DOI: 10.3745/KIPSTD.2010.17D.1.041

1대1 요구사항 모델링을 통한 테스트 케이스 자동 생성

오 정 섭[†] · 최 경 희[††] · 정 기 현[†††]

요 약

생성된 테스트 케이스와 요구사항과의 연관관계가 중요하지만, 모델을 이용한 테스트 케이스 자동생성에서는 모델이 요구사항과 m:n의 관계를 맺기 때문에 테스트 케이스와 요구사항과의 관계도 매우 복잡해진다. 본 논문에서는 1:1 모델링 도구인 REED(REquirement EDitor)를 이용하여 테스트 케이스를 생성하는 방법에 대하여 기술한다. 테스트 케이스는 커버리지 타겟 생성, IORT(Input Output Relation Tree) 생성, 테스트 케이스 생성의 3단계를 거치며, 모든 단계는 자동으로 진행된다. 생성된 테스트 케이스는 하나의 요구사항에서 생성될 수 있으며 실제 시스템에 적용한 결과, 온도조절장치 경우는 5,566개, 버스카드 단말기의 경우는 3,757개, 굴착기 제어기는 4,611개의 테스트 케이스가 생성되었다.

키워드 : 요구사항, 요구사항 모델, MC/DC, 테스트 케이스, REED, RBT

Automatic Test Case Generation Through 1-to-1 Requirement Modeling

Jungsup Oh[†] · Kyunghee Choi[††] · Gihyun Jung[†††]

ABSTRACT

A relation between generated test cases and an original requirement is important, but it becomes very complex because a relation between requirement models and requirements are m-to-n in automatic test case generation based on models. In this paper, I suggest automatic generation technique for REED (REquirement EDitor), 1-to-1 requirement modeling tool. Test cases are generated though 3 steps, Coverage Target Generation, IORT (Input Output Relation Tree)Generation, and Test Cases Generation. All these steps are running automatically. The generated test cases can be generated from a single requirement. As a result of applying to three real commercial systems, there are 5566 test cases for the Temperature Controller, 3757 test cases for Bus Card Terminal, and 4611 test cases for Excavator Controller.

Keywords : Requirement, Requirement Model, MC/DC, Test Cases, REED, RBT

1. 서 론

고객의 요구사항이 제품에 올바르게 구현되었는지를 검사하기 위해서 테스트 케이스를 생성한다. 테스트 케이스는 테스트 엔지니어에 의해서 수동으로 작성되거나 자동화 도구에 의해서 자동으로 생성될 수 있다. 테스트 케이스를 수동으로 작성하는 방법은 테스트 케이스를 작성하는 테스트 엔지니어의 능력에 의해서 테스트 케이스의 품질이 달라질 수 있고 많은 시간과 비용이 소요되는 단점을 가지고 있어 점차 자동화 기법을 많이 사용하고 있다 [1-3].

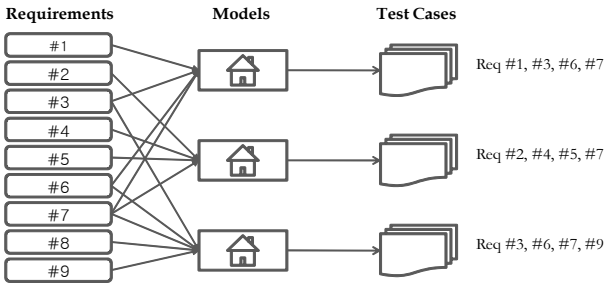
테스트 케이스를 자동으로 생성하기 위하여 요구사항을 일정한 형식의 모델로 변환하고, 모델로 부터 테스트 케이스를 생성하는 방식을 많이 사용한다 [4-7]. 그러나 기존의 테스트 케이스를 생성하는 데 사용된 Petri-Nets, EFSM, UML, Simulink/Stateflow 등의 모델들은 (그림 1)과 같이 요구사항과 모델 사이에 m:n의 관계가 성립한다.

일반적으로 테스트 케이스와 요구사항과의 연관관계를 쉽게 추적할 수 있도록 하기 위해서 테스트 케이스를 생성할 때 요구사항 추적 표(Requirement Traceability Matrices)를 작성한다. 요구사항 추적 표는 테스트 케이스와 요구사항의 연관관계가 매우 중요하다는 것에 대한 반증이다. 그러나 m:n 관계를 맺고 있는 모델로부터 테스트 케이스를 생성하면 요구사항 추적 표는 사실상 무의미해진다. 하나의 테스트 케이스가 매우 많은 요구사항과 관계를 맺고 있을 뿐만이 아니라 심지어는 하나의 요구사항이 모든 테스트 케이스와 관계를 맺는 경우도 생기기 때문이다.

요구사항은 제품의 개발 기간에 걸쳐 지속적으로 변경(생성/수정/삭제)된다. 고객의 변심, 고객의 요구를 잘 못 이해한 요구사항, 다른 외부 환경에 의한 제약조건 등이 그 이

† 준 회 원 : King's College London 방문 연구원
†† 정 회 원 : 아주대학교 정보통신전문대학원 교수
††† 정 회 원 : 아주대학교 전자공학부 교수

논문접수 : 2009년 10월 27일
수 정 일 : 1차 2009년 12월 4일, 2차 2009년 12월 11일
심사완료 : 2009년 12월 29일



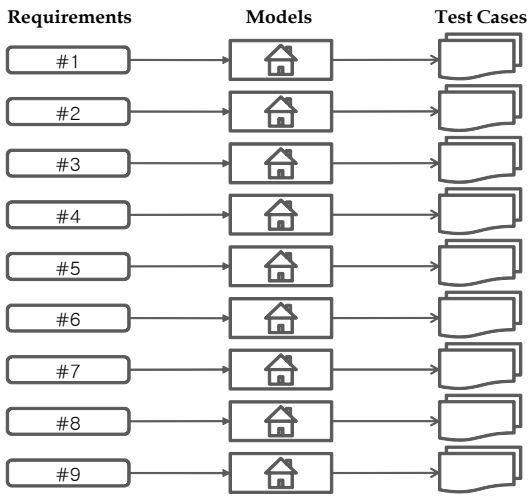
(그림 1) 요구사항과 모델간의 m대n 관계

유가 될 수 있다. 요구사항 관리란 제품 개발 기간에 걸쳐 요구사항이 변경되는 것을 관리하는 것이다. 그러나 테스트 케이스과 요구사항과의 관계가 매우 복잡하게 얽혀 있다면, 요구사항 관리는 매우 복잡한 일이 될 수밖에 없다. 따라서 요구사항과 테스트 케이스의 관계를 간단하게 만들 필요성이 있다.

요구사항과 테스트 케이스의 관계가 (그림 2)와 같이 단순화 될 경우 다음과 같은 장점을 얻을 수 있다.

(1) 테스트 케이스가 하나의 요구사항에 대하여 생성되므로 해당 요구사항을 충분히 테스트할 수 있는지를 테스트 엔지니어가 직접 판단할 수 있다. 따라서 자동 생성으로 말미암은 미비점을 테스트 엔지니어가 쉽게 보완할 수 있다.
(2) 테스트 결과를 바탕으로 제품에 제대로 반영되지 않은 요구사항을 정확히 찾을 수 있으므로 제품의 오류 수정을 쉽게 한다.
(3) 요구사항 변경(추가/수정/삭제) 시에 해당 테스트 케이스 만을 재생성하면 되므로 테스트 케이스 생성 시간 및 테스트 수행 시간이 단축된다.

REED(REquirement EDitor)는 요구사항과 모델 사이의 관계를 1:1로 만들어 주는 유일한 언어이다[8]. REED로 생성된 모델로부터 테스트 케이스를 생성하면 요구사항과 테스트



(그림 2) 요구사항과 모델간의 1대1 관계

케이스의 관계를 (그림 2)와 같이 매우 단순화할 수 있다.

REED를 이용하여 요구사항을 1:1 관계로 모델링 하는 방법에 대하여는 이전 논문[8]에서 설명하였으므로 본 논문에서는 REED를 기반으로 하여 테스트 케이스를 자동으로 생성하는 방법에 대하여 제안한다.

본 논문은 구성은 다음과 같다. 2절에서는 모델을 기반으로 하여 테스트 케이스를 자동으로 생성하는 연구들에 대하여 살펴보고, 3절에서는 REED에 대한 간략한 소개와 REED를 기반으로 하여 테스트 케이스를 자동으로 생성하는 방법을 기술한다. 4절에서는 테스트 케이스를 이용하여 실제로 실험한 실험결과를 제시하며 끝으로 5절에서는 결론과 향후 과제에 대하여 기술할 예정이다.

2. 관련 연구

요구사항을 모델로 표현하는 방법 중에서 가장 많이 사용되는 방법은 UML[9]과 Simulink/Stateflow[10]이다. 그러나 이러한 모델링 언어들은 모두 m:n 모델링 언어이다. 즉, 요구사항을 모델로 표현하기 위해서 요구사항을 재조합하고 재해석하는 일을 수행해야 한다. 이러한 과정은 테스트 케이스와 요구사항과의 관계를 복잡하게 할 뿐만이 아니라, 요구사항이 모델링 하는 과정에서 변질할 가능성을 내포하고 있다. 그러나 아직 1:1 모델링 언어를 통하여 테스트 케이스를 자동으로 생성하는 기법은 알려진 바가 없다. 따라서 이 절에서는 대표적인 모델링 언어인 UML과 Simulink/Stateflow를 기반으로 테스트 케이스를 자동으로 생성하는 기법에 대하여 기술한다.

UMLTest는 IBM의 Rational Rose에서 생성한 UML MDL 파일로부터 테스트 케이스를 자동으로 생성한다[4]. Use cases, object collaboration, class, statechart diagram 등이 테스트 케이스 생성에 관여한다. UMLTest는 UML의 statechart diagram을 기반으로 하여 테스트 케이스를 생성하기 위해서 미리 정의한 use cases, object collaboration, class diagram 등을 사용한다. Conformiq Qtronic도 UML을 기반으로 테스트 케이스를 자동 생성한다[11]. Conformiq Qtronic는 semantics-driven[12]을 사용하여 Java-compatible source file, statechar, class diagram을 기반으로 테스트 케이스를 생성한다. 즉, 언어적인 분석이나 단순한 사용자의 지식에 근거한 것이 아니라 모델이 표현하는 행동을 분석하여 테스트 케이스를 생성한다. UMLTest나 Conformiq Qtronic은 UML로부터 테스트 케이스를 자동으로 생성하지만, 여러 다이어그램에 요구사항이 분산되어 표현된다는 단점을 가지고 있다. 요구사항이 여러 개의 다이어그램에 나뉘어 있으면, 하나의 요구사항이 변경되었을 때 여러 개의 다이어그램을 수정해야 하기 때문에 관리하기가 쉽지 않다. 이는 UML 모델링 기법의 단점이기도 하다[13].

Simulink/Stateflow는 임베디드 시스템을 모델링하기 위해서 많이 사용되는 도구이다. 이를 기반으로 테스트 케이스를 생성하는 상용 도구로는 Reactis System Inc.의 Reactis

[14], TNI Software의 Safety Test Builder(STB)[15], Applied Dynamics International의 BEACON Tester[16], T-VEC technologies의 T-VEC tester[17], Mathworks의 Design Verifier[18] 등이 있다. 이들 중 가장 대표적인 Reactis는 수동 생성이나 랜덤 생성 방식을 지원한다. Reactis로 자동 생성되는 테스트 케이스는 중복된 테스트 케이스 생성을 방지하는 한편, 주어진 테스팅 시간 내에서 최대한 많은 결함을 발견할 수 있는 테스트 케이스를 생성한다. 생성된 테스트 케이스가 만족하는 커버리지도 분석한다.

UML이나 Simulink/Stateflow 모델 외에 다른 모델로부터 테스트 케이스를 생성하는 시도도 있다. High-level Petri-nets 모델로부터 테스트 케이스를 생성하기 위하여 Cause-effect graphing을 이용하는 방법이 사용된다[6]. 요구사항을 high-level Petri-nets으로 모델링을 하고, 이를 이진 그래프를 이용하여 입력과 출력의 관계를 Cause-effect graph로 변환한다. 각각의 effect(출력)에 대하여 뒤에서부터 추적하여 cause(입력)의 모든 조합을 찾는 방법으로 테스트 케이스를 생성한다. 이러한 방식은 프로그램의 최종 결과에 대한 요구사항만을 테스트하기 때문에 일종의 블랙박스 테스트이다.

EFSM(Extended Finite State Machine)을 기반으로 테스트 케이스를 생성하는 방법은 control-flow testing과 data-flow testing을 병합하여 테스트 케이스를 생성한다[7]. 테스트 케이스를 생성하는 과정에서 미리 실행가능성을 검사하기 때문에 실행할 수 없는 테스트 케이스가 생성되는 것을 방지한다는 장점이 있다.

3. REED (REquirement EDitor) 개요

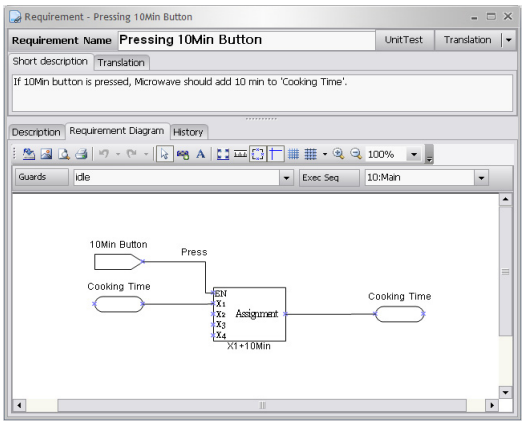
REED(REquirement EDitor)는 요구사항과 모델 사이의 관계를 1:1로 만들어 주는 유일한 언어이다. REED는 하나의 요구사항을 직관적인 의미가 있는 graphic notation을 이용하여 모델링할 수 있도록 하는 언어이다[8]. REED로 생성된 모델은 요구사항과 1:1 관계를 맺고 있으므로 REED로부터 생성된 테스트 케이스는 요구사항과 단순한 관계를 맺게 된다.

REED를 이용하여 요구사항을 1:1 관계로 모델링 하는 방법에 대하여는 이전 논문[8]에서 설명하였으나 이 절에서는 본 논문을 이해하는 데에 도움될 수 있도록 개략적인 내용에 대하여만 설명한다.

Graphic으로 표현하는 표현 기법이나 사용된 graphic object 측면에서 볼 때, REED는 Simulink/Stateflow와 매우 흡사한 모습을 가진다. 그러나 Simulink/Stateflow는 요구사항을 전체적으로 모델링해야하는 m:n 모델링 언어이며 요구사항을 모델링하기 위해서 시스템에 대한 상세한 정보를 바탕으로 “how-to-do”를 모두 기술해야 하는 도구라면, REED는 하나의 요구사항을 하나의 모델로 모델링하는 1:1 모델링 언어이며 시스템이 무엇을 해야 하는지에 대한 요구사항에 집중하여 “what-to-do” 만을 기술한다. REED는 상위레벨의 요구사항의 표현은 물론 하위 레벨 요구사항 개발 단계까지 기술하거나 검증하기 위한 목적으로 개발되었다는 점에서 모델링 및 시뮬레이션을 목적으로 개발된 Simulink/Stateflow와 다르다.

(그림 3)은 REED에서 전자레인지의 요구사항 중 하나를 작성하는 화면의 모습이다. 전자레인지 요구사항은 현재 판매되고 있는 제품의 매뉴얼을 기반으로 REED를 이용하여 모델링한 것이다. 화면 위의 ‘Requirement Name’에 기술된 ‘Pressing 10Min Button’은 이 요구사항의 이름이다. (그림 3)의 아래에 보이는 모델은 ‘If 10Min button is pressed, Microwave should add 10 min to ‘Cooking Time’. (10Min 버튼을 누르면, 전자레인지는 ‘Cooking Time’에 10분을 더해야 한다.)’는 자연어 요구사항을 graphic 언어로 모델링한 것이다. (그림 3)에서 보는 바와 같이, graphic 언어로 표현한 모델은 자연어 요구사항을 그대로 모델링한 것이다. 이처럼 REED에서는 요구사항과 요구사항 모델이 1:1 관계를 맺게 된다.

<표 1>은 (그림 3)의 요구사항 모델에 사용된 객체들의 의미를 가리킨다. REED는 이 이외에도 요구사항 문장을 표현하기 위한 다양한 graphic notatation 을 지원한다. REED



(그림 3) REED에서의 요구사항에 대한 요구사항 모델

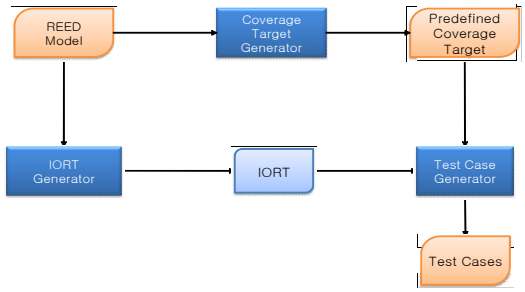
〈표 1〉 REED의 graphic notation의 예

기호	이름	의미
<이름>	입력장치	시스템의 입력을 표현한다. 예를 들면, 버튼, 센서 값 등이 이에 속한다. 입력장치의 이름은 <이름>이다.
<이름>	메모리	요구사항의 기술을 쉽게 하기 위해서 사용되는 변수이다. 변수의 이름은 <이름>이다.
EN X1 X2 X3 X4 Assignment	Assignment	EN 포트로 어떤 값이 입력될 때, 입력 포트 X1~X4으로 입력된 값을 사용하여 함수를 실행하고 나서, 그 값을 출력 포트로 출력한다.

에서 객체는 엔티티 객체와 연산 객체로 구별된다. 입력장치, 출력장치, 메모리 등의 객체를 엔티티 객체로 분류하고, 'Assignment'와 같이 어떠한 기능을 수행하는 객체는 연산 객체로 분류한다. 각 객체는 서로의 연결을 위한 포트가 있다. 엔티티 객체와 연산 객체는 연산 객체의 포트로 연결된다. 연산 객체의 출력은 다른 연산 객체의 입력 포트와 연결을 할 수 있다. Graphic language에 해당하는 요구사항 모델에서 연산 객체들은 엔티티 객체로부터 입력을 받아 다른 엔티티 객체 혹은 연산 객체로 값을 출력하는 그래프의 형태를 보인다.

4. 자동 테스트 케이스 생성

테스트 케이스를 생성하는 과정은 (그림 4)의 기본구조를 따른다. REED 모델로부터 테스트 케이스를 생성하기 위하여 우선은 커버리지 타겟 생성기(Coverage Target Generator)를 통하여 커버리지 타겟을 생성한다. 생성된 커버리지 타겟은 이후 테스트 케이스 생성기가 테스트 케이스를 생성하는 데 사용한다. 또한, REED 모델을 분석하여 IORT(Input-Output Relation Tree)를 생성한다. IORT는 테스트 케이스를 최종적으로 생성하기 위한 중간단계의 결과물이다. 모델의 입력과 출력 간의 연관관계를 분석하여 트리 형태로 만든 것이다. 생성된 IORT를 기반으로 하여 커버리지 타겟을 이용하여 테스트 케이스를 생성하면 원하는 커버리지에 맞는 테스트 케이스를 생성할 수 있다.



(그림 4) REED를 위한 테스트 케이스 생성 기본 구조

4.1 커버리지 타겟 생성

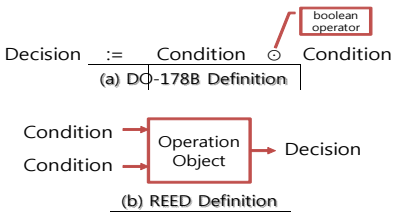
테스트 케이스의 품질은 커버리지에 의해서 결정된다. 테스트 케이스를 자동으로 생성하는 것만이 중요한 것이 아니라, 어떠한 커버리지를 만족하는 테스트 케이스를 생성하는가가 중요한 문제이다. 커버리지 타겟 생성기를 이용하는 구조에서는 다양한 커버리지를 만족하는 테스트 케이스 생성이 쉽다. 커버리지 타겟 생성기를 조금만 수정하면 다양한 커버리지를 만족하는 테스트 케이스를 생성할 수 있다. 이 논문에서는 REED로 표현된 요구사항으로부터 MC/DC [19] criteria를 만족하는 테스트 케이스를 생성하는 방법에 대해 설명한다.

MC/DC(Multiple Condition/Decision Coverage)는 프로그램의 소스 코드에서 나타나는 condition이나 decision 값들에 대한 테스트 케이스를 생성할 때 사용하는 커버리지 척도이다. 그러나 MC/DC는 항공기와 같은 매우 치명적인 소프트웨어를 테스트하기 위한 커버리지로 사용될 정도의 정밀한 테스트를 수행할 수 있는 테스트 케이스 커버리지이다. 구조적 테스팅에 적합하도록 정의된 MC/DC를 요구사항에 적용하기 위해서는 요구사항에 적합하도록 MC/DC의 개념을 확장할 필요가 있다.

DO-178B에 나와있는 MC/DC의 정의를 따르면, condition은 boolean operator를 포함하지 않은 boolean expression이고, decision이란 하나 이상의 boolean operator가 포함된 boolean expression이다[19]. Condition과 decision의 가장 큰 차이는 boolean operator의 유/무이다. 즉, decision이란 condition들을 boolean operator로 연산한 결과이다.

우선, MC/DC를 REED로 표현되는 다양한 요구사항에 적용하기 위하여 condition과 decision의 개념을 다음과 같이 확장하였다. (그림 5)의 (a)는 전형적인 MC/DC의 decision과 condition의 관계를 보여준다. 입력과 출력을 기준으로 요구사항을 작성하는 REED에서는 (그림 5)의 (a) 와 같은 요구사항이 (그림 5)의 (b)와 같이 표현된다. 즉, 연산객체(Operation Object)의 입력은 condition으로, 연산객체의 출력은 decision으로 간주할 수 있다. REED에 정의된 연산객체는 operator와 같은 기능을 하고, 연산객체의 입력은 operand와 같은 구실을 하기 때문이다. 따라서 각각의 연산객체에 대하여 연산객체의 입력은 condition, 출력은 decision에 대응시킬 수 있다.

전통적인 MC/DC의 정의에서는 condition은 이진값을 가진다. 즉, condition은 참/거짓만을 가진다. 그러나 REED의 graphic notation에서 입출력은 2가지 종류의 정보를 동시에 가진다: 하나는 값의 유효성을 가리키는 활성화(activation) 정보이고, 다른 하나는 활성화되었을 때 실제로 입/출력되는 값(value)이다. REED에서 사용되는 입출력 엔티티의 값은 이진값(참/거짓) 만을 가지지는 않고 정수형 값, 열거형 값 등 여러 가지 형태의 값을 가질 수 있다. 반면, 활성화 정보는 이진값과 유사하게 활성화 상태 혹은 비활성화 상태의 값을 가질 수 있다. 만일 활성화 상태라면, 연결된 값이 전달되고, 비활성화 상태라면 연결된 객체로 값이 전달되지 않는다. 따라서 활성화되는 경우를 이진값인 '참'을 갖는 경우로 간주하고, 비활성화되는 경우를 이진값인 '거짓'을 갖는 경우로 간주할 수 있다. 이렇게 하면 REED의 condition을 마치 이진값을 가지는 것으로 취급할 수 있다. 따라서 일차적으로 활성화 정보에 기반을 두어 테스트 케이스를 생성한



(그림 5) MC/DC에 대한 DO-178B 정의와 REED 정의

다. 이렇게 하면 전통적인 MC/DC의 결과와 매우 유사한 결과를 얻을 수 있다.

MC/DC 정의에 의하면 각각의 condition들이 가질 수 있는 모든 값이 테스트 케이스에 나타나야 한다. 따라서 단순히 활성화 정보에만 의존하는 것이 아니라 실제 값에 대해서도 테스트 케이스가 생성되어야 한다. 즉, 활성화 정보 뿐만이 아니라, 입력 엔티티(condition)가 가질 수 있는 값을 고려하여 테스트 케이스가 생성되어야 한다. 만일 엔티티 객체의 타입이 어떤 범위 내의 정수형일 경우에 각 범위를 만족하게 하는 대표값을 사용한다. 즉, 대표값은 해당 엔티티 객체가 테스트 되어야 하는 값이다. 예를 들어, 정수형의 A라는 엔티티 객체의 범위가 (5 < A < 11)이라면 대표값은 6, 8, 10이 선택 될 수 있고, A의 값 6, 8, 10을 가지고 생성된 테스트 케이스를 생성해야 한다.

다음의 <표 2>는 REED의 대표적인 3개의 graphic object에 대하여 각각의 연산 객체에 대하여 MC/DC를 만족하는 테스트 케이스를 정의한 것이다. REED는 약 20여 개의 graphic notation을 지원하고 있다. 이렇게 각각의 연산 객체에 대하여 MC/DC를 만족하는 커버리지 타겟을 생성하고 이를 이용하여 하나의 요구사항에 대한 테스트 케이스를 생성한다.

4.2 IORT(Input Output Relation Tree) 생성

하나의 요구사항 모델은 여러 개의 엔티티 객체와 연산 객체가 서로 연결된 다이어그램의 모양을 가지고 있다. 전자레인지의 'Start Button'이 눌려졌을 때의 요구사항을 하나의 모델로 표현한 다음의 (그림 6)을 예로 들어 보자.

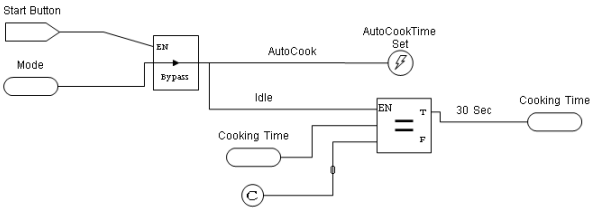
이 요구사항 모델은

"사용자가 'Start Button'을 눌렀을 때, 전자레인지의 'Mode'가 'AutoCook'으로 설정되어 있으면 조리 시간을 자동 조리에 적합하도록 설정하고, 'Mode'가 'idle'로 설정되어 있을 때, 만약 설정된 'Cooking Time'이 0이라면 'Cooking Time'을 30sec로 설정한다."

라는 요구사항을 모델링한 것이다. 요구사항 모델에 사용된 객체의 의미는 <표 3>과 같다.

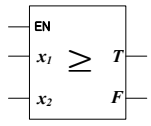
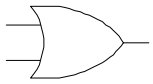
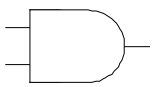
(그림 6)의 요구사항 모델에는 'AutoCookTimeSet'과 'Cooking Time' 라는 두 개의 출력이 있다. 'AutoCookTimeSet'을 출력하기 위하여 'Cooking Time' 앞에 있는 Equal 객체는 필요가 없다. 그러나 'Bypass'라는 객체는 'AutoCookTimeSet'과 'CookingTime'을 출력하기 위하여 모두에게 필요하다. 모델을 각각의 출력에 영향을 미치는 객체 들로만 연결된 다이어그램으로 분리할 수 있다. 이렇게 각각의 출력에 영향을 미치는 객체로만 연결된 다이어그램을 IORT(Input-Output Relation Tree)라고 한다. (그림 6)을 출력 별로 관계되는 객체만을 추출하면 (그림 7)과 같이 2개의 IORT를 만들 수 있다. 각각의 IORT는 해당 출력이 어떤 객체를 거쳐서 생성되는지 명확하게 보여주고 있다. IORT는 출력을 기준으로 한 전통적인 Cause-Effect Relation을 기반으로 생성한다.

(그림 6)과 같은 요구사항 모델로부터 분할된 IORT는 각 출력 엔티티 객체에서부터 거꾸로 다이어그램을 탐색하여 구한다. 출력 엔티티 객체 'AutoCookTimeSet'부터 거꾸로 탐색을 시작하여 마지막 연산 객체인 'Bypass' 객체에서 종료하면, (그림 7)의 위쪽에 도시된 IORT 가 생성된다. 또한, 출력 엔티티 객체인 'Cooking Time'에서 출발하여 거꾸로



(그림 6) 전형적인 요구사항 모델 (전자레인지 예제)

〈표 2〉 Examples of Predefined Coverage Targets for each block

기호	이름	의미	커버리지 타겟
	GTE	EN 포트가 활성화되었을 때, 입력 x_1이 x_2보다 크거나 같으면 출력 T를 활성화하고, 그렇지 않으면 출력 F를 활성화한다.	*(1) for ∀x_1, input x_2 set at a value below x_1* *(2) for ∀x_1, input x_2 set at a value above x_1* *(3) for ∀x_2, input x_1 set at a value below x_2* *(4) for ∀x_2, input x_1 set at a value above x_2* *(5) for ∀x1, input x_2 set at a value slightly below x_1* *(6) for ∀x1, input x_2 set at a value slightly above x_1* *(7) for ∀x1, x_1 is equal to x_2*
	Flow-OR	Input1이나 Input2가 활성화되면 출력을 활성화한다.	(1) Input 1: Activate(all), Input 2: Deactivate(one) (2) Input 1: Deactivate(one), Input 2: Activate(all) (3) Input 1: Deactivate(all), Input 2: Deactivate(all) ※괄호 속의 (all)은 해당 대표값 모두를 대입해야 함을 의미함. 즉, "Activate(all)"은 Input을 activate할 수 있는 모든 대표값을 다 넣어야 함을 의미함. 반대로 (one)은 대표값 중에서 하나만 넣는 것을 의미함.
	Flow-AND	Input1와 Input2가 모두 활성화되면 출력을 활성화한다.	(1) Input 1: Activate(all), Input 2: Activate(all) (2) Input 1: Activate(one), Input 2: Deactivate(all) (3) Input 1: Deactivate(all), Input 2: Activate(one)

〈표 3〉 (그림 6)에 사용된 REED 객체의 의미

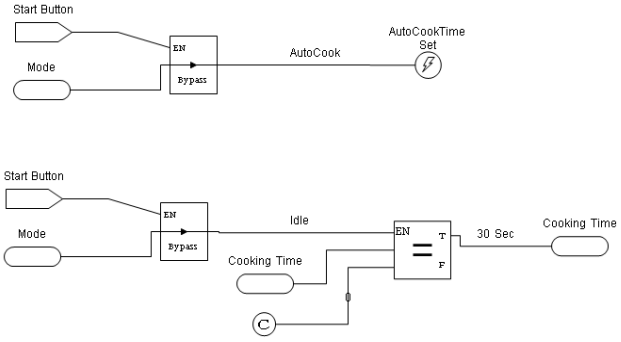
기호	이름	의미
	Bypass	EN 포트가 활성화되었을 때, 입력으로 들어온 값을 출력으로 내보낸다.
	Equal	EN 포트가 활성화되었을 때, 입력으로 들어온 2개의 값이 같으면 출력 T를 활성화하고, 입력 값이 서로 다르면 출력 F를 활성화한다.
	Internal Event	<name>이라는 이벤트를 발생시킨다.
	Constant	연결된 링크에 적힌 상수 값을 의미한다.

탐색하면, (그림 7)의 아래쪽에 도시된 IORT가 만들어진다.

그러나 전통적인 Cause-Effect Relation만을 이용하여서는 IORT를 생성할 수 없는 요구사항 다이어그램도 존재한다. 반복 작업을 표현하기 위한 'Loop' 객체나 순차적인 작업을 표현하기 위한 'Sequential' 객체를 포함한 모델에서는 IORT를 추출하는 알고리즘을 설계하기가 약간 복잡하다. 설명의 편의를 위하여 'Loop'과 'Sequential' 객체를 FC(Flow of Control) 객체라 부른다. <표 4>는 이 두 객체의 의미를 기술하고 있다.

'Sequential' 객체의 경우는 각 포트에 연결된 출력들이 차례대로 발생해야 하는 것을 명시하고 있다. 'Sequential' 객체에 연결된 출력들은 이전 출력이 발생한 후에 다음 출력이 발생하기 때문에 서로 연관성을 가진다. 따라서, 각 포트에 연결된 출력들이 하나의 IORT로 생성되어야 한다. 서로 다른 포트에 연결된 객체들은 거꾸로 탐색할 경우 하나의 IORT로 연결될 수 없지만, FC 객체를 만날 경우는 하나의 IORT로 연결해 주어야 한다. FC 객체를 포함한 요구사항 모델로부터 IORT들의 리스트로 만드는 알고리즘의 C# 스타일의 가상 코드는 (그림 8)과 같다.

(그림 7) (그림 6)의 요구사항 모델으로부터 생성된 2개의 IORT



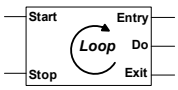
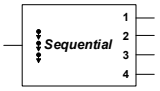
우선, 요구사항 모델의 최종 출력 리스트를 만들고서, 이 리스트 내의 각각 객체 "L"에서 출발하여 거꾸로 탐색하면서 "L"을 생산하는 객체들이 사용되는 값을 출력하는 객체들을 찾는다. 거꾸로 탐색하는 도중, 'Loop'나 'Sequential' 등의 FC 객체를 만나면, 이들을 사용하여 이미 어떤 IORT가 만들어졌는지를 확인하기 위하여 "SetG"라는 IORT 집합을 검색한다. 이미 만들어진 IORT는 작업 중이던 IORT에 병합시켜 하나의 IORT로 구축하고, 작업을 계속한다. 이러

```
// IORT Generator
GraphSet GenIORT (Graph ReqDiag)
{
    Let SetG be an empty set
    ListofLastOP = List of last operation objects of ReqDiag

    foreach (L in ListofLastOP) {
        1)mark all objects ∈ ReqDiag as unvisited
        2)Create an IORT G, and initially L∈ G, mark L as visited
        3)while(there is an unvisited object Obj that is researchable by
          backward traverse from an object already visited) {
            (1)mark Obj as visited
            (2)if(Obj is 'Sequential' or 'Loop' object and Obj is already
                used by other IORT F in SetG) {
                    Set all objects ∈ F as visited
                    Merge F into G
                    remove F from SetG
                }
                else {
                    Expand G using Obj
                }
        4)Insert G into SetG
    }
    return SetG
}
```

(그림 8) IORT Generator를 위한 알고리즘

〈표 4〉 FC(Flow of control) 객체

기호	이름	의미
	Loop	'Loop'은 같은 작업이 반복되는 것을 표현할 때 사용하는 객체이다. 'Start' 포트로 입력되면, 'Entry'에 연결된 작업을 수행하고 나서, 'Do'에 연결된 작업을 'Stop'으로 입력이 발생할 때까지 반복한다는 것을 의미한다. 'Stop'에 입력이 발생하면, 'Exit'에 연결된 작업을 수행하고, 'Loop'은 종료된다.
	Sequential	'Sequential'은 일련의 행동을 차례대로 기술할 때 사용된다. 각 출력 포트에 연결된 작업을 번호순으로 차례대로 수행하라는 것을 가리킨다.

한 작업을 다이어그램 내의 모든 객체가 적어도 하나의 IORT에서 사용될 때까지 계속된다.

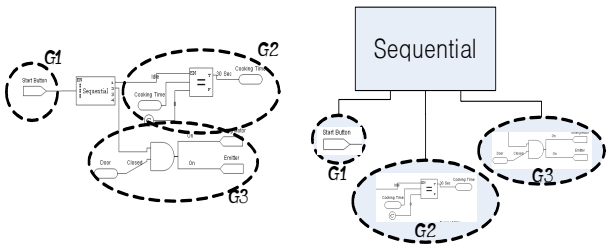
4.3 테스트 케이스 생성

앞에서 생성된 IORT로부터 테스트 케이스를 생성한다. IORT를 입력에서 출력방향으로 탐색하면서 각각의 객체에 대하여 미리 정의한 커버리지 타겟에 따라 생성한다. 특정 객체의 테스트 케이스 결과는 그 객체와 연결된 다른 객체의 테스트 케이스의 입력으로 사용된다. IORT를 차례대로 순회하면서 각각의 객체에 정의된 테스트 케이스를 이용하여 전체 테스트 케이스를 생성한다.

객체를 순회하는 순서는 기본적으로는 위상 정렬을 사용한다. 그러나 FC 객체가 IORT에 포함되어 있으면 FC 객체를 기준으로 순서를 조정한다. FC 객체는 각 출력 포트의 출력에 순서가 부여되기 때문에 FC 객체의 첫 번째 출력 포트와 연결된 모든 객체에 대한 테스트 케이스의 생성이 끝나고 나서 다음 출력 포트와 연결된 객체에 대한 테스트 케이스를 생성해야 한다. 위상 정렬만 이용하면, FC 객체 이후의 객체들에 대한 순서를 보장하지 못한다. 따라서, FC 객체를 중심으로 간단한 트리를 생성하여 순서를 조정한다. 이 트리는 IMT(InterMediateTree)로 부르며, FC 객체나 FC 객체에 연결된 모든 다른 객체 그룹을 노드로 갖는다.

(그림 9)의 왼쪽에 있는 다이어그램은 FC 객체인 'Sequential' 객체에 모든 출력이 연결되어 있기 때문에 IORT가 1개만 생성이 된다. 즉, 모델과 IORT가 같은 모양을 가진다. 이 모델에 대한 IMT를 생성하면 (그림 9)의 오른쪽에 있는 그림과 같은 결과가 생성된다. FC 객체를 중심으로 그와 연결된 객체 그룹이 트리의 자식으로 연결된다. (그림 9)의 오른쪽과 같은 IMT를 생성하고, FC 객체의 입력(G1), FC 객체(Sequential), FC 객체의 출력(G2, G3) 순으로 순회한다. 각각의 서브그래프는 앞에서 설명한 바와 같이 간단한 위상 정렬에 의해서 정렬된 순서를 사용한다. IORT를 이용하여 테스트 케이스를 생성하는 전체적인 알고리즘은 (그림 10)과 같다.

(그림 10)의 generateTC 함수는 IORT를 입력으로 받는다. 우선 입력된 IORT를 IMT로 변환한다(Covert_IORT2IMT). IMT의 루트가 FC 객체일 경우는 FC 객체의 입력(1), FC 객체(2), FC 객체의 출력(3) 순으로 각각의 서브그래프에 대해서 테스트 케이스를 생성하고, IMT의 루트가 FC 객체가 아닐 경우는 IMT 자체가 서브그래프 1개로 이루어진 트리



(그림 9) IMT 및 서브그래프의 예

이므로 루트에 대해서 테스트 케이스를 생성한다(MCDC_TCGen).

(그림 9)와 같이 IORT를 IMT로 변환하는 Convert_IORT2IMT 함수의 가상 코드는 (그림 11)과 같다. IORT에서 FC 객체를 찾아 FC 객체의 입력과 출력에 연결된 모든 객체를 각각의 서브그래프로 묶어서 FC 객체의 자식 노드로 만든다. 출력에 연결된 객체 중에서는 다시 FC 객체가 나타날 수 있으므로 객체에 연결된 서브그래프에 대해서는 다시 Convert_IORT2IMT 함수를 재귀적으로 호출하여 트리를 생성한다.

(그림 10)의 MCDC_TCGen 함수는 FC 객체가 존재하지 않는 서브그래프에 대하여 MC/DC를 만족하는 테스트 케이스를 생성하는 함수이다. 이 함수의 알고리즘은 (그림 12)와 같다. 먼저 서브그래프를 위상 정렬하여 객체의 순서를 결정하고 순서대로 Coverage Generator에서 생성한 MC/DC를 만족하는 커버리지 타겟을 이용하여 테스트 케이스를 생성한다. 이때 각 객체마다 정의된 do_MCDC 함수를 호출한다. do_MCDC 함수는 각 block의 의미에 맞게 생성된 커버리지 타겟을 가져오는 함수이다.

```
// Test Case Generator
TestCaseTable generateTC (IORT ioTree)
{
    IntermediateTree IMT = Convert_IORT2IMT (ioTree);
    if (IMT.rootnode.type == FC_Object) {

        // (1) Test cases for input nodes of FC Object
        foreach (subGraph in IMT.rootnode.input)
            MCDC_TCGen(subGraph);

        // (2) Test case for FC Object
        IMT.rootnode.do_MCDC();

        // (3) Test cases for output nodes of FC Object
        foreach (subGraph in IMT.rootnode.output)
            MCDC_TCGen(subGraph);
    }
    else if (IMT.rootnode.type == subgraph)
        MCDC_TCGen(IMT.rootnode);
}
```

(그림 10) IORT로부터 테스트 케이스 생성 알고리즘

```
// Generate IMT from IORT
IntermediateTree Convert_IORT2IMT (IORT G)
{
    Let IMT as a Empty Intermediate Tree
    FC = Find 'Loop' or 'Sequential' object in G
    if FC is not found {
        Initialize IMT with G
    }
    else {
        1)Set FC as a root node of IMT
        2)Make Gin, connected to each input port of FC, as a child of FC
        3)Make Gout, connected to each output port of FC, as a child of FC
        4)Call Convert_IORT2IMT (Gout)
    }
    return IMT
}
```

(그림 11) IMT 생성 알고리즘

```
// Generate test case with graph
void MCDC_TCGen (ReqDiag G)
{
    topoOrderList = topologicalSort(G);
    foreach (obj in topoOrderList)
        obj.do_MCDC(G);
}
```

(그림 12) 서브그래프로부터 테스트 케이스 생성 알고리즘

5. 예제 및 실험 결과

5.1 테스트 케이스 생성 예제

단일 요구사항에 대한 테스트 케이스 생성 방법을 다음의 (그림 13)을 예로 살펴보자.

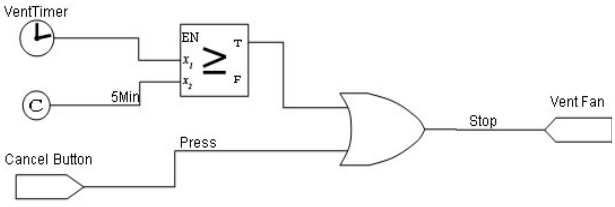
(그림 13)은 하나의 IORT를 가지는 요구사항 모델이다. 또한, FC 객체가 포함되지 않았기 때문에, 하나의 루트 노드 만을 가진 IMT가 생성된다. (그림 13)의 요구사항 모델을 위상 정렬에 의해서 정렬하면서 차례로 테스트 케이스를 생성한다. 입력 엔티티 객체에 대한 대표값은 요구사항을 작성할 때 입력할 수도 있고, 입력된 값이 없다면 domain theory를 기반으로 자동으로 추가한다. (그림 13)의 입력 엔티티 객체의 정보는 다음의 <표 5>와 같다.

위상 정렬에 의해서 가장 먼저 나오는 연산객체는 GTE (≥) 객체이다. 이 객체의 커버리지 타겟은 <표 2>에 정의된 내용을 참고해서 생성한다. x_2는 고정된 상수 값이기 <표 2>의 (1), (2)는 해당 사항이 없다. 따라서 x_1에 대한 대표값 만을 변화시켜 테스트 케이스를 생성한다. 생성된 테스트 케이스는 다음의 <표 6>과 같다. 표의 비고란에는 <표 2>에 정의된 번호를 넣었다. 또한, 표의 T(Output)란에는 GTE 객체의 출력 결과를 예측하여 생성하였다. 이처럼 각각의 연산객체마다 커버리지 타겟을 정하고, 그 타겟을 이용하여 값을 변화시키면서 테스트 케이스를 생성한다.

이제 GTE 객체에 대한 테스트 케이스 생성이 완료되었으므로 다음 연산객체에 대하여 테스트 케이스를 생성한다. 다음 연산 객체는 Flow-OR 객체이다. Flow-OR 객체에 대한 커버리지 타겟은 역시 <표 2>에 정의되어 있다. 그런데 이 Flow-OR 객체의 첫 번째 입력이 GTE 객체의 출력이다. 따라서 <표 6>에서 생성한 테스트 케이스 테이블을 입력으로 이용하여 Flow-OR 객체의 테스트 케이스를 생성하면 <표 7>과 같다. <표 7>의 Input 1은 (그림 12)의 Flow-OR 객체의 위쪽 입력이고, Input 2는 Flow-OR 객체의 아래쪽 입력을 의미한다.

<표 2>의 (1)번 테스트 케이스는 Input 1의 값이 'Activate (all)'로 정의되어 있다. 이것이 <표 7>에서는 TC #1~4까지로 생성되었다. 즉, <표 7>의 Input 1이 GTE객체의 출력이기 때문에 <표 6>의 T(Output)에서 'Activate'값을 가지는 모든 테스트 케이스를 (1)번 테스트 케이스로 대치하는 것이다. 반면에 Input 2는 <표 2>에 "Deactivate(One)"이라고 되어 있으므로 Input 2를 'Deactivate'할 수 있는 값 중의 하나인 'Release'를 대입하였다. Input 2를 'Activate'할 수 있는 값은 'Press' 하나밖에 없어서 Flow-OR의 (2)번 테스트 케이스를 생성하기 위한 실제 테스트 케이스는 1개만 생성이 되었다. 같은 이유로 Flow-OR의 (3)번 테스트 케이스를 생성하기 위한 실제 테스트 케이스도 1개만 생성이 되었다.

이제 더 처리할 연산객체가 없으므로 최종 결과인 <표 7>에서 Input 1, Input 2 값을 실제 입력 엔티티의 이름으로 바꾼 최종 테스트 케이스의 결과는 <표 8>과 같다. <표 7>에서 Flow-OR 객체의 Input 1은 GTE 객체를 의미하고 Input 1의 x_1은 VentTimer, Input 1의 x_2는 상수(Constant)이다. 상수는 테스트 케이스에서 입력으로 줄 수 없는 것이므로 제외한다. 마찬가지로 Flow-OR 객체의 Input 2는 'Cancel Button'에 연결되어 있으므로 <표 8>과 같이 정리



(그림 13) 전자레인지 규격 중 하나 요구사항에 대한 요구사항 모델

〈표 5〉 (그림 13)의 입력 엔티티 객체의 정보

이름	객체 타입	값 타입	값 범위	대표값
VentTimer	Memory	Integer	0 ~ 999	0, 5, 60, 999, 1000
Cancel Button	Device	Enum	Press, Release	Press, Release

〈표 6〉 GTE 객체의 테스트 케이스

TC #	x_1	x_2	T(Output)	비고
1	0	5	Deactivate	(3)(5)
2	60	5	Activate	(4)(6)
3	999	5	Activate	(4)
4	1000	5	Activate	(4)
5	5	5	Activate	(7)

〈표 7〉 Flow-OR 객체의 test case

TC #	Input 1		Input 2	T(Output)	비고
	x_1	x_2			
1	60	5	Release	Activate	(1)
2	999	5	Release	Activate	(1)
3	1000	5	Release	Activate	(1)
4	5	5	Release	Activate	(1)
5	0	5	Press	Activate	(2)
6	0	5	Release	Deactivate	(3)

〈표 8〉 (그림 12)에 대한 최종 테스트 케이스

TC #	VentTimer	Cancel Button	Expected Output
			VentFan
1	60	Release	Stop
2	999	Release	Stop
3	1000	Release	Stop
4	5	Release	Stop
5	0	Press	Stop
6	0	Release	-

할 수 있다. <표 8>에서 'Expected Output'열은 REED 시뮬레이터를 사용하여 실제 시뮬레이션을 수행한 결과이다. REED 시뮬레이터의 입력은 최종 테스트 케이스이고, 테스트 케이스 값을 넣었을 때, 시뮬레이터에서 생성된 출력값을 'Expected Output'에 추가하였다.

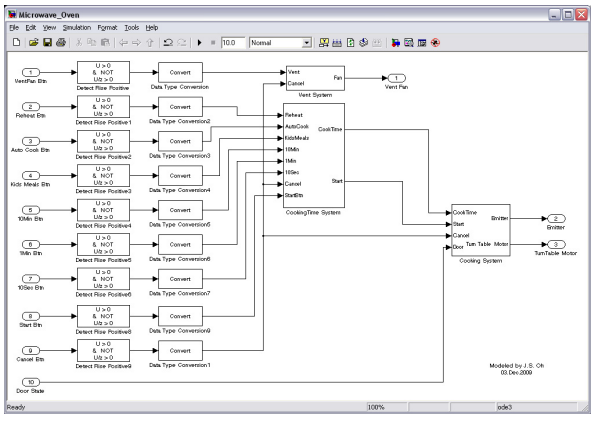
5.2 Simulink/Stateflow 모델을 이용한 테스트 케이스 생성 방법과의 비교

앞서 언급한 바와 같이, graphic object를 사용한다는 면에서 REED는 Simulink/Stateflow와 매우 흡사한 모습을 가진다. 이 절에서는 앞서 살펴본 전자레인지 모델 예제를 통하여 Simulink/Stateflow 모델과 REED의 요구사항 모델을 직접 비교한다.

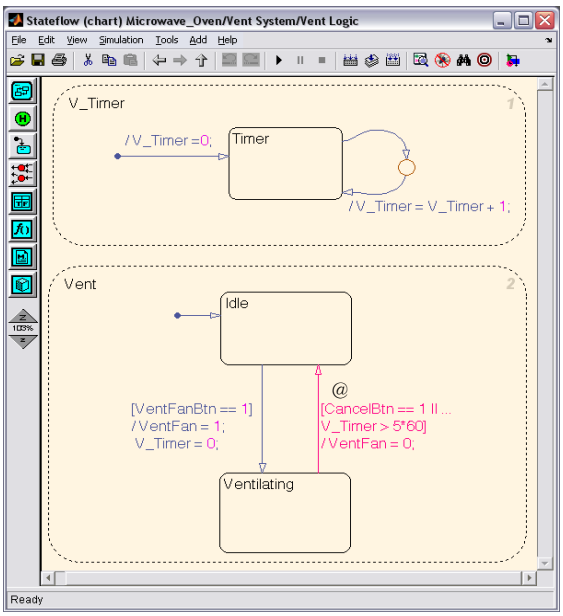
Simulink/Stateflow는 요구사항을 전체적으로 모델링해야 하는 m:n 모델링 언어이기 때문에 (그림 14)와 같이 하나의 통합된 모델로 표현되었다. 물론 최대한 각각의 요구사항을 구분할 수 있도록 subsystem을 이용하여 하위분류로 나누었다. 'CookingTime System' 내에는 4개의 하위 subsystem으로 다시 나누어서 모델링 하였다. 그럼에도, m:n 모델링 언어에서는 요구사항과 모델 간의 1:1 관계는 설정할 수 없었다.

(그림 13)의 REED 요구사항 모델은 (그림 15)와 같이 Simulink/Stateflow 모델로 모델링 되었다. 정확히 표현하면 (그림 15)의 Simulink/Stateflow 모델은 (그림 13)의 REED 요구사항 모델을 포함한다. 즉, (그림 15)의 'Ventilating' State에서 'Idle' State로 이동하는 @표시가 있는 transition (이하 @ transition)이 (그림 13)의 요구사항 모델에 해당하는 부분이다. REED를 이용할 때는 하나의 요구사항에 대한 테스트 케이스를 생성할 수 있었던 것과 달리 Simulink/Stateflow를 기반으로 테스트 케이스를 생성한다면 @ transition만을 테스트하기 위한 테스트 케이스 생성은 불가능하다. 물론 Simulink/Stateflow 모델도 전체적인 테스트 케이스를 생성할 수 있고, 그 가운데에서 @ transition을 테스트할 수 있다. 그러나 앞서서 이야기한 바와 같이 요구사항과 테스트 케이스의 관계가 m:n의 관계로 복잡해진다는 문제가 생긴다.

Simulink/Stateflow 모델을 대상으로 테스트 케이스를 생



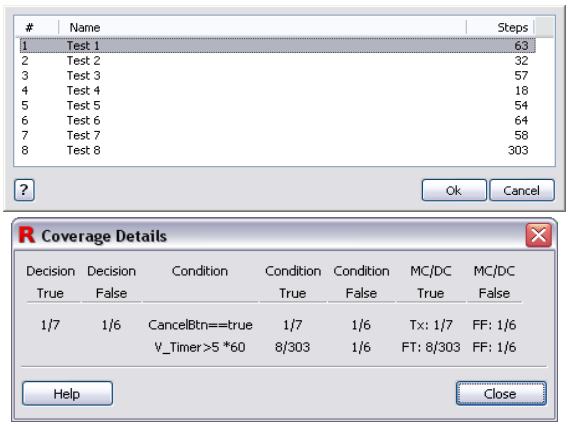
(그림 14) 전자레인지 규격에 대한 Simulink/Stateflow 모델



(그림 15) (그림 13)의 요구사항과 대응되는 Simulink/Stateflow 모델

성하는 상용도구 중에서 가장 널리 알려진 Reactis[14]를 사용하여 테스트 케이스를 생성하였다. (그림 16)은 모델 전체의 테스트 케이스 생성 결과(좌측)와 @ transition에 대한 커버리지 정보(우측)를 나타낸다.

전자레인지 모델에 대하여 총 8개의 테스트 케이스가 생성되었고 각 테스트 케이스의 Step은 최소 18개(Test 4)에서 최대 303개(Test 8) 이므로 총 649개의 Step을 갖는다. @ transition에 대한 커버리지 정보를 살펴보면, 이 transition의 Decision이 'True'가 되는 경우를 테스트하기 위한 테스트 케이스는 테스트 케이스 1번의 7번째 Step이며, Decision이 'False'가 되는 경우는 테스트 케이스 1번의 6번째 Step임을 알 수 있다. 해당 transition이 condition 2개를 가지고 있으므로 각각의 condition에 대한 정보도 모두 포함된 것을 알 수 있다. 특이할 점은 다른 조건은 모두 테스트 케이스 1번으로 테스트할 수 있지만, 'V_Timer > 5*60' condition이



(그림 16) Reactis를 이용한 전자레이지 모델의 테스트 케이스 생성 결과

'True'임을 살펴보기 위한 테스트 케이스는 테스트 케이스 8번의 303번째 Step이다. 전자레인지 규격에 대한 총 요구사항 개수는 26이다. 그러나 전체 256개의 요구사항에 대한 테스트 케이스가 총 8개밖에 생성되지 않았다는 말은 하나의 테스트 케이스가 여러 개의 요구사항과 관련이 있다는 말의 반증이다. 요구사항의 개수보다 테스트 케이스의 개수가 적다는 이야기는 하나의 테스트 케이스에서 여러 개의 요구사항에 대한 테스트를 동시에 수행하고 있다는 의미이다. 이처럼 하나의 요구사항을 테스트하기 위해서는 여러개의 테스트 케이스가 필요하고, 또한 하나의 테스트 케이스는 여러 개의 요구사항을 테스트하는 매우 복잡해 지는 결과가 나타나게 되었다.

앞서 설명한 바와 같이 REED를 통하여 생성된 단일 요구사항에 대한 테스트 케이스는 전체 시스템을 테스트하는 것이 아니므로 단일 요구사항에만 국한된다. 따라서 직접적으로 해당 요구사항만을 테스트하기 위한 테스트 케이스이다. 앞서 본 예제와 같이 대부분의 테스트 케이스가 Step이 1개로 구성된다(물론 2개 이상의 Step을 가지는 테스트 케이스가 생성될 수도 있다). 즉, <표 8>에서 생성된 6개의 테스트 케이스는 모두 (그림 13)의 요구사항을 테스트하기 위한 테스트 케이스 이며, 해당 테스트 케이스들은 다른 요구사항을 테스트하는데 사용될 수 없다. 이것은 다른 테스트 케이스 생성 기법에 대하여 본 논문이 제안하는 1:1 모델링을 통한 테스트 케이스 생성의 차이를 가장 잘 보여주는 것이라고 할 수 있다.

요구사항과 테스트 케이스가 1:n의 관계를 갖는 REED와는 달리 Simulink/Stateflow에서 하나의 요구사항(@ transition)을 테스트하기 위해서는 여러 테스트 케이스에서 해당 부분을 찾아서 실행시켜야 한다. 다행히 Reactis는 해당 transition이 실행되는 테스트 케이스를 알려주는 기능을 포함하고 있다. (그림 16)의 우측에 나타난 정보를 보면 하나의 요구사항인 @ transition을 테스트하기 위해서 테스트 케이스 1번과 테스트 케이스 8번을 수행해야 한다. 이 2개의 테스트 케이스는 총 366개의 Step을 가지며 전체 테스트 케이스의 약 56%의 비중을 차지한다. 즉, 하나의 요구사항을 테스트하기 위하여 전체 테스트 케이스의 절반 이상을 수행해야 한다. 시스템 전체를 테스트하는 데는 Simulink/Stateflow에서 생성한 테스트 케이스들이 유용하게 사용될 수 있겠지만, 하나의 요구사항을 테스트하기 위하여 이러한 방법으로 생성된 테스트 케이스를 사용하는 것은 매우 비효율적인 방법이라고 할 수 있다.

5.3 실험 분석

REED를 이용하여 임베디드 시스템의 요구사항을 모델링하고, 모델링된 요구사항 모델을 바탕으로 테스팅 작업을 수행하였다. 상용 자동차의 내부 온도 자동조절장치(TC), 버스 요금 계산기(BusCard Caching Machine), 굴착기 제어기 등 3개의 시스템의 요구사항은 자연어로 작성되어 있었다. 본 연구팀에서는 이들 요구사항을 REED를 이용하여 요구사항 모델로 모델링하고, 모델링된 요구사항 모델을 바탕으로 시스템들의 임베디드 소프트웨어를 테스트하였다. 이 3개의 모델은 실제 모델을 대상으로 진행되었던 프로젝트이기 때문에 자세한 모델은 공개할 수 없기에 여기에서는 간략한 통계치만 보인다. 전자레인지(Oven)는 사용자 메뉴얼을 분석하여 요구사항을 작성하였다. 전자레인지는 사용자 메뉴얼로부터 요구사항 다이어그램 작성이 가능한가, 메뉴얼은 충분히 그리고 충돌없이 전자레인지의 기능을 설명하고 있는지를 알아보기 위함이었다. 전자레인지에 대하여는 시뮬레이션만을 수행하였으며, 실제 전자레인지에 임베디드된 프로그램을 테스트하지는 못하였다.

요구사항 다이어그램의 작성 및 검증이 완료되고 나서, 각 시스템의 다이어그램을 기반으로 본 논문에서 기술한 방법으로 테스트 케이스를 생성하였다. 각 장비들의 간단한 특성, 요구사항 다이어그램 관련 통계, 테스트 케이스 생성에 관련된 통계는 <표 9>와 같다.

<표 9>에서 보는 바와 같이, 시스템들이 요구사항의 개수는 많으나 요구사항 모델로 모델링하기 위하여 사용된 객체의 수는 비교적 적다. TC는 다양하며 정교한 온도 조절 기능을 299개의 요구사항으로 표현하고 있다. 예를 들면, TC의 요구사항은 다양한 센서로부터 감지된 온도의 평균을 사용하거나, 센서의 고장에 대한 처리 기능 등 매우 다양하며 세밀한 제어 기능을 명시하고 있다. 이에 따라, 34종의 다양한 객체를 사용하여 요구사항 모델을 작성하였다. 그러나 굴착기 제어기는 TC과 비교할 때 요구사항의 수는 많으나 사용된 객체의 수는 적다. 굴착기 제어기는 TC보다는 많은 수의 입출력 장치들에 대한 제어를 기술하기 위하여 많은 요구사항을 명시하고 있다. 그러나 TC보다는 적은 29종의 객체를 사용하여 요구사항을 재구성할 수 있다. BusCard는 입출력 장치의 수는 적으나 다른 시스템과 TCP/IP 통신을 하는 분산 시스템이다. BusCard의 일부 기능만을 요구사항으로 표현하고 테스트를 수행하였다. TCP/IP 패킷에 기록된 정보는 많으나 그 처리가 비교적 간단하여 다이어그램에 사용된 그래픽적인 객체의 종류는 적으며 요구사항 모델당 많은 수의 객체를 사용하고 있다. 전자레인지는 다이어그램의 수는 작으나 다양한 객체를 사용하고 있다. 이는 전자레인지의 요구사항이 사용자 메뉴얼로부터 구성되면서, 사용자 메뉴얼에는 제어 목표는 표현되었으나, 제어의 상세한 요구사항이 없기 때문이다. 이러한 실험을 통하여 볼 때, 시스템의 종류나 기능, 복잡도에 따라 차이는 있으나, 대략 20~30종의 객체를 사용하여 250~400개 정도의 요구사항 다이어그램이 표현됨을 알 수 있다.

테스트 케이스의 개수는 TC는 5,566개, BusCard의 경우는 3,757개, 굴착기 제어기는 4,611개, Oven은 200개가 생성되었다. 사람이 수동으로 생성하기에는 쉽지 않은 개수의 테스트 케이스가 생성되었다. 3개의 실제 시스템 중에서 요구사항의 개수가 가장 적은 TC가 가장 많은 테스트 케이스를 생성하였다. 이러한 이유는 TC는 입출력 장치의 대표값이 많이 입력되었기 때문이다. 사용자가 테스트하기를 원하는 대표값을 많이 넣었기 때문에 그에 맞는 테스트 케이스

〈표 9〉 요구사항 모델을 이용한 테스트 케이스 생성 통계

		시스템			
		TC	Bus Card	굴착기 제어기	Oven
개요		자동차에 내부 온도조절장치	버스 요금 계산을 위한 운전자 단말기	굴착기 제어기용 엔진과 펌프의 마력 매칭을 위한 장비	일반 전자 레인지
장치	입력 장치	센서(7종), 스위치(13종)	키패드, GPS	센서(6종), 스위치(15종)	스위치(10종)
	출력 장치	Actuator(8종)	LCD, 스피커	Solenoid Valve 등 (7종)	LCD, Actuator (Motor, Microwave Emitter)
	입/출력장치	CAN	RS-232, 무선 랜 USB	CAN bus(engine 등) Gauge Pannel	
요구사항 총 개수		276	287	316	26
사용된 객체의 종류		31	23	28	20
객체의 최대 사용 개수 (요구사항별)		58	53	37	17
객체의 평균 사용 개수 (요구사항별)		9.49	7.29	4.54	7.96
테스트 케이스 총 개수		5566	3757	4611	200
테스트 케이스 최대 개수 (요구사항별)		347	186	172	40
평균 테스트 케이스 개수 (요구사항별)		20.17	13.09	14.59	8.00

를 모두 생성하기 위해서 많은 테스트 케이스가 생성된 것이다. 이는 각 요구사항 별 평균치를 보아도 다른 시스템에 비해서 월등히 많은 것을 볼 수 있다. Oven은 버튼의 입력이 다양하지 않기 때문에 대표값이 적게 입력이 되었고 이에 따라 요구사항당 8개씩의 테스트 케이스만 생성되었다.

Oven은 실제 하드웨어 없이 사용자 메뉴얼을 기반으로 생성한 모델이었기 때문에 <표 9>의 시스템 중에서 Oven을 제외한 3가지 시스템에 대해서만 테스트를 진행하였다. 3개의 모델에 대하여 테스트를 진행하기 위해서는 생성된 테스트 케이스를 실제 시스템의 입력으로 변환하는 과정이 필요하다. 이 과정을 테스트 케이스 프리픽스(Prefix) 생성 과정이라고 한다.

테스트 케이스 프리픽스를 생성하기 위해서는 2가지가 필요하다. 첫 번째는 요구사항 별로 생성된 테스트 케이스를 실행하기 위해서 해당 요구사항을 실행할 수 있는 상태로 시스템을 유도하기 위한 입력을 생성하는 것이다. 두 번째는 요구사항에서 생성된 테스트 케이스를 입력하기 위해서 실제 시스템에 어떠한 입력을 해야 하는지를 찾는 일이다. 예를 들어, <표 8>에서 생성한 테스트 케이스 1번을 수행하기 위해서는 첫 번째로 해당 요구사항을 실행할 수 있는 상태인 VentTimer가 시작된 상태로 만드는 입력들을 찾아야 하고, 두 번째로 VentTimer에 60이란 입력을 하기 위해서 필요한 실제 시스템의 입력을 찾아야 한다. 이 경우에는 해당 요구사항을 실행하기 위한 입력은 Oven의 Vent 버튼을 선택하는 것이고, VentTimer에 60이란 입력을 하기 위해서는 60초간 기다리는 것이 될 것이다. 요구사항 테스트 케이스의 입력은 VentTimer에 60이란 값을 입력하는 것이지만, 실제 시스템을 위한 테스트 케이스는 wait 60으로 변환되는 것이다.

3가지 시스템에 대하여 테스트 케이스 프리픽스를 생성한 결과 TC는 77.22%, BusCard는 81.95%, 굴착기 제어기는 33.11%가 나왔다. 비록 굴착기 제어기에서는 매우 낮은 커버리지가 도출되었지만, 나머지 2가지 시스템에서는 유의미한 커버리지를 도출할 수 있었다. 앞에서 Simulink/Stateflow 모델과의 차이점에서 살펴본 바와 같이 REED를 통하여 생성된 테스트 케이스는 전체적인 시스템의 동작을 살펴보기 위한 테스트 케이스라기보다는 하나의 요구사항을 평가하기 위한 모델링 수준의 테스트 케이스이므로 굴착기 제어기는 매우 낮은 커버리지를 보였다고 할 수 있다. 그러나 나머지 2개의 시스템에 대하여서는 실제 시스템의 입력을 기준으로 약 80%가량의 MC/DC 커버리지를 나타낸다는 것은 매우 높은 수준의 커버리지라고 할 수 있다.

생성된 테스트 케이스를 가지고 실제로 테스트를 수행해 본 결과 실제로 판매되고 있는 제품을 대상으로 테스트하였음에도 몇 가지 오류를 검출할 수 있었다. 판매되고 있는 제품에서 발견된 오류는 업체에서도 미처 파악하지 못한 것들이었다. 결과적으로 REED를 통하여 생성된 테스트 케이스가 비록 하나의 요구사항을 자세히 테스트하기 위한 것이지만, 이를 이용하여 요구사항 뿐만 아니라 전체 시스템을 테스트하는 것도 매우 유의미한 작업이라고 할 수 있었다.

6. 결론 및 향후 과제

본 논문에서는 1:1 요구사항 모델링 도구인 REED로 표현된 요구사항 모델로부터 MC/DC criteria를 만족하는 테스트 케이스를 자동으로 생성하는 알고리즘을 제안하였다. MC/DC는 항공전자에서 표준으로 사용하는 커버리지이므로, 이를 만족하는 테스트 케이스를 이용하여 테스트한 시스템은 높은 신뢰성을 가질 수 있다. 또한, 요구사항과 모델과의 관계가 1:1 관계를 맺고 있는 요구사항 모델로부터 테스트 케이스를 자동으로 생성하기 때문에, 테스트 오류에 대한 수정할 부분이나 재시험 항목을 명확하게 구별할 수 있다. 실제로 제안된 알고리듬으로 세 개의 상용 임베디드 시스템(자동차의 내부 온도 자동조절장치, 버스 요금 계산기, 굴착기 제어기)을 대상으로 테스트 케이스를 생성하여 보았고, 이를 우리가 HIL (Hardware In the Loop)환경에서 직접 테스트하여 보았다. 테스트된 시스템은 실제 상용 제품임에도 몇 가지의 오류가 검출되었고, 이는 이 논문이 제안하는 방법의 유용성을 보이는 증거라 할 수 있다.

본 연구에서는 테스트 케이스 프리픽스를 생성하였지만, 그 커버리지는 만족할 만한 수준이 못 되었다. 따라서 추후

연구에서는 테스트 케이스 프리픽스를 효과적으로 생성하는 방법을 연구할 예정이다. 또한, 다른 시스템을 테스트하여 본 알고리즘의 유용성을 높일 계획이다.

참 고 문 헌

[1] J.J. Gutierrez, M.J. Escalona, M.Mejias, and J.Torres, "Generation of test cases from functional requirements. A survey," Congreso:SV06. 4th Workshop on System Testing and Validation, 30 March, 2006.

[2] Tahat, L.H., Vaysburg, B., Korel, B., and Bader, A.J., "Requirement-based automated black-box test generation," 25th Annual International Computer Software and Applications Conference, Vol.2. pp.636-639, 2001.

[3] Ajitha Rajan, "Automated requirements-based test case generation," ACM SIGSOFT Software Engineering Notes, Vol.31, Issue6, pp.1-2, 2006.

[4] Offutt J, and Abdurazik A, "Generating tests from UML specifications," Proceedings of the Second International Conference on. the Unified Modeling Language (UML '99), pp.416-429, October 1999.

[5] Alessandra Cavarra, Charles chrichton, Jim Davies, Alan Hartman, Thierry Jeron, and Laurent Mounier, "Using UML for Automatic Test Generation," Tools and Algorithms for the Construction and Analysis of Systems, (TACAS'2000), 2000.

[6] Jörg Desel, Andreas Oberweis, and Torsten Zimmer, "A test case generator for the validation of high-level Petri nets," 1997 6th International Conference on Emerging Technologies and Factory Automation Proceedings, ETFA '97., pp. 327-332, Sep 1997.

[7] C. Bourhfir, R. Dssouli, E. Aboulhamid, and N. Rico, "Automatic executable test case generation for extended finite state machine protocols," Proceedings of IFIP TC6 10th International Workshop on Testing of Communicating Systems, pp.75-90, Sep., 1997.

[8] Jungsup Oh, Hongseok Lee, Hyunsang Park, Jangbok Kim, Kyunghee Choi, Kihyun Jung, "A Single Requirement Modeling with Graphical Language for Embedded System," The KIPS Transactions : Part D, Vol.15-D, No.4, (Serial Number 121), August 2008.

[9] Object Management Group, "Unified Modleling Language (UML), Version 2.1.2," http://www.omg.org/spec/UML/2.1.2/, November 2007.

[10] The MathWorks, Inc., http://www.mathworks.com/products/simulink/

[11] Conformiq Software Whitepaper, "Conformiq Qtronic™ SG : Semantic and Algorithms for Test Generation," http://www.conformiq.com, 2008.

[12] N. De Francesco1 and P. Inverardi2, "A semantic driven method to check the fineteness of CCS processes," Computer Aided Verification, Vol.575, pp.266-276, 1992.

[13] A.Gargantini and C.Heitmeyer, "Using Model Checking to Generate Tests from Requirements Specifications," In Proceedings of the Joint 7th Eur. Software Engineering Conference and 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering. Toulouse, France, pp.146-162, September 1999.

[14] Reactive Systems, Inc., http://www.reactive-systems.com/reactis.msp

[15] TNI Software, "Safety Test Builder, Automatic Test Generation, for Simulink/Stateflow," http://www.tni-software.com/commun/docs/safetytestbuilder.pdf.

[16] Applied Dynamics International, "BEACON for Simulink/Stateflow," http://www.adi.com/products_be_bss.htm.

[17] T-VEC Technologies, "T-Vec Tester for Simulink," http://www.t-vec.com/solutions/ simulink.php.

[18] The Mathworks, "Simulink Design verifier," http://www.mathworks.com.

[19] Hayhurst, K. J., Veerhusen, D. S., Chilenski, J. J., and Rierson, L. K. "A Practical Tutorial on Modified Condition/Decision Coverage," Report NASA/TM-2001-210876, May 2001, http://citeseer.ist.psu.edu/hayhurst01practical.html

오 정 섭

e-mail : jungsup.oh@gmail.com
1997년 아주대학교 정보및컴퓨터공학부(학사)
1999년 아주대학교 컴퓨터공학과(석사)
2009년 아주대학교 컴퓨터공학과(박사)
2010년~현 재 King's College London 방문 연구원
2001년~2003년 ㈜디오텔 선임연구원
2003년~2006년 ㈜삼성탈레스 책임연구원
관심분야 : 소프트웨어 공학, 요구사항 공학, 임베디드 시스템, 실시간 시스템, 분산 시스템 등

최 경 희

e-mail : khchoi@ajou.ac.kr
1976년 서울대학교 수학교육과(학사)
1979년 프랑스 그랑데꼴 Enseeiht대학(석사)
1982년 프랑스 Paul Sabatier대학 정보공학부(박사)
1982년~현 재 아주대학교 정보통신전문대학원 교수
관심분야 : 운영 체제, 분산시스템, 실시간 및 멀티미디어시스템 등

정 기 현

e-mail : khchung@ajou.ac.kr
1984년 서강대학교 전자공학과(학사)
1988년 미국 Ilinois주립대 EECS(석사)
1990년 미국 Purdue대학 전기전자공학부(박사)
1991~1992년 현대반도체 연구소
1993년~현 재 아주대학교 전자공학부 교수
관심분야 : 컴퓨터구조, VLSI 설계, 멀티미디어 및 실시간 시스템 등