적으로 정의된 서브루틴 패턴의 패턴-스펙 라벨의 어느 하나인 문자열이다. 이 피연산자는 궁극적으로 서브루틴을 로딩/처리하기 위한 목적으로 해석될 것이다. 서브루틴 호출 피연산자에 대한 라벨은 시스템에 걸쳐 고유할 것이 요구된다.

파형-시간-참조의 명칭은 구문적으로 올바르다면 어떤 것이라도 좋지만, 특정한 하드웨어적 관계로 인하여, 상기 파형-시간-참조의 명칭은 이미 알려진 잘 정의된 세트(추가적인 가독성을 위하여, 그것은 사용자에 의하여 사용자가 선택한 명칭으로 선택적으로 매핑될 수 있으며, 상기 매핑은 선택적 파일에 제시될 수 있다.)으로 제한될 필요가 있다.

또한, 상기 패턴 및 파형-시간-참조 소스 파일은, 물리적 테스터 채널에 접속된 모든 피시험 디바이스의 채널에 대한 초기 구성 데이터를 제공하여야 한다는 것을 주의하여야 한다. 만약 어떠한 피시험 디바이스에 대하여 그 후의 데이터가 생략된다면, 상기 패턴 컴파일러는 상기 초기 레벨로부터의 출력을 보유하기 위하여 상기 패턴 데이터를 "패드(pad)"할 것이다.

패턴 파일의 예

MAIN 패턴 소스 파일의 간단한 예를 통하여 그 사용법을 개시한다. 즉,

```
#

# Filename : goodl.pat

#

Version 1.0;
```

```
#-----------------------------------------------------------------
# Main Pattern definition:
#-----------------------------------------------------------------
MainPattern good1
{
CommonSection
{
MacroDef efaultDataVal (XXXXXXXX)
MacroDef nopInstr (NOP)
MacroDef label1 (Label1:)
MacroDef jniInst (JNI)
#-----------------------------------------------------------------
# Timing Specifications
#-----------------------------------------------------------------
Timing "productionTiming.tim";
TimingMap "productionTimingOpenSTARMap.tmap";
#-----------------------------------------------------------------
# Default Domain Cycles
#-----------------------------------------------------------------
Domain default
{
#-----------------------------------------------------------------
#label: instruction {Vector/Waveform Data}
#-----------------------------------------------------------------
NOP {V {DATA = $defaultDataVal; CLK = 1;} W
{DATA = wfs1; CLK = wfs1;}}
JAL myAPG {V {DATA = 00000000;}}
```

```
JSC mySCAN {V {DATA= 10101010;}}

JSRC mySubroutine {V {DATA = 01010101;}}

JSR myAPG {V {DATA = 00110011;}}

STI 100 {}

labZero: NOP {V {DATA = 00000011;}}

JNI labZero {V {DATA = 11111100;}}

IDXI 3000 {V {DATA =10101010;}}

IDXIn 3 {V {DATA=01010101;}}

$label1 NOP {V {DATA = $defaultDataVal;}}

IDXI 2000 {V {DATA= 10101010;}}

NOP {}

EXIT {V {DATA = LLHHLLHH;}}

}

}

}
```

서브루틴 패턴 소스 파일의 다른 예를 이하에 개시한다. 즉,

```
#_____________________________________________________________________

# Subroutine Pattern mySubrPat1 definition:

#---------------------------------------------------------------------

SubrPattern mySubrPat1

{

CommonSection

{

#---------------------------------------------------------------

# Timing Specifications

#---------------------------------------------------------------

Timing "productionTiming.tim";
```

```
TimingMap "productionTimingOpenSTARMap.tmap";
#-----------------------------------------------------------------
# Source Selection Specifications
#-----------------------------------------------------------------
SourceSelectionTable
{
SourceSelector SrcSelDef
{
DATA=SUBR; CLK=SUBR; DATA=SUBR;
}
SourceSelector SrcSelOne
{
DATA=MAIN; CLK=MAIN;
}
}
#-----------------------------------------------------------------
# Default Domain Cycles
#-----------------------------------------------------------------
Domain default
{
#-----------------------------------------------------------------
#label: instruction {Vector and Waveform Data setups}
#-----------------------------------------------------------------
STI 100 {Vector {DATA= 00000000;}}
IDXI 3000 {Vector {DATA = 00001111;}}
IDXIN 3 {Vector {DATA = 00110011;}}
$label1 NOP {Vector {DATA = LLHHLLHH;}}
```

NOP {Vector {DATA = LLXXXXX;}}

NOP {Vector {DATA = LLHHXXX;}}

JNI Label1 {Vector {DATA = LLHHLLHH;}}

STSS SrcSelOne {Vector {DATA= LHLHLHLH;}}

RTN {Vector {DATA = LLXXXXXX;}}

}

}

}

상기 패턴 소스 파일의 메인 헤더 및 공통 섹션으로부터의 요약 정보는 상기 객체 메타 파일의 메인 헤더에 저장된다. 상기 요약은, 어드레스들의 로드 전 해석(pre-load resolution)을 돕거나 데이터 로깅에서 돕기 위하여, 전형적으로 빠른 추출을 요구하는 정보로 구성된다. 상기 공통 섹션의 의미론은 모든 컴파일러에 있어서 정확히 동일하며 모든 컴파일러는 동일한 요약 정보를 제공할 수 있기 때문에, 상기 메타 파일을 작성하는 첫번째 컴파일러가 이 정보를 저장할 것이다. 이하는 저장될 정보이다. 즉,

1. 상기 패턴 소스 파일의 명칭

2. 상기 소스 파일에 선언된 대로의 패턴의 형식

3. 상기 소스 파일로부터의 버전 정보.

4. 상기 패턴 소스 파일의 공통 섹션에 사용된 모든 파형-및-타이밍 명칭들의 리스트.

5. 상기 패턴 소스 파일의 공통 섹션의 (상대적) 벡터 주소에 대한 모든 서브루틴 참조의 지도(map).

6. 상기 패턴 소스 파일의 공통 섹션의 (상대적) 벡터 주소에 대한 모든 라벨 참조의 지도(map).

7. 일반적인 부기 정보(bookkeeping information): 벡터 계수, 지시 계수, 등.

상기 개방형 구조의 테스트 시스템은, 상기 패턴 및 패턴 리스트 파일들이 모두 명시적이고 서로 다른 확장자를 가질 것을 요구한다. 패턴 파일들에 대하여는, 평범한 텍스트 소스 및 컴파일된 객체 파일 모두에 이것이 적용된다. 이것은, 확장자에 기초한 관련을 만들 수 있게 할 뿐만 아니라, 사용자로 하여금 디렉토리 리스트 등에서 파일 형식을 시각적으로 빠르게 식별할 수 있도록 하는 편의성으로 볼 수 있다. 상기 패턴 리스트 파일 분석기는 이하의 확장자들을 가진 파일 명칭들을 기대할 것이다. 즉,

평범한 텍스트 패턴 소스 파일: .pat

컴파일된 패턴 객체 메타 파일: .pobj

패턴 리스트 파일: .plst

상기 사용자는 이들 디폴트 값들을, 예컨대 테스터 환경 변수들 또는 셋업 선택 사양을 통해 무시할 수 있다.

상기 테스터는, 이하에 기술된 적어도 하나의 환경 구성 파일에서 파일 검색 경로들에 대한 다음의 "환경" 변수들의 정의를 요구할 것이다. 즉,

Tester_PATLIST_PATH: 패턴 리스트 파일들에 대하여.

Tester_PATSRC_PATH: 패턴 소스 파일들에 대하여(선택적임).

Tester_PATOBJ_PATH: 패턴 객체 파일들에 대하여.

만약 선택적인 환경/셋업 변수 Tester_PATSRC_PATH가 정의되지 않았다면, 그것은 Tester_PATOBJ_PATH과 동일하다고 가정될 것이라는 점을 주의하여야 한다. 일반적으로, Tester_PATSRC_PATH를 Tester_PATOBJ_PATH과 동일한 값으로 정의하는 것보다 정의하지 않는 것이 보다 효율적이다.

E6. 소프트웨어 표현(Software Representation)

패턴 객체는 사용자에 의하여 창출된다. 오히려, 사용자는 항상 패턴 리스트 객체를 다루며, 그것은 다른 패턴 리스트들 및/또는 패턴들의 모음이다. 패턴 리스트 객체는, 사용자가 억세스할 수 있도록 하면서 그에 수록된 패턴 객체들을 창출하고, 소유하며 유지한다. 상기 사용자의 테스트 프로그램에서의 패턴 리스트 객체는, 디스크 상의 패턴 리스트 파일과 관련되는데, 그것은 상기 패턴 리스트의 실제 정의를 수록하고 있다. 패턴 리스트의 정의는 패턴 리스트를 위한 명시적 명칭을 제공하며, 파일 명칭 관련을 통해 패턴들 및/또는 패턴 리스트들의 정리된 리스트를 식별한다. 이 섹션은, 그들이 상기 테스터 프레임워크에서 조작되는 방법에 대한 도입(prelude)으로서, 패턴 리스트들 및 패턴들의 소프트웨어 표현에 관하여 기술한다.

패턴 리스트 관련들(Pattern List Associations)

상기 테스트 시스템의 하나의 테스트 사이트(그리고 확장하여 그 안의 테스트 계획)는 다수의 최고위 레벨 패턴 리스트(top-level pattern list)들과 관련된다. 그러나, 특정 시간에서 테스트 계획들에 대하여 단 하나의 실행 정황(execution context)만이 존재한다. 최고위 레벨 패턴 리스트가 그에 의하여 (계층적으로) 지칭되는 패턴들에 대한 실행 시퀀스를 정의하므로, 활성화된 실행 정황은 현재 선택된 최고위 레벨 패턴 리스트에 대응되는 것이다. 이것은 하나의 패턴 리스트에 수록된 패턴들만이 특정 시간에 하드웨어 상으로 로드될 수 있다는 것을 암시하는 것이 아니라, 오히려 실행 시퀀스를 실행 가능하게 하기 위하여 상기 하드웨어 상으로 로드되도록 요구된 패턴들의 세트가 언제나 현재 로드된 패턴들의 서브-세트(sub-set)이어야 한다는 것을 주의하여야 한다.

패턴 트리(Pattern Trees)

직관적으로, 최고위 레벨 패턴 리스트를 표현하는 방법은 특정한 종류의 트리 데이터 구조(tree data structure)에 의한다는 것을 느낄 수 있다. 도 7은 본 발명의 일 실시예에 의한 정리된 패턴 트리를 개시하는데, 여기서 상기 패턴 리스트 A는 상기 최고위 레벨 패턴 리스트이다.

패턴 트리 정보 내용(Pattern Tree Information Content)

이하의 정보가 상기 패턴 트리의 모든 노드(node)에 저장될 것이다. 즉,

1. 그 노드와 관련된 엔티티(패턴 리스트 또는 패턴)의 명칭.

2. 상기 정의 소스의 형식. 특정의 리프(패턴 노드)에 있어서, 언제나 패턴 파일이 존재할 것이며; 중간(패턴 리스트) 노드에 있어서, 이것은 "최고위 레벨 파일"(최고위 레벨 패턴 리스트 정의에 대하여) 또는 "파일에 내장된"(함유된 패턴 리스트 정의에 대하여)의 어느 하나일 수 있다.

3. 상기 노드가 관련된 디스크 상의 파일의 마지막 수정의 타임 스탬프.

이하의 추가적인 정보는 중간(패턴 리스트) 노드에만 저장될 것이다. 즉,

1. 그 노드에 의하여 표현된 상기 패턴 리스트 객체의 실행 선택 사항(존재한다면)의 세트 - 즉, 그의 객체 선택 사항.

2. 그 노드에 의하여 표현된 상기 패턴 리스트 정의 내의 각 자손 참조의 실행 선택 사항(존재한다면)의 세트 - 즉, 그의 자손 각각에 대한 참조 선택 사항.

그러므로, 상기 루트로부터 중간 노드로의 고유한 경로 상에서 마주치는 노드들의 모음과, 그들이 마주친 순서는 그 노드에 의하여 표현되는 결합된 유효한 실행 선택 사항들을 결정하기 위하여 필요한 모든 정보를 수록한다. 패턴의 실행 선택 사항은, 그의 직상위 선조(parent)가 가지고 있을 참조 선택 사항과 결합되어 그의 직상위 선조의 유효 실행 선택 사항에 의하여 결정된다.

여기서, 상기 패턴 리스트 분석기가 상기 패턴 트리를 창출하는 과정에 있는 동안, 그들의 사용의 정황이 나중에까지 해석되지 않을 수 있으므로, 소정의 실행 선택 사항들은 단순히 문자열로서 값들을 초기에 저장할 것을 요구할 수도 있다는 점을 주의하여야 한다. 그러한 선택 사항의 예로서, "**마스크(mask)**" 선택 사항이 있을 수 있는데, 그것은 핀 마스크 정보를 규정하며, 패턴 리스트가 소켓 정보와 관련되지 않으며, 따라서 핀 마스크 선택 사항(핀 및 그룹 명칭들)이 로딩 전에 해석되기 위하여 문자열로서 저장된다.

이하의 추가적인 정보들은 리프(패턴) 노드에만 저장될 것이다. 즉,

1. 상기 패턴에 의하여 호출된 서브루틴으로의 모든 외적 및 내적인 (가능하게 과도적인) 참조들로서, 실행 트리로서 조직화된 것.

물론, 모든 패턴 노드들은 추가적으로 상기 객체 메타 파일 공통 헤더에서 활용 가능한 모든 패턴 파일 요약 정보를 억세스할 (또한 캐시하도록 선택할) 것이다.

패턴 리스트 수정의 처리(Handling Pattern List Modifications)

패턴 리스트의 내용에 가하여진 변경은 개념적으로 그 패턴 리스트으로의 모든 참조에 영향을 미친다. 패턴 리스트 객체들뿐만 아니라 적절한 대로 패턴 객체들에도 적용되는 이하의 규칙들이 그러한 변경들을 관리하기 위하여 사용될 것이다.

1. 디스크 상의 패턴 리스트 파일의 내용에 가하여진 변경은, 그 패턴 리스트(또는 그것을 참조하는 임의의 다른 패턴 리스트)에 대하여 실행 중인 load() 명령에 대하여만 상기 테스트 시스템을 통하여 전파될 것이다. 다시 말하면, 소프트웨어에서의 상기 패턴 리스트 계층은 현재 상기 하드웨어에 로드된 것을 항상 반영할 것이다.

2. 상기 사용자는, 패턴 리스트들을 그들의 디스크 파일 소스들과 동기화하기 위하여 로드 시간 동안 만들어진 점검들을 물리치는 모드를 설정할 수 있다. 이것은 생산 모드에서 더욱 빠르고 안전한 동작을 허용한다.

패턴 트리 항해(Pattern Tree Navigation)

테스트 사이트(그리고, 확장하여 그 사이트에 대한 테스트 계획)와 관련된 상기 최고위 레벨 패턴 리스트는, 대중적 (전역적) 범위를 갖는다. 상기 시스템은 최고위 레벨의 패턴 리스트를 나타내는 패턴 트리를 항해하기 위한 API를 제공하여 사용자들이 개별 노드들과 서브 트리(sub-tree)들로 억세스할 수 있게 한다.

E7. 패턴 리스트 역학(Pattern List Dynamics)

패턴 리스트의 정적 규칙에 관하여는 상기한 바 있다. 패턴 리스트들의 동적(실행) 규칙에 관하여 이하에서 기술한다.

상기 패턴 트리는 일반적인 패턴 관리를 위하여 필수적이다. 예를 들어, 패턴 로드 시퀀스에 대한 시작점은, 현재 사이트 또는 테스트 계획과 관련된 패턴 트리로의 상기 load() 메소드의 호출이다. 그러나, 패턴 트리는 고립되어서는 동작하지 않는다. 완전히 초기화된 패턴 트리가 이하의 두 가지 프레임워크 객체를 창출하기 위하여 사용될 것이다. 즉,

1. 최고위 레벨 패턴 리스트는 그 패턴에 대한 **패턴 실행 시퀀스(Pattern Execution Sequence)**를 정의한다. 그것은 그러한 실행 시퀀스가 그 최고위 레벨 패턴 리스트에 대응되는 패턴 트리로부터 도출될 수 있는 방법을 기술한다. 예를 들어, 도 7에 도시된 패턴 트리 A에 대응되는 패턴 실행 시퀀스는 {q, s, t, q, r, q, u, u, v}이다. 상기 패턴 실행 시퀀스는, 개념적으로, 상기 패턴 트리를 통하여 기술된 상기 실행 시퀀스를 반영한 정리된 리스트이다. 상기 프레임워크는 패턴 트리 노드들과 상기 패턴 실행 시퀀스의 대응되는 엔트리들 사이의 임의의 필요한 항해 링크를 수립하고 유지한다.

2. 상기 패턴 트리의 모든 고유한 패턴들(서브 루틴 포함)의 단순한 리스트인 **패턴 세트(Pattern Set)**. 그러므로, 이것은 상기 하드웨어 상으로 로드되어야만 하는 개별 패턴들을 결정하는 데에 사용되는 리스트이다. 상기 프레임워크는 패턴 트리 노드들과 상기 패턴 세트의 대응되는 엔트리들 사이의 임의의 필요한 항해 링크를 수립하고 유지한다. 도 7에 도시된 패턴 트리에 대한 패턴 세트는 (q, s, t, r, u, v)이다(패턴 리스트 A의 패턴들은 어느 것도 서브루틴 호출을 포함하지 않는다고 가정하였다.).

상기 패턴 실행 시퀀스 및 상기 패턴 세트는 항상 상기 패턴 트리로부터 도출되지만, 그러나 최초의 구축 이후에 길어진 것이 실행 가능할 수도 있으므로, 그들을 캐시 처리하는 것도 종종 의미가 있다는 것을 주의하여야 한다.

패턴 리스트 실행 선택 사항(Pattern List Execution Options)

상기한 바와 같이, 각 패턴 리스트 선언(그의 정의에 선행하는) 또는 패턴 리스트/패턴 참조 엔트리 이후에는 복수의 실행 선택 사항(*execution option*)이 올 수 있다. 패턴 리스트 실행 선택 사항은 패턴 리스트들의 실행 시간 실행을 수정한다. 향후의 확장을 허용하기 위하여, 이러한 선택 사항들에 대한 명칭들(그리고 선택적 값들)이, 적합한 특정 버전에 의하여 해석될 수 있도록, 상기 패턴 컴파일러의 패턴 리스트 파일 분석기에 의하여 단순히 문자열로서 취급될 것이다. 테스터는, 이하에 기술되는 바와 같이, 선택 사항들 및 그들의 해석들의 세트를 규정한다. 그러나, 벤더들은 선택 사항의 세트를 확장한다. 선택적 구문의 분석시 유효화를 허용하기 위하여, 상기 패턴 리스트 파일 분석기는 특정 버전을 위한 정보 파일을 읽을 수도 있다. 그러한 정보 파일은 특정 버전이 실행 선택 사항의 규정을 모두 지원하는지의 여부를 규정하기 위하여 사용될 수도 있다.

실행 선택 사항의 세트를 지원하는 버전에 있어서, 이하의 일반적인 규칙들이 그들의 사용을 지배할 것이다. 이들 규칙들을 이해하기 위하여, 패턴 리스트들/패턴들의 계층적 모음을 정리된 트리로서 보는 것이 유용하다.

1. 패턴 리스트 정의들에(즉, 파일에서의 "local-pattern-list-declaration, global-pattern-list-declaration" 생성들에) 설정된 진성 선택 사항들(intrinsic options)의 세트는, 실제로 상기 사용자의 테스트 프로그램의 대응되는 패턴 리스트 객체의 선택 사항 설정을 지휘한다. 그러므로, 그들은 그 패턴 리스트 객체로의 모든 참조에 적용되며, 객체 선택 사항(*object options*)이라 불린다.

2. 패턴 리스트들/패턴들에 대한 참조에서(즉, 파일에서의 "pattern-entry" 및 "pattern-list-entry" 생성들에) 설정된 참조적 선택 사항들(referential options)의 세트는 계층 내에서의 특정 경로(패턴 리스트들/패턴들의 선언 순서에 의하여 수립되는 것으로서 상기 트리의 루트로부터 고려중인 참조로 이끄는 경로)에 대한 상기 선택 사항의 범위를 제한한다. 그러므로, 이들은 특정의 객체 참조 상의(그리고 객체들 자체 상이 아닌) 선택 사항이며, 참조 선택 사항(*reference option*)이라 불린다.

3. 모음 계층에서의 임의의 리스트/패턴에 대한 효과적 선택 사항 설정(*effecitve* option setting)은, 상기 트리의 루트로부터 그 리스트/패턴으로의 경로를 따라 마주치는 상기 객체 및 참조 선택 사항들의 조합이다. 상기 특정 조합 메커니즘(예를 들어, 합집합, 교집합 또는 다른 임의의 불일치 해결 알고리즘)이 상기 선택 사항 자체의 속성이다.

상기 규칙들의 결론-그리고, 패턴 파일에서 패턴 정의에 대한 실행 선택 사항을 설정할 설비가 없다는 사실-은 패턴으로의 모든 참조에 적용되는 선택 사항을 설정할 직접적인 규칙이 없다는 것이다. 이것을 성취하기 위한 메커니즘은 단일-패턴의 패턴 리스트를 사용하는 것이다.

상기 테스터는, 그의 버스트 동작(burst behavior)을 수정할, 그리고 그의 실행 시퀀스를 수정할 패턴 리스트 실행 선택 사항의 특정 세트를 규정한다.

패턴 리스트에 대한 실행 시퀀스가 상기 하드웨어에 제출될 때, 상기 하드웨어는 버스트(burst)를 생산한다. 버스트는, 소프트웨어의 개입 없이, 직접 상기 하드웨어에 의한 패턴의 시퀀스의 실행이다. 버스트 불연속(burst discontinuity)은, 이전의 버스트가 종료되고 새로운 버스트가 시작되는 실행 시퀀스 내의 장소이다.

패턴 관리 소프트웨어의 목적의 하나는 상기 하드웨어에게 그것이 버스트를 생산해야 할 필요가 있는 실행 시퀀스를 제공하는 것이다. 디폴트로서, 패턴 트리는 실행 시퀀스를 산출하고, 그것은 만약 하드웨어에 제출되면 하나의 버스트를 야기할 것이다. 이 동작은, 그러나, 상기 패턴 리스트에 선택 사항을 사용함으로써 수정될 수 있다. 그러므로, 선택 사항의 사용은 버스트 불연속을 야기할 수 있다.

나아가, 사용자들은 때때로 모든 패턴 또는 모든 버스트의 앞 또는 뒤에 프롤로그 또는 에필로그 패턴이 실행되도록 요구할 것이다. 이것은 하드웨어에 제출될 실행 시퀀스를 수정한다.

상기 패턴 실행 시퀀스 객체를 창출 또는 수정하는 동안, 상기 프레임워크는, 규정된 실행 선택 사항과 상기 패턴 트리에 의하여 구현된 특정의 실행 시퀀스의 조합으로부터 야기되는 패턴 버스트의 중단을 판단하고 그리고 필요하다면 보고하기 위하여 필요한 모든 정보를 가진다. 이것을 수행하는 동안, 상기 시스템의 상기 모듈들의 하드웨어적 능력을 조사할 필요가 있을 수 있다. 예를 들어, 하나의 하드웨어 구현이 핀 마스크들에 대한 네 가지의 저장된 구성을 허용할 수 있는데, 그들 중 두 가지(0 및 3)은 디폴트 마스크된 동작("Mask This Vector(MTV)"를 지원하기 위한) 및 마스크되지 않은 동작에 사용된다. 그러므로, 사용자에게는 버스트 모드를 중단시키지 않고서 두 가지 서로 다른 전역 핀 마스크 구성이 허용된다.

만약 특정의 모듈 벤더가 하드웨어의 패턴 리스트 구현을 지원하지 않는다면, 상기 벤더의 상기 패턴 실행 시퀀스의 처리는 상기 실행 시퀀스의 모든 패턴의 개별적인 실행을 야기할 것이라는 점에 유의한다. 사이트-호환 가능(Site-Compatible) 그리고 사이트-이질적(Site-Heterogeneous) 시스템 모두에 있어서, 상기 사이트들의 버스트 능력은 "최하위 공통 분모(lowest common denominator)"에 의하여 제한될 것이다. 상기 테스터는 소정의 선택 사항의 디폴트 세트를 제공하며, 그들의 파라미터는 이하에서 기술될 것이다. 각 선택 사항은 다음 내용에 의하여 규정된다. 즉,

그것이 진성(Intrinsic)(즉, 전역 또는 지역 키워드를 가진 정의와 관련됨) 또는 참조적(Referential)(즉, Pat 또는 PList 키워드를 가진 참조와 관련됨)인지의 여부. 진성 선택 사항은 정의의 지점 및 모든 참조에서 적용되지만, 그러나 참조적 선택 사항은 그들이 관련된 참조에서만 적용된다.

게다가, 만약 상기 선택 사항이 모든 정적으로(구문적으로) 또는 동적으로(참조됨으로써 의미론적으로) 함유된 패턴들 또는 패턴 리스트들에 재귀적으로 적용된다고 가정된다면, 상기 선택 사항은 그 자손에 의하여 상속되었다고 한다.

이하는 선택 사항들의 리스트이다. 모든 협조적인 벤더들은 이들 선택 사항들을 규정된 대로 해석할 것이다. 즉,

1. 마스크(Mask) <핀/ 핀 그룹>

GlobalPList, LocalPList에 적용될 때 진성

PList, Pat에 적용될 때 참조적

자손에 의하여 상속됨

이 패턴 리스트는, 항상, 지시된 활용 불가능한 핀 또는 핀 그룹에 의하여 참조된 핀들의 비교 회로를 갖는다. 때때로, 하드웨어적 제한은 버스트 불연속을 야기할 수 있다.

2. 버스트오프(BurstOff)

GlobalPList, LocalPList에 적용될 때 진성

PList, Pat에 적용될 때 참조적

자손에 의하여 상속되지 않음

이 패턴 리스트는 항상 비 버스트 모드에서 실행할 것이다. 이 선택 사항은 자손에 의하여 상속되지 않으나, 버스트오프딥(BurstOffDeep) 선택 사항(후술)은 자손에 의하여 상속된다.

3. 버스트오프딥(BurstOffDeep)

GlobalPList, LocalPList에 적용될 때 진성

PList, Pat에 적용될 때 참조적

자손에 의하여 상속됨

이 패턴 리스트는 항상 비 버스트 모드에서 실행할 것이다. 이 선택 사항은 자손에 의하여 상속되나, 상기 버스트오프 선택 사항(상기)은 자손에 의하여 상속되지 않는다. 상기 버스트오프딥 선택 사항은 자손에 의하여 중지될 수 없음을 주의하여야 한다.

4. 프리버스트(PreBurst) <패턴>

GlobalPList, LocalPList에 적용될 때 진성

버스트 선택 사항이 규정되지 않은 자손 노드에 의하여만 상속됨

지시된 패턴은 이 패턴 리스트 내의 모든 버스트들의 앞에 붙는다. 상기 프리버스트 패턴은 이 패턴 리스트 노드에 의하여 시작된 모든 버스트 직전에 발생한다. 이 선택 사항은, 동일한 패턴인 프리버스트 선택 사항을 가진 버스트 내에 이미 존재할 때에는 적용되지 않는다.

5. 포스트버스트(PostBurst) <패턴>

GlobalPList, LocalPList에 적용될 때 진성

버스트 선택 사항이 규정되지 않은 자손 노드에 의하여만 상속됨

지시된 패턴은 이 패턴 리스트 내의 모든 버스트들의 뒤에 붙는다. 상기 프리버스트 패턴은 이 패턴 리스트 노드에 의하여 시작된 모든 버스트 직후에 발생한다. 이 선택 사항은, 동일한 패턴인 프리버스트 선택 사항을 가진 버스트 내에 이미 존재할 때에는 적용되지 않는다.

6. 프리패턴(PrePattern) <패턴>

GlobalPList, LocalPList에 적용될 때 진성

자손에 의하여 상속되지 않음

지시된 패턴은 이 패턴 리스트 내의 모든 패턴들의 앞에 붙는다.

7. 포스트패턴(PostPattern) <패턴>

GlobalPList, LocalPList에 적용될 때 진성

자손에 의하여 상속되지 않음

지시된 패턴은 이 패턴 리스트 내의 모든 패턴들의 뒤에 붙는다.

8. Alpg <alpg 객체 명칭>

GlobalPList, LocalPList에 적용될 때 진성

자손에 의하여 상속되지 않음

명명된 ALPG 객체는, 저속 APG 레지스터 설정, 독출 레이턴시(Read latency), 즉각적인 데이터 레지스터들, 어드레스 스크램블(address scramble), 데이터 반전(data inversion), 데이터 생성기들, 등과 같은 관련 정보를 저장한다.

9. 시작 패턴(StartPattern) <패턴>

GlobalPList, LocalPList에 적용될 때 진성

자손에 의하여 상속되지 않음

상기 패턴 리스트는 그의 실행 시퀀스에서 상기 시작 패턴의 첫번째 발생에서 실행을 시작할 것이다.

10. 정지 패턴(StopPattern) <패턴>

GlobalPList, LocalPList에 적용될 때 진성

자손에 의하여 상속되지 않음

상기 패턴 리스트는 그의 실행 시퀀스에서 상기 정지 패턴의 첫번째 발생에서 실행을 정지할 것이다.

11. 시작 어드레스(StartAddr) <벡터 오프셋 또는 라벨>

GlobalPList, LocalPList에 적용될 때 진성

자손에 의하여 상속되지 않음

이것은 시작 패턴 선택 사항에 의하여 달성되어야 한다. 상기 패턴 리스트는 그의 실행 시퀀스에서 상기 시작 패턴의 첫번째 발생의 시작 어드레스에서 실행을 시작할 것이다.

12. 정지 어드레스(StopAddr) <벡터 오프셋 또는 라벨>

GlobalPList, LocalPList에 적용될 때 진성

자손에 의하여 상속되지 않음

이것은 정지 패턴 선택 사항에 의하여 달성되어야 한다. 상기 패턴 리스트는 그의 실행 시퀀스에서 상기 정지 패턴의 첫번째 발생의 정지 어드레스에서 실행을 정지할 것이다.

13. 인에이블비교_시작패턴(EnableCompare_StartPattern) <패턴>

GlobalPList, LocalPList에 적용될 때 진성

자손에 의하여 상속되지 않음

지시된 패턴의 첫번째 발생에서 패턴 비교가 개시될 것이다.

14. 인에이블비교_시작어드레스(EnableCompare_StartAddr), 인에이블비교_시작주기(EnableCompare_StartCycle)

GlobalPList, LocalPList에 적용될 때 진성

자손에 의하여 상속되지 않음

이것은 인에이블비교_시작패턴과 동반되어야 한다. 상기 패턴 비교가 시작되는 패턴 내의 어드레스 또는 주기를 지시한다.

15. 인에이블비교_정지패턴(EnableCompare_StopPattern) <패턴>

GlobalPList, LocalPList에 적용될 때 진성

자손에 의하여 상속되지 않음

지시된 패턴의 첫번째 발생에서 패턴 비교가 완료될 것이다.

16.기동비교_정지어드레스(EnableCompare_StopAddr), 인에이블비교_정지주기(EnableCompare_StopCycle)

GlobalPList, LocalPList에 적용될 때 진성

자손에 의하여 상속되지 않음

이것은 인에이블비교_정지패턴과 동반되어야 한다. 상기 패턴 비교가 완료되는 패턴 내의 어드레스 또는 주기를 지시한다.

17. 건너뛰기(Skip)

PList, Pat에 적용될 때 참조적

자손에 의하여 상속되지 않음

패턴 리스트에 의하여 지배되던 패턴 또는 전체 서브시퀀스(subsequence)를 건너뛰게한다. 이것은 또한 이 패턴 리스트의 서브 트리에서 모든 선택 사항들의 건너 뛰기도 또한 야기할 것이다. 그것은 실행의 의도에 대하여 마치 이 패턴의 서브 트리가 존재하지 않았던 것과 같다.

패턴 리스트 버스트 제어(Pattern List Burst Control)

상기한 바와 같이, 패턴 리스트에 대한 실행 시퀀스가 상기 하드웨어에 제출되면, 상기 하드웨어는 소프트웨어로부터의 어떠한 관여도 없이 패턴의 시퀀스의 버스트를 생산한다. 버스트 불연속은, 이전의 버스트가 종료되고 새로운 버스트가 시작되는 실행 시퀀스의 위치이다. 상기 프리버스트, 포스트버스트, 버스트오프및 버스트오프딥의 선택 사항들은, 상기한 선택 사항 리스트에서 기술한 바와 같이, 상기 버스트 불연속이 발생하는 지점을 제어한다. 프리버스트 및 포스트버스트 선택 사항은 이하에 기술된 특정의 추가적인 규칙들에 따라 버스트 불연속을 결정한다. 즉,

1. 선조 리스트가 프리버스트 및 포스트버스트 선택 사항을 가지며, 함유된 리스트가 동일한 대응되는 선택 사항을 가지고 있을 때에는 버스트 불연속이 존재하지 않으며, 상기 함유된 리스트의 프리버스트 및 포스트버스트 선택 사항이 적용되지 않는다. 단 하나의 버스트만이 상기 선조 리스트의 프리버스트 및 포스트버스트에 적용된다.

2. 함유된 리스트가 버스트 선택 사항을 가지고 있지 않을 때에는, 이들 선택 사항의 기술에 의하여, 선조 리스트과 동일한 프리버스트 및 포스트버스트 선택 사항을 가지는 것과 마찬가지다. 결과적으로, 버스트 선택 사항을 가지지 않는 함유된 리스트는 버스트 불연속을 야기한다.

3. 상기의 규칙 1이 적용되지 않고, 상기 패턴 리스트의 시작으로부터 상기 함유된 리스트의 시작까지 상기 패턴 실행 시퀀스에의 기여가 존재한다면, 상기 함유된 리스트의 시작에서 버스트 불연속이 존재한다. 이 경우, 상기 선조 리스트의 프리버스트 및 포스트버스트는, 상기 선조 리스트으로부터의 패턴 실행 시퀀스에 대한 이 기여에 적용된다. 상기 함유된 리스트의 프리버스트 및 포스트버스트는 상기 함유된 리스트에 적용된다.

4. 만약 상기 규칙 1이 적용되지 않고, 상기 패턴 리스트의 끝으로부터 상기 함유된 리스트의 끝까지 상기 패턴 실행 시퀀스에의 기여가 존재한다면, 상기 함유된 리스트의 끝에서 버스트 불연속이 존재한다. 이 경우, 상기 선조 리스트의 프리버스트 및 포스트버스트는, 상기 선조 리스트으로부터의 패턴 실행 시퀀스에의 이 기여에 적용된다. 상기 함유된 리스트의 프리버스트 및 포스트버스트는 상기 함유된 리스트에 적용된다.

5. 만약 상기 규칙 1이 적용되지 않고, 상기 패턴 리스트의 끝으로부터 상기 함유된 리스트의 끝까지 상기 패턴 실행 시퀀스에의 기여가 존재하지 않는다면, 상기 선조 리스트의 프리버스트 및 포스트버스트는 적용되지 않는다. 상기 함유된 리스트의 프리버스트 및 포스트버스트에 적용되는 단 하나의 버스트가 존재한다.

이하는, 상기 실행 시퀀스의 선택 사항의 영향을 기술하는 몇가지 예시이다. 간단히 하기 위하여, 모든 패턴 리스트가 하나의 파일에 규정되어 있다고 가정하였다. 즉,

예1: 버스트오프의 사용

이 예는 버스트오프 및 프리버스트를 예시한다. 버스트오프는, 패턴들이 하나의 패턴 길이의 버스트 내에서 단독적으로 실행되도록 한다는 점을 강조한다. 그러므로, 프리버스트 선택 사항이 여전히 적용된다. 입력 패턴 리스트들은 다음과 같다. 즉,

```
Global A [BurstOff][PreBurst pat_z]

{

Pat q;

PList B;

Pat r;

Pat s;

Global C

{

Pat t;

PList D;

};

PList D;

PList E;

};

Global B

{

Pat a;

Pat b;

};

Global D [BurstOff]

{

Pat c;
```

Pat d;

};

Global E

{

Pat e;

};

A에서 루트를 갖는 트리는 도 8에 도시될 수 있다.

이 패턴의 실행 시퀀스는 이하와 같다. '|' 문자는 버스트 중단(burst break)를 지시한다. 이 패턴 리스트는 10개의 버스트에서 실행되며, 첫번째 것은 패턴 z 및 q를가지며, 마지막 것은 패턴 c를 갖는다. 즉,

z q | a b | z r | z s | t | c | d | c | d | e

이 실행 시퀀스에 관하여 다음을 주의하여야 한다. 즉,

1. A에서의 버스트오프 선택 사항은 B에 의하여 상속되지 않기 때문에, B의 패턴 a 및 b는 버스트로서 동작한다.

2. A에서의 프리버스트 선택 사항은 B에 의하여 상속되지 않기 때문에, B에의한 버스트의 a 및 b는 z에 의하여 앞에 붙지 않는다.

3. z에 의한 접두사의 붙임은, a의 직계 자손인 q, r 및 s임으로 인하여 실행되는 패턴들에 대하여만 발생한다. 이들 패턴들은, A가 버스트오프 선택 사항을 가짐에 의하여, 마치 단 하나의 패턴 길이인 버스트 내에 있는 것처럼 단독적으로 실행된다. 상기 버스트오프는 패턴들이 하나의 패턴 길이의 버스트 내에서 개별적으로 실행될 것을 요구한다. 그러므로, 프리버스트 및 포스트버스트 선택 사항은 여전히 적용된다.

4. 패턴 리스트 D는, 자손 c 및 d가 단독적으로 실행되도록 하는 진성 버스트오프 선택 사항을 갖는다. 그들은 A로부터 프리버스트 z를 상속하지 않는다.

예2: 버스트오프딥의 사용

이 예는 상기 버스트오프딥 선택 사항을 예시한다. 패턴 리스트 정의 동안에 버스트오프딥은 함유된 정의 및 참조된 리스트에 영향을 미친다. 그러나, 프리버스트 및 포스트버스트 선택 사항들은 함유된 그리고 참조된 리스트에 의하여 상속되지 않는다. 이 예에서는 예 1에서와 같은 패턴들 A, B, C, D, E를 사용하지만, 선택 사항들은 서로 다르다. 즉,

5. A의 정의 상에서의 선택 사항들: [BurstOffDeep], [PreBurst z], [PostBurst y]

6. 다른 노드 어디에도 다른 선택 사항이 없음.

실행 시퀀스는 이하와 같다. 앞서와 마찬가지로, '|' 문자는 버스트 중단을 지시한다. 즉,

z q y | a | b | z r y | z s y | t | c | d | c | d | e

이 실행 시퀀스에 관하여는 다음을 주의하여야 한다. 즉,

1. 프리버스트 및 포스트버스트는 B, C, D, E에 의하여 상속되지 않는다.

2. 버스트오프딥은 B, C, D 및 E에 의하여 상속된다.

예 3. 프리버스트 및 포스트버스트 금지(Preburst and PostBurst Inhibition)

이제, 예 1의 패턴 리스트 트리에 있어서, 다음의 선택 사항이 존재한다고 하자. 즉,

1. A의 정의 상의 선택 사항들: [PreBurst z][PostBurst y]

2. C의 정의 상의 선택 사항들: [PreBurst z][PostBurst z]

3. 다른 노드 어디에도 다른 선택 사항이 없음.

실행 시퀀스는 다음과 같다. 즉,

x q a b r s t c d c d e y

상기 "t c d" 서브시퀀스가 "x t c d z"가 아닌 이유는 다음과 같다. 즉,

1. 첫번째 x는, 그것이 현재 유효한 버스트와 관련된 프리버스트 선택 사항 x와 같기 때문에, 금지되었다.

2. 마지막 z는, 포스트버스트 z가 D로 상속되지 않으며, z가 첨부될 C로부터 어떠한 패턴도 생성되지 않았기 때문에, 금지되었다.

예 4. 건너뛰기의 사용(Use of Skip)

이 예는 함유된 정의들 및 참조된 리스트들 상의 건너뛰기 선택 사항의 영향을 예시한다. 이 예에서는 예 1에서와 같은 패턴들 A, B, C, D, E가 사용되지만, 그 선택 사항들은 서로 다르다. 즉,

1. A의 정의 상의 선택 사항: [Skip], [PreBurst z], [PostBurst y]

2. r로의 참조 상의 선택 사항: [Skip]

3. C의 정의 상의 선택 사항: [Skip]

실행 시퀀스는 중단 없는 단일의 버스트로서 이하와 같다. 즉,

z q a b s c d e y

이 실행 시퀀스에 관하여 다음을 주의하여야 한다. 즉,

1. r 및 C에 대한 노드는 건너뛰었다.

2. 어떠한 버스트 중단도 없다.

예 5. 마스크의 사용(Use of Mask)

이 예는 마스크 선택 사항의 영향을 예시하며, 그의 패턴 및 패턴 리스트 정의와 참조에의 영향을 예시한다. 이 예에서는 예 1에서와 같은 패턴들 A, B, C, D, E가 사용되지만, 그 선택 사항들은 서로 다르다. 즉,

1. A의 정의 상의 선택 사항: [mask pin1_pin2], [PreBurst z]

2. B의 정의 상의 선택 사항: [mask pin3]

3. C의 정의 상의 선택 사항: [mask pin4]

4. e의 참조 상의 선택 사항: [mask pin5]

5. 다른 노드 어디에도 다른 선택 사항이 없음.

상기 명칭들 "pin1_pin2"는 Pin1 및 Pin2를 마스크하는 그룹을 규정한다. 명칭들 "pin3", "pin4", 및 "pin5"는 각각 Pin3, Pin4 및 Pin5의 마스킹을 규정한다. 그 실행 시퀀스는 버스트 중단을 지시하는 '|'과 함께 이하에 제공되었다. 이하의 각 패턴의 숫자들은 패턴 실행 도중에 마스크되는 핀들을 지시한다. 즉,

z q a b z r z s t c d c d | e

1 1 1 1 1 1 1 1 1 1 1 1 1 1

2 2 2 2 2 2 2 2 2 2 2 2 2 2

3 3 5

4 4

이 실행 시퀀스에 관하여는 다음을 주의하여야 한다. 즉,

1. 상기 벤더의 하드웨어는 버스트 중단 없이 단 2 개의 마스크 블록만을 제공한다. e가 실행될 때까지의 두 개의 마스크 블록들은 핀들 {1, 2} 및 핀들 {1, 2, 3, 4}이다. 핀들 {1, 2, 5}의 마스크 블록을 가진 패턴 e가 도달하면, 상기 하드웨어는 버스트 중단을 필요로 한다.

예 6. 상속된 선택 사항 및 참조들의 사용(Use of Inherited Options and References)

이 예는, 정의시의 상속된 선택 사항은 상기 정의가 참조될 때에는 적용되지 않는다는 것을 예시한다. 이하의 예를 생각해 보자. 즉,

```
Global A

{

Global B[BurstOffDeep]

{

Global C

{

...

};

...

};

...

PList C;
```

};

Global D

{

PList C;

};

상기 버스트오프딥 선택 사항은 그의 정의 지점에서 C에 의하여 상속된다. 그러나, 그것은 진성 선택 사항이 아니며, 따라서 그것은 그의 참조 위치 모두에서 C에 적용되지 않는다.

예 7. 함유된 리스트를 가진 프리버스트 및 포스트버스트(PreBurst and PostBurst with Nested Lists)

이하의 예를 생각해 보자. 즉,

GlobalPList A [PreBurst x][PostBurst y]

{

Pat p1;

LocalPList B [PreBurst x][PostBurst y]

{

Pat p2;

}

LocalPList C

{

Pat p3;

}

LocalPList D [PreBurst x][PostBurst z]

{

Pat p4;

}

LocalPList E [PreBurst w][PostBurst y]

{

Pat p5;

}

Pat p6;

}

그 실행 시퀀스는 다음과 같다. 즉,

x p1 p2 p3 y | x p4 z | w p5 y | x p6 y 1

1. 상기 함유된 리스트의 상기 프리버스트 및 포스트버스트 선택 사항은 선조와 동일하게 규정되었으므로, 패턴 p2는 p1과 동일한 버스트에 존재한다. 패턴 p3는, 이들 선택 사항들이 선조와 동일하게 상속되었으므로, 역시 같은 버스트에 존재한다. 이들 선택 사항들은, 버스트 불연속을 주면서, 남은 함유된 리스트들에서 적어도 하나의 서로 다른 멤버를 갖는다.

타이밍(Timing)

사용자는 패턴 파일을 사용하면서 상기 시험 셋업을 정의함으로써, 상기 시스템과 주로 상호 작용을 한다. 상기 타이밍 파일은 이들 패턴들의 타이밍을 기술하기 위하여 사용된다. 이 파일은 내재된 정의가 해석될 수 있도록 다른 시스템 파일(예를 들어, 핀(*Pin*), 규정선택자(*SpecSelector*))를 필요로 한다. 나아가, 상기 타이밍 정의에 사용된 다양한 변수들을 해석하기 위하여 사용되는 상기 규정 선택자 및 전역적 정의들은, 혼성된 테스트 조건 그룹 객체로 캡슐화 된다. 테스트 계획 파일과 같은 더 높은 레벨의 파일들은 차례로 이 테스트 조건 그룹 인스턴스를 사용한다.

상기 테스트 계획 파일은 상기 테스트 조건 그룹 객체로의 참조를 포함한다. 상기 패턴 소스 파일은 타이밍 맵(*TimingMap*) 객체 내의 파형 선택자 컴포넌트로의 참조를 만든다. 상기 타이밍 객체 자체는 상기 핀 객체를 참조한다. 선택적으로, 상기 타이밍 객체는 또한 규정 선택자 객체에 의하여 수정된 변수들을 참조할 수도 있다. 이들 관계들은 도 9에 도시되었다.

상기 패턴 리스트 내의 패턴 객체들은 패턴 문자들의 세트에 대하여 사용하기 위한 상기 파형 선택자 객체의 명칭을 규정한다. 또한, 상기 타이밍 맵 파일은 상기 패턴에 규정된다는 것을 주의하여야 한다. 패턴들은 이 맵이 변경되지 않으면 컴파일 될 필요가 없다.

Version 1.0;

MainPattern

{

CommonSection

{

...

Timing = myGalxy.tim;

TimingMap = myGalxyMap.tmap;

...

Domain default

{

NOP V {SIG=1; CLK=1; DATA=L;} W {SIG=wfs1;

FASTCLK=wfs1;}

NOP W {SIG=wfs2;}

NOP V {SIG=L;}

NOP V {SIG=0;}

}

}

}

상기의 SIG에서, 우리는 패턴 캐릭터를 사용하기 위해 타이밍 맵의 파형 선택자 컴포넌트를 정의한다.

상기 테스트 조건 그룹 파일 객체는 사용할 타이밍 객체와 사용할 타이밍 맵 객체를 인입한다. 각 시험은, 그 인스턴스에 대한 상기 테스트 조건 그룹 객체로부터 도출된 타이밍 조건 인스턴스를 사용한다. 그러므로, 같은 세트의 파형 테이블을 지원하는 다수의 타이밍 객체들은 상기 테스터 프레임워크에 저장될 수 있으며, 요구되는 대로 교환될 수 있다. 이와 유사하게, 다수의 테스트 계획 파일은 공통의 테스트 조건 그룹 객체를 공유할 수 있다.

테스트 계획 기술 파일의 예를 통하여 타이밍 객체의 사용을 이하에서 예시한다. 즉,

Import patlist1.plist;

Import tim1.tim;

Import tim2.tim;

Import tmap1.tmap;

TestConditionGroup tim1_prod

{

SpecSet = prodTmgSpec(min, max, typ)

{

period = 10ns, 15ns, 12ns;

}

Timings

{

Timing = tim1;

TimingMap = tmap1;

```
}
}
TestConditionGroup tim2_prod
{
SpecSet = prodTmgSpec(min, max, typ)
{
period = 10ns, 15ns, 12ns;
}
Timings
{
Timing = tim2;
TimingMap = tmap1;
}
}
TestCondition tim1_prod_typ
{
TestConditionGroup = tim1_prod;
Selector = typ;
}
TestCondition tim2_prod_max
{
TestConditionGroup = tim2_prod;
Selector = max;
}
Test FunctionalTest MyFunctionalTestSlow
{
PListParam =patlist1;
```

TestConditionParam = tim1_prod_typ;

}

Test FunctionalTest MyFunctionalTestFast

{

PListParam = patList1;

TestConditionParam = tim2_prod_max;

}

상기 tim1 및 tim2는 앞서 정의된 서로 다른 타이밍 객체들을 사용하는 테스트 계획 내의 두 가지 시험들이다.

상기 타이밍(Timing) 객체는 각 핀 기준으로 다양한 파형들을 정의한다. 상기 타이밍 파일 및 타이밍 맵 파일에서 사용된 핀들은 상기 핀 기술 파일에서 적절히 정의되어 있을 필요가 있다.

상기 타이밍 객체는 상기 파형(Waveform) 객체 내의 값들을 정의하기 위하여 규정 세트(SpecificationSet) 객체를 사용할 수 있다. 상기 타이밍 객체가 다양한 속성들에 대한 고정적으로 코딩된 값들(hard-coded values)을 포함할 수 있지만, 사용자들은 다양한 속성들이 변수들을 사용하여 값이 할당되도록 하는 것이 통상적인 경우이다. 이들 값들은 차례로 규정 세트 객체에 의존할 수 있다. 이 사용의 예를 이하에서 개시한다. 즉,

Version 1.0;

Timing basic_functional

{

...

Pin SIG

{

WaveformTable wfs1

{

{1{U@t_le; D@t_te D; Z@45ns;}}

};

};

Pin CLK

{

WaveformTable wfs1

{

{0 {U@20ns; D@40ns;}};

};

};

}

에지의 배치를 정의하는 U@t_le 변수는 다른 어딘가에서 정의되며 규정 세트에 의존한다.

상기 규정 세트는 다음과 같이 정의된다. 즉,

SpecificationSet prodTmgSpec(min, max, typ)

{

t_le = 10ns, 14ns, 12ns;

t_te = 30ns, 34ns, 32ns;

...

}

상기 규정을 변경함으로써 사용되는 타이밍의 변경은 다음 예에서 설명된다. 즉,

TestCondition prodTmp_typ

{

TestConditionGroup = prodTmgSpec;

SpecSelector = typ;

}

TestConditionGroup prodTmp_max

{

TestConditionGroup = prodTmgSpec;

SpecSelector = max;

};

상기 typ 타이밍은 상기 규정 선택자의 전형적인 규정을 사용한다. 상기 max 타이밍은 규정 선택자에서 최대(max) 규정을 사용한다.

F2. 테스터의 타이밍 컴포넌트로의 매핑(Mapping to the Timing Components of a tester)

테스터 모듈의 두 가지 컴포넌트는 파형의 생성 및 그들의 관련 타이밍과 직접적으로 연관된다. 상기 두 가지 모듈들은 패턴 생성기(Pattern Generator; PG) 및 프레임 프로세서(Frame Processor; FP)이다. 상기 개방형 구조의 테스트 시스템에서 프레임 프로세서에 의한 파형 정형(wave shape formatting)과 타이밍 생성을 예시하는 간단한 블록도를 도 10에 도시하였다. 파형의 생성에 관한 간략한 설명을 이하에 기술한다.

상기 패턴 생성기 1002는 상기 모듈의 모든 핀들에 대하여 공통인 타이밍 세트를 생성한다. 이 타이밍 집합은 전역 타이밍 세트(Global Timing Set; GTS)라 불린다. 상기 패턴 생성기가 셋업될 수 있는 세 가지 모드가 존재한다. 이들 세 가지 모드는 상기 GTS를 기술하는 데에 사용될 수 있는 비트의 개수에 영향을 준다. 나아가, 이들 설정들은 뱅크(bank) 및, "캡쳐 디스 벡터(Capture This Vector: CTV)"와 "마스크 디스 벡터(Mask This Vector; MTV)의 비트들이 설정되었는지 아닌지의 여부를 선택하는 데에 사용되는 비트들의 개수에도 또한 영향을 미친다. 상기 테스터로 하여금 이 벡터의 결과를 포착(capture)하도록 지시하기 위하여, 상기 사용자는 상기 패턴 파일에서 상기 CTV 플래그(flag)를 사용한다. 이와 유사하게, 상기 사용자는 현재 벡터의 결과를 마스크하도록 상기 테스터에게 지시하기 위하여 상기 MTV 플래그를 사용한다. 이것을 이하의 표에 예시하였다. 상기 패턴 생성기 1002는 또한 파형 문자들(Waveform Characters; WFC)를 생성할 책임도 갖는다. WFC는 핀 단위로 생성된다. 상기 테스터 모듈은 상기 WFC를 기술하기 위하여 고정된 개수의 비트들을 사용한다.

[표 1]

GTS 비트들	뱅크의 GTS	GTS 뱅크	CTV	MTV
8비트	256	4	아니오	아니오
7비트	128	8	예	아니오
6비트	64	16	예	예

상기 테스터 모듈은 핀마다 프레임 프로세서 1004를 제공한다. 각 프레임 프로세서는 타이밍 세트 스크램블러(Timing Set Scrambler; TSS) 1006을 포함하는데, 그것은, 이 실시예에 의하면, 전체 1024까지의 깊이를 갖는다. 상기 TSS는 도 10에 도시하여 상기한 바와 같이 패턴 생성기의 모드에 따라 복수의 뱅크들로 구획될 수 있는데, 도 10에서는 뱅크당 64개의 엔트리를 갖는 16개의 뱅크가 사용되고 있다. 상기 TSS는 각 핀에 대하여 파형 테이블을 정의할 수 있는 능력에 더욱 큰 유연성을 허용하기 위하여 제공된다. "FP" 모드에서, 상기 TSS는 2 비트를 사용하는 타이밍 세트를 출력한다. 그러므로, 상기 TSS는 핀 당 네 개의 서로 다른 물리적 타이밍 세트들을 생성할 것이다. 이들 타이밍 집합들은 지역 타이밍 세트(Local Timing Set; LTS)라 불린다.

상기 프레임 프로세서 1004는 LTS와 WFC를 조합하여, 파형 메모리 1012 및 타이밍 메모리 1014에 인덱스 1010를 창출한다. 상기 "FP" 모드에서, 상기 5 비트의 값은, 상기 LTS에서 생성된 2 비트와 상기 WFC에서 생성된 3비트로 분할된다. 그러므로, 상기 물리적 파형 메모리 및 타이밍 메모리는 최대 4 개의 물리적 타이밍 세트들이 사용될지라도, 핀 당 32의 깊이를 갖는다. 상기 파형 메모리는 상기 파형을 형성하는 가동된 타이밍 모서리들을 포함한다. 상기 가동된 모서리들에 대한 타이밍 값들은 상기 타이밍 메모리로부터 획득된다. 따라서, 상기 프레임 프로세서는 파형들을 정형한다.

매핑 방법론(Mapping Methodology)

상기 방법론은 개별 핀 기반의 파형 테이블 블록 모두를 상기 테스터의 LTS로 매핑하는 것이다. 만약 테스터 하드웨어가 4개의 LTS를 지원한다면, 사용자는 최대 4 개의 파형 테이블 블록들을 정의할 수 있다. 각 파형 테이블 블록은 상기 테스터의 디지털 모듈에 대한 최대 n개의 파형 정의들을 가질 수 있다.

상기 타이밍 맵 파일은, 상기 타이밍 맵 블록에서 정의된 논리적 파형 선택자를 상기 개방형 구조의 테스트 시스템의 모듈을 위한 파형 테이블로 매핑하는 것을 제공한다. 이 경우, 상기 테스터는 256개까지의 논리적 파형 선택자를 지원한다. 개방형 구조의 테스트 시스템에 있어서, 상기 논리적 파형 선택자는 직접 상기 GTS로 매핑한다. 상기 패턴 컴파일러는 상기 패턴 파일을 컴파일 할 수 있도록 하기 위하여, 상기 타이밍 맵과 상기 타이밍 블록 모두에 의존한다. 그러나, 만약 상기 타이밍 블록의 상기 파형 테이블의 상기 파형 문자들이 변경되지 않았거나, 또는 상기 타이밍 맵 블록의 상기 파형 선택자 매핑이 변경되지 않았다면, 그 패턴을 다시 컴파일 할 필요는 없다.

이 매핑 방법론을 사용하는 예

테스터 디지털 모듈로의 매핑을 예시하기 위하여, 다음의 가정이 필요하다. 즉, 상기 프레임 프로세서는 FP 모드로 설정된다. 그리고, CTV 및 MTV 비트들은, GTS 비트들의 전체 개수가 6개이고, 타이밍 뱅크 선택자의 비트들의 전체 개수가 4개가 되도록 설정된다.

상기 타이밍 블록에 정의된 각 파형 테이블은 상기 타이밍 파일 내에서 별개의 LTS로 매핑된다. 이것은 개별 핀 기반으로 수행된다. 그러므로, 파형 테이블 시퀀스 1은 LTS1으로 매핑된다. "SIG" 핀의 경우에, 모두 8개의 가능한 파형 엔트리들이 소모된다. 그러나, 상기 "CLK" 핀은 단 하나의 파형 엔트리를 요구하며, 그러므로 상기 파형 메모리(WFT)와 상기 파형 타이밍 메모리(WTM)의 하나의 열을 소모한다.

도 11에, 상기 "SIG" 핀의 첫번째 2가지 물리적 파형들의 매핑이 도시되었다. 이 파형 테이블이 모서리들의 서로 다른 구성들을 필요로 하는 두 개의 파형 문자들을 매핑하므로, 상기 파형 메모리(WFT) 1112 및 상기 파형 타이밍 메모리(WTM) 1114에 두 개의 엔트리를 할당함으로써 종료한다. 상기 파형의 모양은 상기 WFM에 저장되며, 상기 타이밍의 상세는 상기 WTM에 저장된다. 일 실시예에 의하면, 상기 모듈은 T1, T2, T3, T4, T5 및 T6라는 전체 6개의 타이밍 모서리들을 갖는다. 이들은 상기 타이밍 블록의 모서리 리소스(Edge Resource) 섹션 내에서 상기 파형에 정의된 이벤트들 E1, E2,... 등으로 직접 매핑된다. 만약 6개 이상의 이벤트들이 상기 타이밍 블록에서 정의되었고, 이것이 상기한 모듈과 사용된다면, 그것은 오류로 귀결될 것이다.

도 11의 예에 있어서, 첫번째 파형 문자 "0"은 "강제 하강(Force Down)", 즉 "D" 이벤트를 프로그램하기 위하여 타이밍 모서리 T1을 사용하는데, 상기 "D" 이벤트는 주기 내의 10ns 시간에 발생한다. 타이밍 모서리 T2도 또한 30ns의 시간에 "강제 하강", 즉 "D" 이벤트를 생성하기 위하여 사용된다. 마지막으로, 타이밍 모서리 T3는 45ns의 시간에 "강제 차단(Force Off)", 즉 "Z" 이벤트를 생성하기 위하여 사용된다.

두 번째 파형 문자 "1"은, 주기 내의 10ns의 시간에 발생하는 "강제 상승(Force Up)", 즉 "U"이벤트를 프로그램하기 위하여, 타이밍 모서리 T1을 사용한다. 타이밍 모서리 T2도 또한 30ns의 시간에 "강제 하강", 즉 "D" 이벤트를 생성하기 위하여 사용된다. 마지막으로, 타이밍 모서리 T3는 45ns의 시간에 "강제 차단", 즉 "Z" 이벤트를 생성하기 위하여 사용된다.

이러한 방식으로, 상기 WFC는 상기 프레임 프로세서의 WFM 메모리와 WTM 메모리로 매핑된다. 핀 "SIG"에 대한 LTS1의 파형 메모리 WFM의 마지막 설정이 이하의 표 2에 예시되었다.

[표 2]

Index	(WFC)	T1Set	T1ReSet	T2Set	T2ReSet	T2Drel	T2Dret	EXPH	EXPHZ	T3Set	T3ReSet	T3Drel	T3Dret	T4Drel	T4Dret	EXPL	EXPHZ
0	0		1		1								1				
1	1	1			1								1				
2	d		1	1									1				
3	u	1			1								1				
4	L															1	
5	H							1									
6	m															1	
7	n							1									

상기 핀 "SIG"에 대한 LST1의 파형 타이밍 메모리 WTM의 최종 설정은 이하의 표 3에 예시되었다.

[표 3]

Index	(WFC)	T1	T2	EXPH	T3	T4	EXPL
0	0	10ns	30ns		45ns		
1	1	10ns	30ns		45ns		
2	d	12ns	32ns		42ns		
3	u	12ns	32ns		42ns		
4	L						17ns
5	H			17ns			
6	m						15ns
7	n			15ns			

상기 "CLK" 핀은 하나의 파형을 소모하며, 따라서 이 핀에 대한 상기 WFM 및 WFT는 매우 간단하다. 상기 "CLK" 핀에 대한 LTS1의 파형 메모리 WFM의 최종 셋업이 이하의 표 4에 예시되었다.

[표 4]

Index	(WFC)	T1Set	T1ReSet	T2Set	T2ReSet	T2Drel	T2Dret	EXPH	EXPHZ	T3Set	T3ReSet	T3Drel	T3Dret	T4Drel	T4Dret	EXPL	EXPHZ
0	1	1			1												
1																	
2																	
3																	
4																	
5																	
6																	
7																	

LTS2의 파형 타이밍 메모리의 최종 셋업이 이하의 표 5에 예시되었다.

[표 5]

Index	(WFC)	T1	T2	EXPH	T3	T4	EXPL
0	1	20ns	40ns				
1							
2							
3							
4							
5							
6							
7							

상기 타이밍 맵 블록은 상기 파형 선택자를 꺼내어 상기 타이밍 블록의 파형 테이블로 명시적으로 매핑한다. 테스터 시스템에 있어서, 이것은 결국 상기 타이밍 세트 스크램블러(TSS) 메모리를 설정하도록 된다. 상기 TSS는 기본적으로 GTS로부터 상기 설정을 보유하는 LTS로의 매핑을 포함한다. 본 실시예에 있어서, 핀 SIG에 대한 상기 TSS 셋업은 이하의 표 6과 같을 것이다.

[표 6]

GTS	LTS
0 (wfs1)	1
1 (wfs2)	1
2 (wfs3)	2
3 (wfs4)	1
4 (wfs5)	3
5 (wfs6)	1
.	
N (wfs1)	1
.	
255	

마지막으로 상기 TSS 및 LTS 셋업 매핑이 해석된 후에, 상기 패턴 컴파일러는 사용할 정확한 파형 테이블(LTS) 및 정확한 파형 문자를 가진 패턴을 프로그램하기 위하여 이 정보를 사용할 수 있다. 그러므로, 상기 예시한 핀 "SIG"만을 고려하는 모의 패턴이 도 11에 도시되었다. 이 컴파일은 타이밍 블록에는 의존하지 않으며, 단지 타이밍 맵 블록에만 의존한다는 것을 주의하여야 한다.

G. 테스터 동작(Tester Operation)

이 섹션은 상기 테스터 운영 체계(tester opertating system; TOS)의 기초적인 동작들에 관하여 기술한다. 이 섹션에서 고려되는 행동들은 다음과 같다. 즉,

시스템 초기화(system initialization)

테스트 계획 로딩(Test Plan loading)

패턴 로딩(Pattern loading)

테스트 계획 실행(Running a Test Plan)

개별 시험 실행(Running an individual Test)

시스템 초기화

본 발명의 일 실시예에 의하여 시스템을 초기화하기 위하여, 특정의 가정이 만족되어야 하며, 특정의 조건이 충족되어야 한다. 이하의 서브 섹션에서 이들을 나열한다.

선행조건(Preconditions)

관련된 시스템 소프트웨어 컴포넌트의 사본들은 중앙의 저장소를 가지고 있으며, 그 위치는 상기 시스템 제어기에 알려져 있다. 이것은 시스템 제어기 그 자체상일 수도 있으며, 또는 네트워크에 장착된 디렉토리(또는 다른 메커니즘을 통해 상기 시스템 제어기에 알려진)를 가진 다른 시스템상일 수도 있으며, 어떠한 메커니즘이건 간에, 상기 시스템이 기능하기 전에 모든 소프트웨어는 상기 시스템 제어기의 사용을 위하여 활용 가능하게 되어 있어야 한다. 이러한 소프트웨어는 다음과 같은 것들이다. 즉,

벤더 하드웨어 제어(즉, 모듈 소프트웨어) DLL들,

표준 또는 사용자 테스트 클래스 DLL들, 및

사용자 테스트 계획 DLL들.

상기 시스템 모듈 구성 파일은 상기 시스템 제어기상에서 활용 가능하다. 이 파일은, 사용자로 하여금 상기 모듈 소프트웨어 DLL들의 명칭들 뿐만 아니라, 예를 들어 상기 시스템 섀시의 각 모듈의 물리적 위치 및 형식과 같은 상기 테스터의 물리적 구성을 규정할 수 있도록 한다는 것을 기억하라.

상기 시스템 구성 파일은 상기 시스템 제어기상에서 활용 가능하다. 이 파일은, 스위치 행렬의 입력 포트 어드레스들에 대한 사이트 제어기의 호스트 명칭들의 맵과 함께, 상기 시스템 내에서의 사이트 제어기들의 리스트를 수록한다는 것을 기억하라.

사이트 제어기는 사이트 구성 관리자(SCM)이라 불리는 서비스가 실행되도록 한다. 이 서비스는, "하드웨어 발견"이라고 불리는 프로세스에 의하여, 각 슬롯에 어떤 하드웨어가 설치되었는지를 결정할 책임을 진다. 그것은 또한 상기 시스템 제어기와 함께 상기 시스템 초기화 프로세스에 참여할 책임도 진다. 본 발명의 일 실시예에 의하면, 상기 스위치 행렬 동작 프로토콜이, 스위치 행렬 입력 포트 접속 어드레스 1을 가진 하나의 사이트 제어기에서의 상기 SCM로 하여금 항상 상기 모듈로의 상기 스위치 행렬 접속을 구성하는 데에 사용되도록 지시한다는 것을 주의하여야 한다. 이 "특별한" 사이트는 SITEC-1으로 표기되는 것을 기억하라.

상기 시스템 제어기는, 각 사이트 제어기의 SCM에게 그의 스위치 행렬 접속 어드레스를 제공할 책임을 진다.

각 사이트 제어기의 SCM은 테스트 계획 서버(Test Plan Server; TPS)라 불리는 프로세스를 시작할 능력이 있다. 각 사이트 제어기상의 상기 테스트 계획 서버는 궁극적으로 상기 사용자의 테스트 계획(또는 하나의 사이트 제어기가 다수의 피시험 디바이스에 대하여 시험들을 실행하고 있는 경우, 테스트 계획들)을 수록하고 실행할 궁극적 책임을 진다.

초기화 제1단계: 시스템 유효화(Initialization Phase I: System Validation)

일단 상기한 가정들 및 선행 조건들이 만족되면, 시스템 초기화가 먼저 다음과 같은 시스템 유효화 단계로부터 진행된다. 즉,

1. 상기 시스템 제어기는 상기 사용자가 규정한 시스템의 관점을 초기화하기 위하여 상기 시스템 및 모듈 구성 파일들을 읽는다.

2. 상기 규정된 시스템 구성 정보를 사용하여, 상기 시스템 제어기는 상기 규정된 사이트 제어기가 활동 중이며, 도달 가능하며, 준비되어 있음(즉, SCM을 실행시킴)을 확인한다. 이 확인 단계에서의 오류는 모두 시스템 오류를 발생시키며, 초기화는 실패한다.

3. 이어서, 상기 시스템 제어기는, 상기 SITEC-1의 SCM 서비스로 하여금 모든 하드웨어 모듈로의 억세스를 구비하도록 상기 스위치 행렬을 구성하게 하며, 그리고 하드웨어 발견을 수행할 것을 요구한다.

4. 상기 SITEC-1의 SCM 서비스는, {벤더, 하드웨어} 집합들을 위하여 모든 활용 가능한 모듈의 슬롯(이미 알려진 하드웨어 위치)들에 대해 폴링(polling)을 수행하며, {벤더, 하드웨어} 집합들의 슬롯에 대한 맵을 생성한다. 결론으로서, 이 폴링은 전체 시스템에 존재하는 {벤더, 하드웨어, 슬롯} 묶음들의 전체 집합을 식별한다. 이 폴링의 결과는 상기 시스템 제어기로 보내진다.

5. 상기 시스템 제어기는, 상기 하드웨어 발견 단계의 결과들이 상기 모듈 구성 파일의 사용자가 규정한 구성과 일치한다는 것을 확인한다. 이 확인 단계에서의 오류는 모두 시스템 오류를 발생시키고, 초기화는 실패한다.

6. 이어서, 상기 시스템 제어기는, 잘 알려진 위치에서 상기 환경 셋업 파일로부터 디폴트 환경(모듈 DLL에 대한 검색 경로, 패턴 리스트, 테스트 계획 DLL, 테스트 클래스 DLL, 등과 같은)을 로드한다.

7. 상기 시스템 제어기는 모든 식별된 모듈 소프트웨어 DLL이 존재한다는 것을 확실히 한다. 만약 시스템 제어기상에서 활용 가능하지 않다면, 가능하다면 그것은 중앙 저장소로부터 검색되며, 그렇지 않으면 시스템 오류가 발생하고, 초기화는 실패한다.

초기화 제2단계: 사이트 구성(Site Configuration)(선택적)

사이트 구성 또는 사이트 분할(site partitioning)은, 활용 가능한 시스템 하드웨어 모듈의 서로 다른 사이트로의 소프트웨어 레벨의 할당을 수반한다(즉, 복수의 피시험 디바이스를 지원하기 위하여). 사이트 분할 정보는 소켓 파일에 제공된다는 것을 기억하라.

상기 테스터 시스템은, 사이트 (재)분할이, 테스트 계획 로드(각 테스트 계획은 특정 소켓과 관련되므로)의 일부로서, 또한 독립적인 사용자 호출 가능 단계로서 모두 수행되도록 허용한다. 후자의 경우에, 상기 사용자는, 상기 시스템을 분할하는 데에만 사용되는 소켓 파일을 제공함으로써 상기 사이트 분할을 개시한다. 이것은, 각 사이트가 서로 다른 형식의 피시험 디바이스를 시험하는 다중 피시험 디바이스 시험의 경우에서의 시스템 초기화에 특히 유용하다. 그러나, 이 단계는 상기 초기화 단계에서는 선택적이며, 상기 사용자는 그것이 실행되지 않도록 선택할 수 있으며, 대신에 테스트 계획 로드로 하여금 상기 시스템을 적절히 분할하도록 선택할 수 있다.

사이트 분할에 영향을 주기 위하여 선택된 수단(독립적인 호출에 의하거나 또는 테스트 계획 로드를 통하여 묵시적으로)이 어떤 것이든, 그 메커니즘은 동일하다. 이 메커니즘을 이하에서 기술한다.

1. 소켓이 주어지면, 상기 시스템 제어기는 먼저 현재 존재하는 시스템 분할이 상기 소켓과 호환적인지 또는 재분할이 필요한지의 여부를 판단한다. 초기화 동안의 디폴트 분할은 모든 활용 가능한 모듈들이 SITEC-1에 접속되는 것이다. 이하의 나머지 단계들은 단지 재분할이 필요한 경우에만 수행된다.

2. 상기 시스템 제어기는, 각 사이트 제어기 SCM에게, 새로운 소켓하에서 그것을 위하여 가동된 피시험 디바이스 사이트들의 개수와 식별자들로 스스로를 재구성하도록, 구성 메시지를 송신한다. 이것은 일반적인 절차이며, 하나의 사이트 제어기에 의하여 제어되는 피시험 디바이스 사이트의 개수가 하나인 경우를 다룬다는 점을 주의하여야 한다. 상기 새로운 소켓 정보도 역시 상기 SCM으로 전달된다.

3. 각 SCM은 만약 존재한다면 실행중인 TPS를 정지시키고, 상기 새로운 소켓들과, 새로운 소켓하에서 그것을 위하여 가동된 피시험 디바이스 사이트들의 개수와 식별자들로 그것을 초기화하며 새로운 TPS를 시작한다.

4. 상기 시스템 제어기는, 어느 사이트가 요구된 시스템 모듈들의 어떠한 서브-세트들이 필요한지를 판단한다. 이를 수행하는 동안, 그것은 그 사이트들을 위한 하드웨어 슬롯 정보를 또한 준비한다. 각 사이트에 대한 최종 결과는 그 사이트에 대하여 할당된 슬롯 대 모듈 DLL들의 리스트이다.

5. 상기 시스템 제어기는, 각 SCM에게 필요한 모듈 DLL들 뿐만 아니라, 적절한 SITE-MDSL을 제공한다. 각 SCM은 차례로 이 정보가 상기 새로이 시작된 TPS에게 활용 가능하게 만든다.

6. 이어서, 상기 시스템 제어기는, SITEC-1으로 하여금 적절한 사이트-슬롯 접속에 대하여, 즉 사이트 분할 동작에 대하여 상기 스위치 행렬을 구성할 것을 요구한다.

7. 사이트 1 내지 n까지의 TPS들은 그들의 SITE-MDSL에 규정된 DLL들을 로드한다. 이들 DLL들의 각각은, 슬롯 번호의 배열을 취하는 'initialize()'로 명명된 함수를 갖는다. 상기 TPS들은 그 모듈 형식에 대한 적절한 슬롯 리스트들을 가지고 initialize()를 호출한다. 이 지점에서의 모든 오동작에 대하여, 시스템 오류가 발생하며, 초기화는 실패한다. 상기 initialize() 메소드는 이하를 수행한다. 즉,

a. 표준 인터페이스의 IXXX 모듈에 기초하여 유형적 클래스들(concrete classes)을 창출한다. 예를 들어, 디지털 모듈과 관련된 DLL은, 그것이 관련된 각 슬롯을 지원하기 위하여 하나의 IPinModule에 기초한 객체를 창출할 것이다.

b. 모듈 내의 각 "리소스 유닛(resource unit)"에 대한 것인 IResource 인터페이스에 기초한 유형적 클래스들을 창출한다. 다시 한 번, 디지털 모듈에 있어서, 각 IPinModule에 기초한 객체는 디지털 모듈들에 의하여 점유된 슬롯들의 모음의 모든 핀들에 대한 ITesterPin에 기초한 객채들을 창출할 것이다.

8. 이어서, 사이트 1 내지 n의 TPS들은, 모듈 내용 정보를 검색하기 위하여 각 로드된 모듈 DLL에 대하여 getXXXModule()을 호출할 것이다.

9. getXXXModule()에 대한 각 호출은, IModule 포인터(예컨대, AdvantestPinModule)로서 <VendorHWType>Module 클래스 객체를 리턴한다. 그러한 IMoudule 포인터는 상기 TPS에 의하여 캐시되며, 그것은 이들을 프레임워크/사용자 코드에 대하여 활용 가능하게 만든다. IModules, IResources, 등의 모음은 (적어도 TPS의 수명 동안은) 영속적(persistent)이라는 점을 주의하여야 한다.

10. 일단 상기한 단계들이 완료되면, 상기 TPS는 그 할당된 (잘 알려진) 포트에서 listen()을 실행한다. 이것은 시스템 제어기에게 상기 TPS가 정상(즉, 사이트를 분할한) 동작을 수행할 준비가 되어 있다는 것을 신호한다.

테스트 계획 로드(Test Plan Load)

이 섹션은, 사용자의 테스트 계획 DLL이 (하나의 또는 다수의 피시험 디바이스의 시험을 위하여) 사이트 제어기로 로드되는 단계들을 기술한다.

일단 시스템 초기화(그리고, 선택적으로, 최초의 사이트 분할)가 완료되면, 사용자 테스트 계획이 로드될 수 있다. 사용자 테스트 계획을 사이트 제어기에 로드하는 것은 다음과 같이 진행된다. 즉,

1. 상기 시스템 제어기는 먼저, 그에게 그의 관련된 소켓 파일과 그의 피시험 디바이스 형식 식별자에 관하여 질의하면서, 시스템 계획 DLL을 그 자신의 프로세스 공간으로 로드한다. 이 정보는 이 테스트 계획을 실행하는 사이트, 그리고 따라서 이 테스트 계획이 로드될 상기 사이트 제어기(들)를 식별하는 데에 사용된다.

2. 이어서, 상기 시스템 제어기는 상기에서 설명한 바와 같이 재분할 프로세스를 개시하는 테스트 계획에 관련된 소켓 정보를 사용한다.

3. 상기 시스템 제어기는, 상기 테스트 계획 DLL로부터의 테스트 계획에 의하여 사용되는 테스트 클래스 DLL의 리스트를 추출하고, 일단 상기 시스템 제어기에 의하여 상기 TPS가 정상(즉, 사이트가 분할된) 동작을 시작할 준비가 되었다는 것이 확인되면, 적합한 TPS에게 상기 테스트 클래스 DLL과 마지막으로 그 테스트 계획 DLL 자체를 송신한다.

4. 상기 TPS는 그것을 그의 프로세스 공간으로 로드하도록 LoadLibrary()를 호출한다. 그것은, 그것이 지원하는 사이트들(즉 피시험 디바이스들)의 개수만큼의 TestPlan 객체들을 창출하기 위하여 상기 DLL 상으로 잘 알려진 함수를 호출한다.

5. 상기 TPS는 필요한 테스터 프레임워크 객체로 상기 TestPlan 객체(들)을 초기화한다. 초기화하는 동안, 상기 TPS는 상기 TestPlan 객체(들)에 의하여 사용되는 테스트 클래스들에 대한 적절한 DLL들을 상기 프로세스 공간으로 로드하고, 상기 테스트 클래스 인스턴스를 창출한다.

6. 상기 TPS는 상기 TestPlan 객체(들)에 상기 시스템 제어기로부터의/로의 통신 채널을 셋업한다.

7. 상기 시스템 제어기는 TPS와 통신하며 상기 TestPlan 객체(들)을 위한 그의 프록시들을 구축한다.

그것은 사이트 제어기 상으로의 사용자 테스트 계획의 성공적인 로드로 결론짓는다.

테스트 계획의 실행(Running a Test Plan)

미리 정의된 흐름 로직에 의한 테스트 계획에서의 모든 시험을 실행하는 방법은 다음과 같다. 즉,

1. 상기 사용자의 응용 프로그램은 상기 TPS에게 *RunTesPlan* 메시지를 전송한다. 상기 TPS는 모든 접속된 응용 프로그램들에게 *ExecutingTestPlan* 메시지를 송신한다. 이어서, 상기 TPS는 상기 테스트 계획상으로 *execute()*를 호출한다.

2. 하나의 사이트 제어기로 다수의 피시험 디바이스를 시험하는 것은, 피시험 디바이스 하나당, 그 사이트 제어기 상의 다수의 쓰레드를 이용함으로써 수행된다. 각 쓰레드는 동일한 TestPlan 객체의 서로 다르고 독립적인 인스턴스를 실행시킨다. 이 경우에는, 모듈 제어 소프트웨어 DLL들이 피시험 디바이스들에 걸쳐서 공유될 수 있으므로, 하드웨어 통신에 대한 모듈 명령들은 피시험 디바이스 식별자 파라미터를 취할 것이 요구된다.

3. 상기 TestPlan 객체는, preExec(), execute() 및 postExec()를 호출하며 그 모음의 각 시험에 대하여 반복된다(또는 그의 흐름 객체로 하여금 그 흐름 로직에p 따라 각 시험을 처리하도록 지시한다.).

4. 각 시험이 실행됨에 따라, 상태 메시지가 모든 접속된 응용 프로그램들에게 회송된다.

하나의 시험의 실행(Executing a Single Test)

사용자는 모든 시험들 대신에 하나의 테스트 계획의 하나의 시험을 실행시키고자 할 수도 있다. 하나의 시험의 실행을 위한 방법은 다음과 같다. 즉,

1. 사용자의 응용 프로그램은 *RunTest* 메시지를 TPS로 전송한다. 상기 TPS는 모든 접속된 응용 프로그램에게 *ExecutingTest* 메시지를 송신한다. 이어서, 상기 TPS는 실행될 시험을 규정하며 상기 테스트 계획 상으로 executeTest()를호출한다.

2. 상기 테스트 계획 객체는 상기 테스트 객체상으로 preExe(), execute() 및 postExec()를 호출하며 규정된 시험을 실행한다.

3. 시험이 실행될 때, 그것은 모든 접속된 응용 프로그램들에게 상태 메시지를 회송한다.

테스트 프로그램을 개발하는 개시된 방법에 의해 성취되는 적어도 두 가지 이점이 있다. 첫째, 상기 방법은 테스트 계획이 처리하도록 계획된 문제 도메인에 더 가까운 언어로 테스트 계획을 기술함으로써, 테스트 계획 개발자들로부터 시스템 프로그래밍의 복잡성을 분리한다. 둘째, 상기 방법은 테스트 프로그램 개발의 초기 단계에서 테스트 엔지니어가 테스트 계획의 오류를 식별하도록 하며, 이는 테스트 프로그램을 개발하는 시간과 대응하는 DUTs가 테스트되는 시간을 줄일 수 있다.

관련 기술 분야에서 숙련된 자는, 기초가 되는 동일한 메커니즘과 방법론을 사용하면서, 개시된 실시예의 많은 가능한 변경과 조합이 사용될 수 있다는 것을 인식할 것이다. 설명의 목적으로, 상기의 기술은 특정 실시예를 참조로 하여 쓰여졌다. 그러나, 상기의 예시적인 논의는 모든 것을 남김없이 규명하거나 또는 본 발명을 개시된 정확한 형태로 제한하고자 의도된 것은 아니다. 많은 변경과 변형이 상기 시사(示唆)의 관점에서 가능하다. 상기 실시예들은 본 발명의 원리와 그 실제 응용을 설명하기 위해 선택되고 기술되었고, 관련 기술 분야의 숙련된 자가 본 발명, 그리고, 고려될 수 있는 특정 사용에 적절한 다양한 변형을 갖는 다양한 실시예를 가장 잘 사용하도록 한다.

도면의 간단한 설명

본 발명의 추가적인 특성 및 이점과 상기한 특성 및 이점은, 다음의 도면과 결합되었을 때 본 발명의 실시예에 관한 이하의 상세한 설명의 결과로서 더욱 명확히 이해될 것이다.

도 1은, 통상적인 테스터 구조를 도시한다.

도 2는, 본 발명의 일 실시예에 의한 시스템 구조를 도시한다.

도 3은, 본 발명의 일 실시예에 의한 소프트웨어 구조를 도시한다.

도 4는, 본 발명의 일 실시예에 의한 테스트 프로그램 컴파일러를 도시한다.

도 5는, 본 발명의 일 실시예에 의하여, 하나의 테스트 클래스로부터 서로 다른 테스트 인스턴스들(test instances)이 유도되는 방법을 도시한다.

도 6은, 본 발명의 일 실시예에 의한 패턴 컴파일러를 도시한다.

도 7은, 본 발명의 일 실시예에 의한 정렬된 패턴 트리(ordered pattern tree)의 예를 도시한다.

도 8은, 본 발명의 일 실시예에 의한 정렬된 패턴 트리의 다른 예를 도시한다.

도 9는 본 발명의 일 실시예에 의한 테스트 프로그램에 의하여 요구되는 파일들 사이의 관계를 도시한다.

도 10은 본 발명의 일 실시예에 의한 파형 생성을 도시한다.

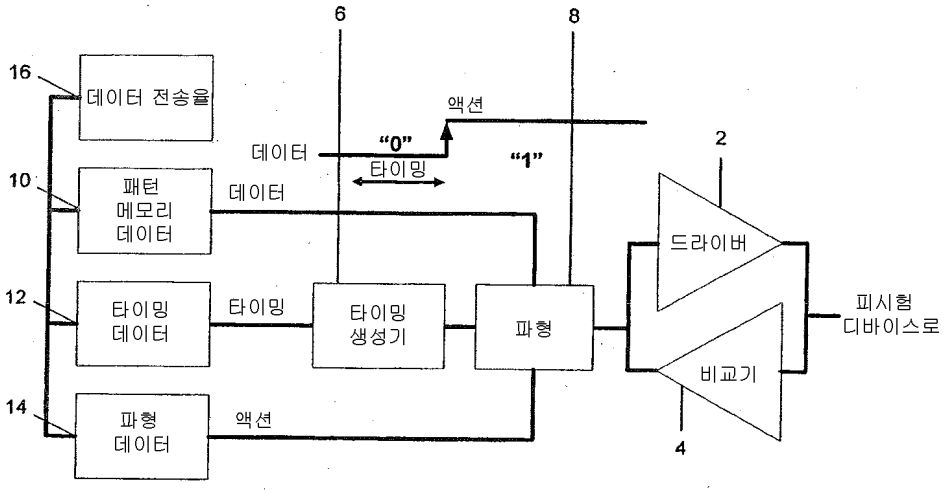
도 11은, 본 발명의 일 실시예에 의한 타이밍에 사용되는 매핑(mapping)을 도시한다.

도 12는, 본 발명의 일 실시예에 의한 타이밍에 사용되는 다른 매핑을 도시한다.

도 13은, 본 발명의 일 실시예에 의한 테스트 프로그램을 개발하는 방법을 도시한다.

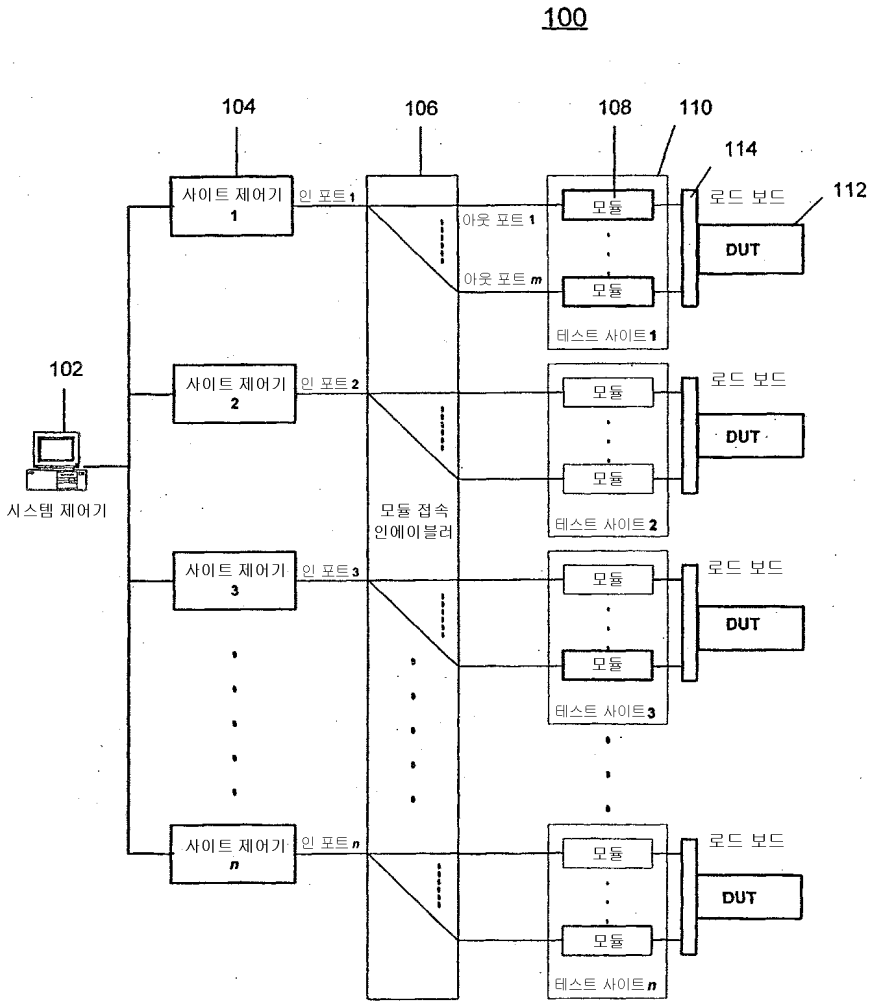
도면

도면1

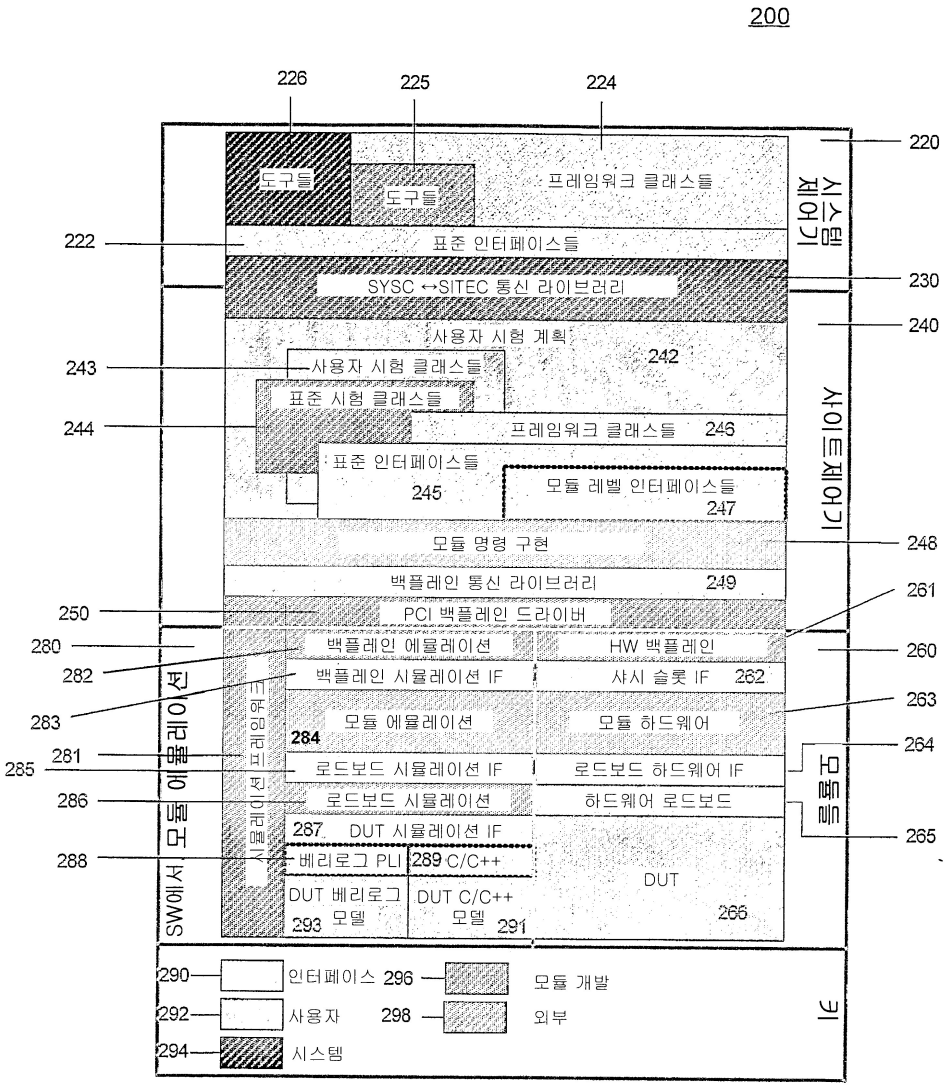


액션 = 패턴 데이터 * 타이밍 데이터 (시간 설정에 의하여 제한됨)* 파형

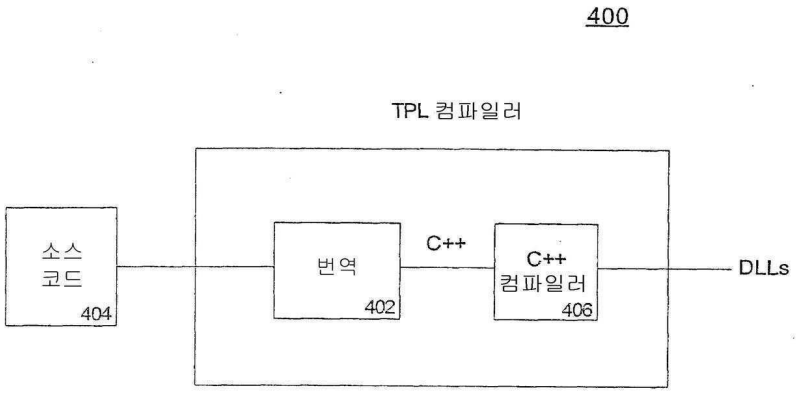
도면2


100
104
106
108
110
114
112
102
사이트 제어기 1
사이트 제어기 2
사이트 제어기 3
사이트 제어기 n
인 포트 1
인 포트 2
인 포트 3
인 포트 n
아웃 포트 1
아웃 포트 m
모듈
로드 보드
DUT
테스트 사이트 1
테스트 사이트 2
테스트 사이트 3
테스트 사이트 n
시스템 제어기
모듈 접속 인에이블러

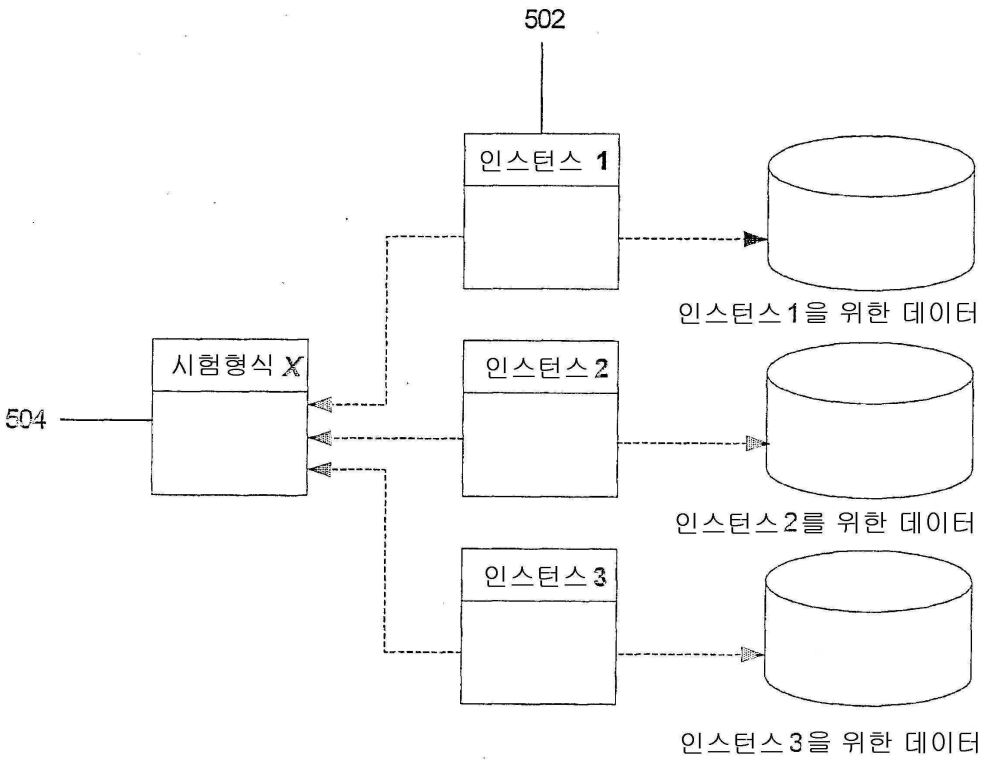
도면3


200
226
225
224
220
도구들
도구들
프레임워크 클래스들
시스템 제어기
222
표준 인터페이스들
SYSC ↔SITEC 통신 라이브러리
230
240
사용자 시험 계획
242
243
사용자 시험 클래스들
표준 시험 클래스들
244
프레임워크 클래스들 246
표준 인터페이스들 245
모듈 레벨 인터페이스들 247
사이트제어기
248
모듈 명령 구현
백플레인 통신 라이브러리 249
250
PCI 백플레인 드라이버
261
280
백플레인 에뮬레이션
HW 백플레인
260
282
백플레인 시뮬레이션 IF
샤시 슬롯 IF 262
283
263
모듈 에뮬레이션
284
모듈 하드웨어
264
281
285
로드보드 시뮬레이션 IF
로드보드 하드웨어 IF
286
로드보드 시뮬레이션
하드웨어 로드보드
모듈들
287 DUT 시뮬레이션 IF
265
288
베리로그 PLI
289 C/C++
DUT
266
DUT 베리로그 모델 293
DUT C/C++ 모델 291
SW에서 모듈 에뮬레이션
하드웨어 인터페이스
290
인터페이스 296
모듈 개발
292
사용자
298
외부
294
시스템
키

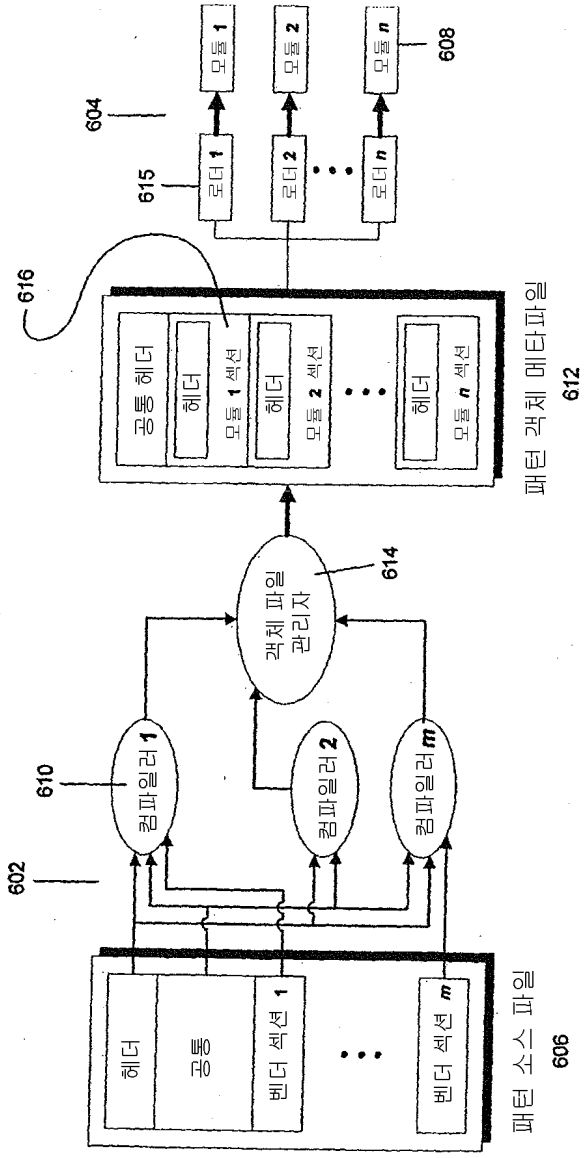
도면4


400
TPL 컴파일러
소스 코드 404
번역 402
C++
C++ 컴파일러 406
DLLs

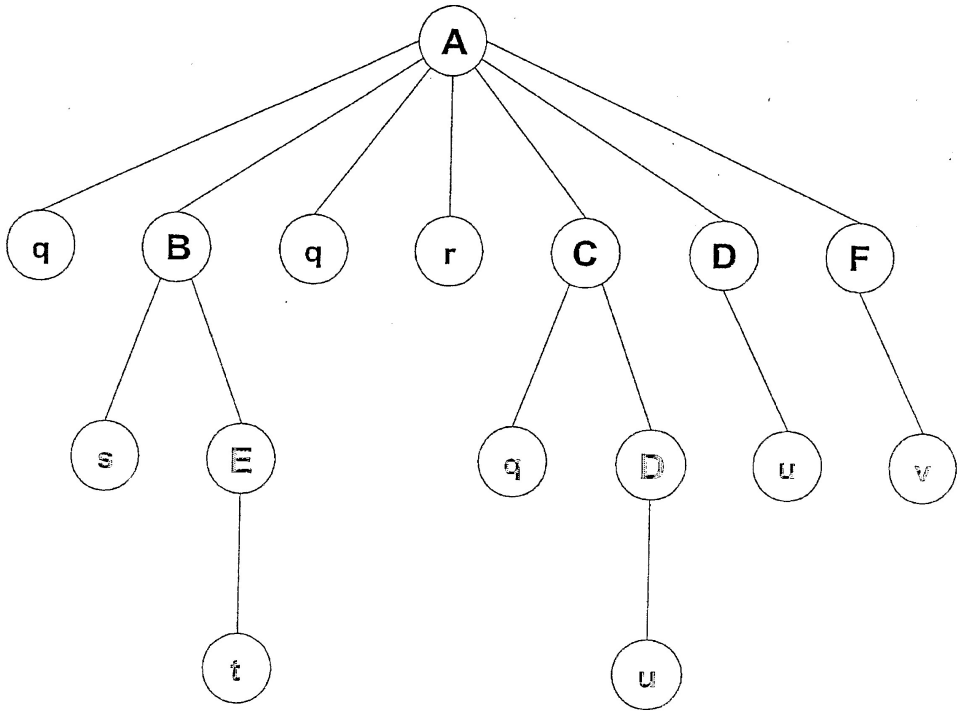
도면5


502
인스턴스 1
인스턴스1을 위한 데이터
시험형식 X
504
인스턴스 2
인스턴스2를 위한 데이터
인스턴스 3
인스턴스3을 위한 데이터

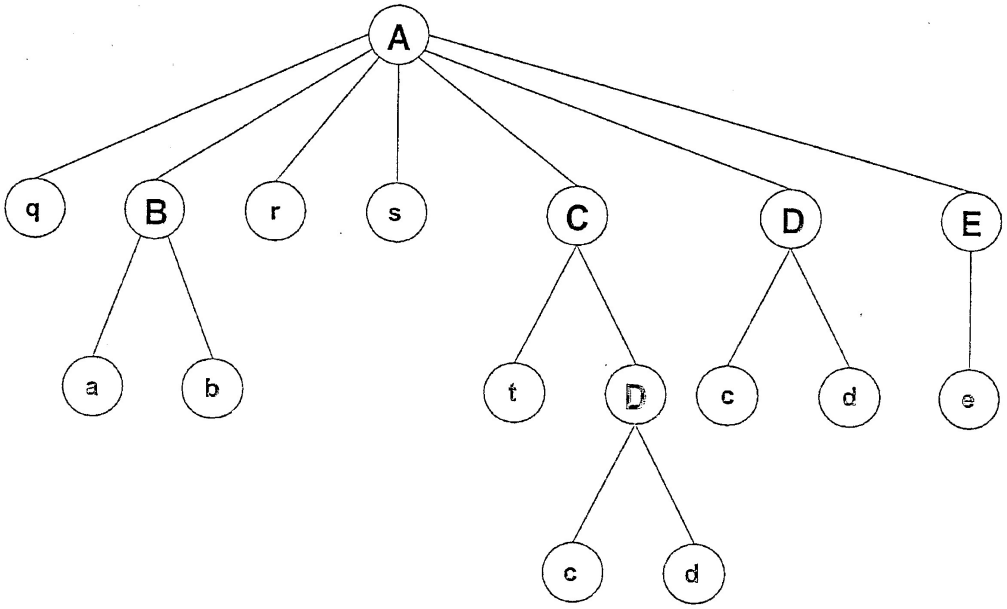
도면6


604
615
616
602
610
614
612
606
608
모듈 1
모듈 2
모듈 n
로더 1
로더 2
로더 n
공통 헤더
헤더
모듈 1 섹션
헤더
모듈 2 섹션
헤더
모듈 n 섹션
패턴 객체 메타파일
객체 파일 관리자
컴파일러 1
컴파일러 2
컴파일러 m
헤더
공통
벤더 섹션 1
벤더 섹션 m
패턴 소스 파일

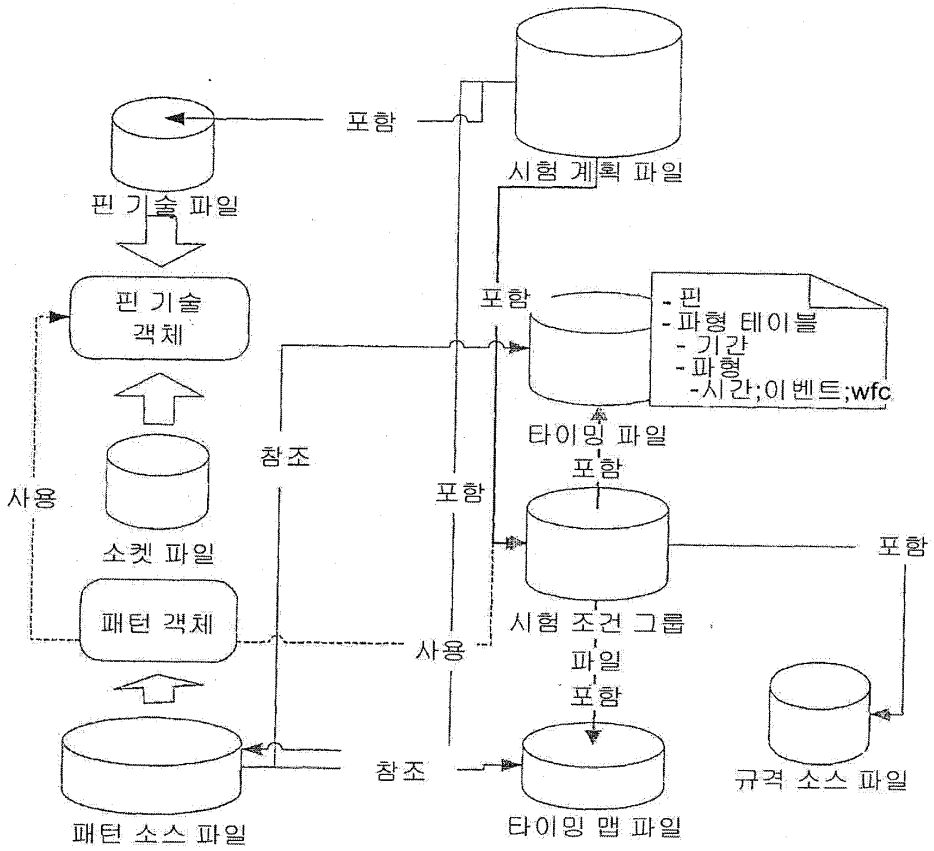
도면7



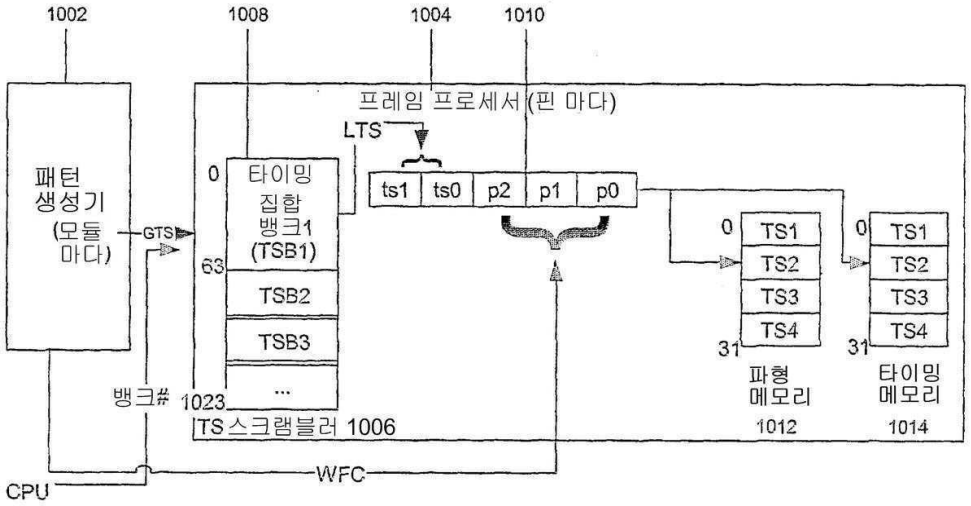
도면8



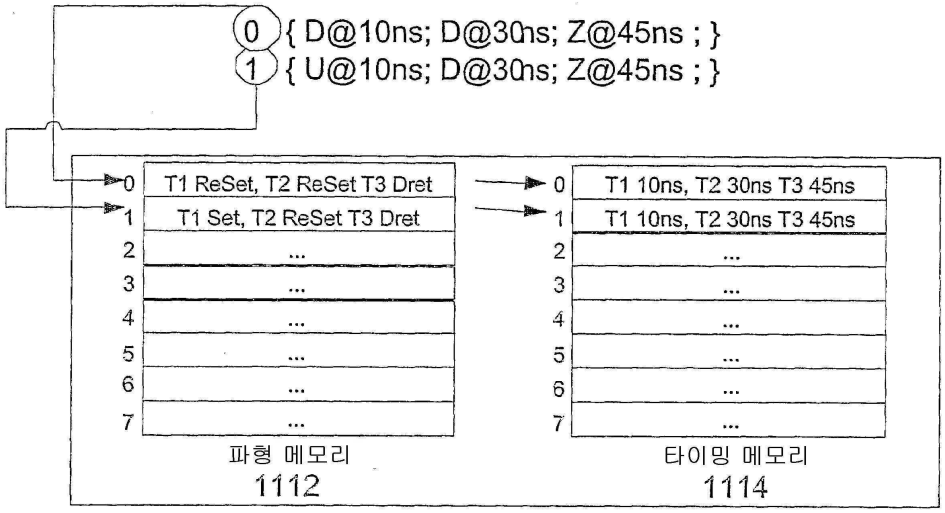
도면9


포함
시험 계획 파일
핀 기술 파일
핀 기술
객체
포함
- 핀
- 파형 테이블
- 기간
- 파형
-시간;이벤트;wfc
타이밍 파일
참조
포함
사용
포함
소켓 파일
포함
시험 조건 그룹
파일
패턴 객체
사용
포함
참조
규격 소스 파일
패턴 소스 파일
타이밍 맵 파일

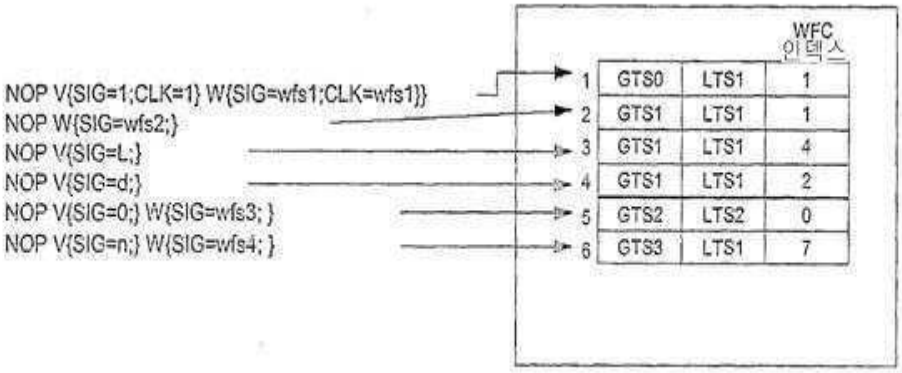
도면10


1002
1008
1004
1010
프레임 프로세서 (핀 마다)
LTS
패턴
생성기
(모듈
마다)
GTS
0
타이밍
집합
뱅크1
(TSB1)
63
TSB2
TSB3
...
1023
ts1
ts0
p2
p1
p0
0
TS1
TS2
TS3
TS4
31
0
TS1
TS2
TS3
TS4
31
뱅크#
TS 스크램블러 1006
파형
메모리
1012
타이밍
메모리
1014
CPU
WFC

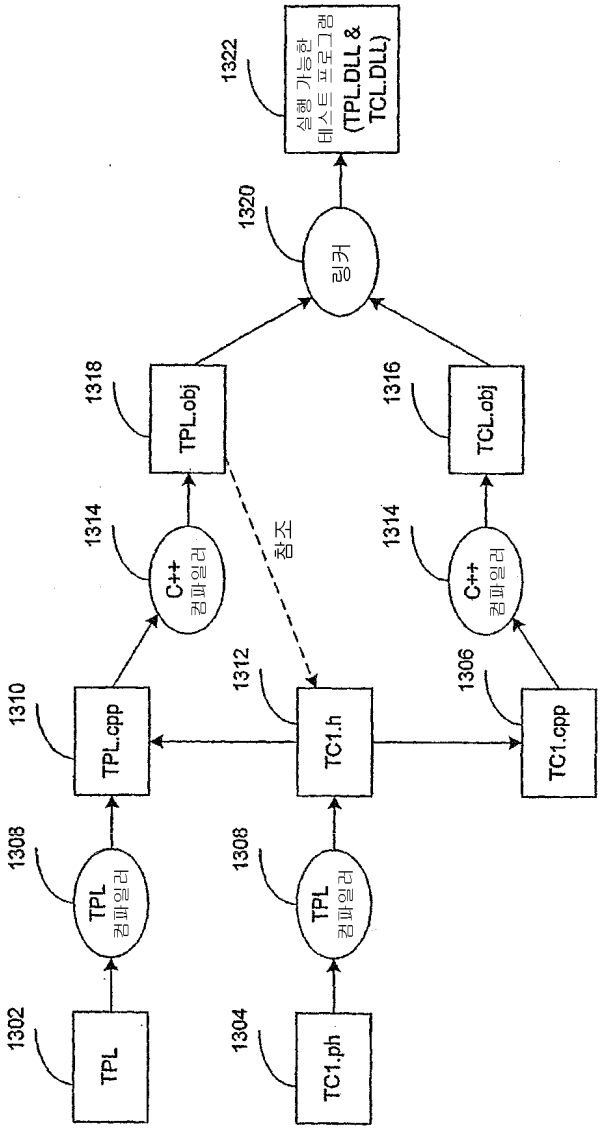
도면11


0 { D@10ns; D@30ns; Z@45ns ; }
1 { U@10ns; D@30ns; Z@45ns ; }
0 T1 ReSet, T2 ReSet T3 Dret
1 T1 Set, T2 ReSet T3 Dret
2 ...
3 ...
4 ...
5 ...
6 ...
7 ...
파형 메모리
1112
0 T1 10ns, T2 30ns T3 45ns
1 T1 10ns, T2 30ns T3 45ns
2 ...
3 ...
4 ...
5 ...
6 ...
7 ...
타이밍 메모리
1114

도면12


NOP V{SIG=1;CLK=1} W{SIG=wfs1;CLK=wfs1}}
NOP W{SIG=wfs2;}
NOP V{SIG=L;}
NOP V{SIG=d;}
NOP V{SIG=0;} W{SIG=wfs3; }
NOP V{SIG=n;} W{SIG=wfs4; }
WFC 인덱스
1 GTS0 LTS1 1
2 GTS1 LTS1 1
3 GTS1 LTS1 4
4 GTS1 LTS1 2
5 GTS2 LTS2 0
6 GTS3 LTS1 7

도면13


1322
실행 가능한
데스트 프로그램
(TPL.DLL &
TCL.DLL)
1320
링커
1318
TPL.obj
1316
TCL.obj
1314
C++
컴파일러
참조
1314
C++
컴파일러
1310
TPL.cpp
1312
TC1.h
1306
TC1.cpp
1308
TPL
컴파일러
1308
TPL
컴파일러
1302
TPL
1304
TC1.ph