상태모델에 기반한 소프트웨어 컴포넌트 명세의 정형적 테스트

서동수†

요 약

C4I 시스템과 같이 신뢰성이 강조되는 시스템의 개발에 있어 정형기법의 도입은 개발 과정에서 나타나는 결과물에 대한 검증을 가능케 함으로서 올바른 시스템을 개발하고 있다는 확신을 개발자와 의뢰자 모두에게 제공해준다. 본 논문은 엄밀한 검증이 필요한 컴포넌트를 설계할 때 사용될 수 있는 컴포넌트의 정형명세 기법과 이에 기반한 테스트 방법을 논한다. 특히 상태기반의 기능명세에 대해 미약선조건을 이용하여 분해하는 방법과 이 과정을 통해 생성된 하위명세에 대한 블랙박스 테스트케이스를 생성시키는 방법을 제시한다.

키워드 : 테스트, 정형기법, 소프트웨어 컴포넌트

Formal tests for State-model based Specifications on Software Components

Dongsu Seo†

ABSTRACT

In developing highly reliable systems such as C4I systems formal methods provide both developers and clients with assurance that they are in the right development processes. This paper investigates into techniques for formal specifications and tests for software components where rigorous verification is required. In particular, the paper suggests decomposition techniques for state-model based specifications using the weakest precondition, and suggests test methods for the specification by generating black box test-cases.

Keywords : Test, Formal Methods, Software Component

1. 서 론

최근 들어 소프트웨어 개발의 새로운 패러다임으로 자리잡아가는 컴포넌트기반 소프트웨어 개발은 컴포넌트를 사용하는데 있어 잘 정의된 인터페이스만을 통해 서비스를 제공하도록 제한함으로서 개발의 복잡성을 줄이고 재사용 비용을 감소하도록 하는데 도움을 주는 것으로 알려졌다 [1]. 이러한 효과를 적절히 얻기 위해서 선행되어야 하는 작업이 올바른 컴포넌트의 기능명세에 관한 사항이다. 컴포넌트의 재사용이 원하는 수준의 효과를 제공하기 위해서는 컴포넌트의 기능에 대해 명료하고 정확한 서술이 제공되어야 한다.

컴포넌트의 기능명세를 위해서는 채택될 수 있는 서술방법으로는 자연어에 의한 서술, 구조적 언어에 의한 서술, 정형언어를 사용하는 서술 등을 들 수 있다. 많은 경우 시스템 개발에 있어서는 주로 자연어에 의한 서술과 구조적 언어에 의한 서술을 사용했으나 엄밀한 검증을 필요로 하는

† 정 회 원: 성신여자대학교 부교수
논문접수: 2004년 8월 6일, 심사완료: 2004년 9월 9일
* 이 논문은 2002년도 성신여자대학교 기초과학연구소 특별과제 연구비에 의해 연구되었음

보안시스템 및 C4I 시스템과 같은 고신뢰 시스템의 개발에 있어 이러한 명세 방법은 심각한 문제를 발생시키는 것으로 지적되어왔다 [2]. 대표적인 문제는 명세 내에 포함되는 불일치성과 모순성, 명세 자체의 불완전성에 관한 것이다. 이러한 문제를 완화시키는 수단으로 정형기법은 적절한 검증 방법을 제공함으로서 사용자가 원하는 수준의 검증을 가능하도록 한다. 정형명세는 개발자들이 프로그램을 작성할 경우 지켜야 할 제약조건을 담고 있다. 따라서 프로그램 코드가 얼마나 정형명세에서 표현된 조건들을 준수하고 있는지를 보여주는 지를 보여주는 일치성 확인은 개발자나 의뢰자 모두에게 중요한 일이다. 그러나 코드의 일치성 외에도 중요하게 검증되어야 할 사항이 분석문서와 설계문서 간의 일치성 혹은 상위 명세와 하위 명세 간의 일치성에 관한 사항이다.

본 논문에서는 이러한 맥락에서 추상성의 차이를 갖는 분석 문서와 설계문서 간의 일치성에 대한 보증이 필요할 경우 이용될 수 있는 테스트케이스의 생성 기법에 관해 논한다. 특히 상태기반의 명세를 사용하는 경우 조건분해를 통해 테스트케이스를 생성하는 규칙에 관해 논하며 이의 적용을 통해 결과의 효용성을 알아본다.

2관련연구

정형기법에 관한 연구는 예전부터 다양한 시각에서 시도되었으나 컴포넌트 소프트웨어에 관한 정형적인 접근은 그리 오래된 편이 아니다. 객체에 관한 정형적 표기법과 시멘틱스에 관한 대표적인 연구로 OOZE[3], VDM++[4]가 있으며 이들 방법은 객체가 갖는 상태를 주로 명세함으로서 객체지향적인 특성을 정형화 시키고자 노력하였다. 특히 객체의 메소드를 호출하는 방법으로 Alencar와 Goguen은 OOZE를 위한 Z 스키마 방식의 호출을, Durr는 VDM++을 위한 메시지 패싱 방식을 제안하였.

다른 연구로는 정형적 개발 절차에 관한 분야로서 명세의 정제(refinement)에 관한 연구가 있다. 정제 기법은 구조적 기법의 분할개념과 유사하나 각 정제 단계는 엄밀한 증명을 요한다는 점에서 차이가 있다. 정제 단계의 증명 기법으로는 Jones의 선조건과 후조건의 관계를 이용한 자료 정제와 오퍼레이션 정제를 분리한 정제기법[5], Woodcock의 스키마에 기반한 정제 기법 [6] 등을 들 수 있다.

객체지향 시스템에 관한 구체적인 테스트 기법들은 관해서는 [7]에 언급되어 있으며 사용자 시나리오로부터 테스트케이스를 유도하려는 시도는 Briand [8]에 의해 제시된 바가 있다. Meudec [9]는 객체지향시스템을 대상으로 하지는 않았으나 일반적인 상태기반 명세에 대해 테스트케이스를 생성하는 알고리즘을 제시하였다. Briand가 제시한 유스케이스의 분석을 통한 테스트케이스 생성은 엑터의 관점에서 보는 시스템의 서비스를 테스트하며 순차도에 나타난 메시지의 분석을 통해 시스템 테스트에 관한 유용한 정보를 추출한다. 하지만 정형적인 엄밀성이 요구되는 시스템에 대한 검증에 관해서는 적절한 테스트 기준을 제공하기 힘들다는 단점이 있다.

본 연구에서는 정형적 정제활동에 대한 검증이 가능하도록 하는 테스트케이스 생성을 위해 Meudec이 제시한 테스트케이스 생성과정을 본 연구의 컴포넌트 명세에 적합하도록 적용하여 보았다.

3상태모형의조건분해

컴포넌트의 기능을 서술하는 방법 중 하나는 상태 기반 모형 (state-based model)을 이용하는 것이다. 상태란 컴포넌트가 갖는 중요한 파라메터, 혹은 속성 정보 값들에 의해 표현되는 특성을 의미하며 상태는 시간의 흐름 혹은 이벤트의 변화에 따라 일정한 양식을 가지고 변화하게 된다. 이 과정에서 초기 상태와 종료상태를 구분할 수 있는데 초기상태와 종료상태를 정확히 정의하는 것은 명확한 기능 서술의 시작이라 할 수 있다.

어떤 기능 f의 초기상태 조건을 $pre\text{-}f$, 종료상태의 조건을 $post\text{-}f$라 할 때 실제로 기능 f가 올바르게 수행되었는지 판단하기 위해서는 f의 호출 전에 f의 초기상태에 관한 제약조건인 $pre\text{-}f$의 진리치를 살펴보고, f의 수행 이후에 종료상태 조건

인 *post-f*의 값 또한 올바른지를 확인함으로써 가능하다. 상태기반모형이란 이러한 시각에서 시스템의 기능을 서술할 때 초기상태의 조건 즉, 선조건(pre-condition)과 종료상태의 조건 즉, 후조건 (post-condition)을 사용하여 서술된 모형을 말한다. 정형명세 표기법 중의 하나인 VDM [5]은 상태를 표현하는데 있어 함수 단위로 서술한다. 함수는 선언부에서 시그니쳐를 포함하며 시그니쳐는 함수의 도메인과 치역에 대한 유용한 정보를 제공해준다. 명세는 함수의 내용을 절차적으로 혹은 직접적으로 표현하는 직접 명세(explicit specification)와 선조건, 후조건에 의한 특성만을 표현하는 간접 명세(implicit specification) 방식을 선택적으로 사용할 수 있다. 직접명세는 알고리즘을 서술하는 방식의 표현이며 가장 널리 사용되는 기능정의 방식이다. 반면에 간접명세는 즉 함수의 기능 자체를 서술하는 것이 아니라 함수가 입출력으로 가질 수 있는 자료의 제약 조건만을 표현하여 함수를 간접적으로 명세하는 방법이다. 함수 f의 간접명세는 다음의 구조를 갖는다.

f (p: Tp) r : Tr

pre *... p ...*

post *...p...r...*

단, *p*는 파라메터 리스트, *Tp*는 파라메터 자료형, *r*은 반환값, *Tr*은 반환값은 자료형이며 **pre** 란 오퍼레이션이 시작되기 전에 만족해야 하는 입력 변수들의 제약 조건을 나타내는 선조건을, **post**는 오퍼레이션의 실행이 끝난 시점에서 만족되어야 하는 입, 출력 변수들의 제약 조건을 표현하는 후조건을 의미는 술어 논리함수이다.

컴포넌트의 서비스를 구현하기 위해서는 일반적으로 하나 이상의 내부 객체들을 필요로 한다. 따라서 개발자 입장에서 서비스를 명세할 경우 내부 객체의 서비스를 어떠한 방식으로 연결시킬 것인가를 고려해야 하며, 이 때 선조건과 후조건의 개념은 유용하게 쓰일 수 있다. 컴포넌트 C의 서비스 C_{S1}은 실행 직전의 시점에서 C_{S1} 내에서 정의되는 선조건 pre-C_{S1}을 만족시킬 경우 서비스 C_{S1}은 수행 종료가 보장되며 동시에 수행종료 후에는 후조건 *post-*C_{S1}을 만족시킨다고 할 때 이 관계는 식 (1)과 같이 표현한다.

$$\{\text{pre-}C_{S1}\}\ C_{S1}\ \{post\text{-}C_{S1}\} \quad (1)$$

컴포넌트 서비스 C_{S1}과 후조건 *post-*C_{S1}사이에 존재하는 최약선조건(weakest pre-condition)이란 만일 C_{S1}이 이들 조건 중의 어느 하나라도 만족시키는 상태에서 출발한다면 C_{S1}은 반드시 종료되며 동시에 조건 *post-*C_{S1}을 만족시키는 것이 보장되는 조건을 말하며, 기호로는 $wp(C_{S1}, post\text{-}C_{S1})$로 표기한다. 만일 선조건 pre-$C_{S1}$와 최약선조건 $wp(C_{S1}, \text{post-}C_{S1})$ 사이에 식 (2)와 같은 관계가 존재한다면 C_{S1}은 또한 반드시 종료되며 동시에 조건 *post-C*를 만족시키는 것이 보장된다고 할 수 있다.

$$\text{pre-}C_{S1} \Rightarrow wp(C_{S1}, \text{post-}C_{S1}) \quad (2)$$

예로서, 만일 다음의 함수 increase() 에 대해 다음과 같은 정의를 할 경우:

```
int increase(int i){
  return i:= i+ 1;}
```

이에 대한 후조건이 $i \geq 1$ 로 확정되었다면 이 문장이 종료되고 또한 후조건을 참으로 성립시키기 위해 필요한 선조건의 최약선조건은 $i \geq 0$ 이다. 따라서 선조건이 $i \geq 0$ 인 경우에만 한해 다음 식 (3)을 참으로 유도한다.

$$i \geq 0 \Rightarrow wp\,(increase, i \geq 1) \quad (3)$$

위 식을 다른 식으로 해석한다면 만일 $i < 0$인 경우는 함수 *increase*는 조건 $i \geq 1$을 만족시키지 못한다고 말할 수 있다. 최약 선조건을 컴포넌트 명세 및 컴포넌트 합성에 활용하기 위해서는 다음과 같은 정의를 필요로 한다.

[정의 1] 최약 선조건 *wp*가 거짓인 상태에서 수행되면서 컴포넌트 서비스 C_{S1}의 수행을 시작하고 종료 시키는 경우는 없다

[정의 1]은 최약선조건 *wp*가 참인 경우에만 그 이후의 논리 전개가 의미 있음을 확언하기 위해 도입된다.

통상적으로 컴포넌트의 서비스는 이를 구성하는 하나 이상의 내부 객체들이 제공하는 오퍼레이션들의 조합에 의해 이루어진다. 가장 간단한 예로서 컴포넌트 C를 구성하는 객체 O_1 과 O_2 가 있다고 할 때 다음과 같은 정리가 유효하다.

[정리1] 하나의 컴포넌트 서비스 C_{S1}에 대해 독립적으로 존재하는 두 개의 최약선조건 $wp(C_{S1}, post\text{-}O_1)$ 와 $wp(C_{S1}, post\text{-}O_2)$ 사이

에는 다음과 같은 논리곱의 관계가 성립한다.

$wp(C_{S1}, post\text{-}O_1 \wedge post\text{-}O_2) \Leftrightarrow wp(C_{S1}, post\text{-}O_1) \wedge wp(C_{S1}, post\text{-}O_2)$

[정리1]은 최약선조건 내에 논리곱이 있다면 이들은 논리곱에 의해 분해를 할 수 있음을 정의한 것이며 이는 항진 명제의 규칙을 통해 다음과 같이 증명할 수 있다.

$wp(C_{S1}, post\text{-}O_1 \wedge post\text{-}O_2)$가 $post\text{-}O_1 \wedge post\text{-}O_2$를 만족시키는 컴포넌트 서비스 C_{S1}을 종료 시킬 수 있는 초기 상태라 한다면, 이것은 어떠한 초기 상태들도 다음 두 가지 조건(4)와 (5)을 만족 시킨다.

$wp(C_{S1}, post\text{-}O_1 \wedge post\text{-}O_2) \Rightarrow wp(C_{S1}, post\text{-}O_1)$ (4)

$wp(C_{S1}, post\text{-}O_1 \wedge post\text{-}O_2) \Rightarrow wp(C_{S1}, post\text{-}O_2)$ (5)

따라서,

$wp(C_{S1}, post\text{-}O_1 \wedge post\text{-}O_2) \Rightarrow wp(C_{S1}, post\text{-}O_1) \wedge wp(C, post\text{-}O_2)$ (6)

또한, $wp(C_{S1}, post\text{-}O_1) \wedge wp(C_{S1}, post\text{-}O_2)$는 $post\text{-}O_1 \wedge post\text{-}O_2$를 만족시키는 컴포넌트 서비스 C_{S1}을 종료 시키는 조건으로 충분하므로 다음을 성립시킨다.

$wp(C_{S1}, post\text{-}O_1) \wedge wp(C_{S1}, post\text{-}O_2) \Rightarrow wp(C_{S1}, post\text{-}O_1 \wedge post\text{-}O_2)$ (7)

따라서 [정리1]은 (6)과 (7)에 의해 증명될 수 있다.

[정리 2] 하나의 컴포넌트 서비스 C에 대해 독립적으로 존재하는 두 개의 최약선조건 $wp(C, post\text{-}O_1)$ 와 $wp(C, post\text{-}O_2)$ 사이에는 다음과 같은 논리합의 관계가 성립한다.

$wp(C, post\text{-}O_1 \vee post\text{-}O_2) \Leftrightarrow wp(C, post\text{-}O_1) \vee wp(C, post\text{-}O_2)$

[정리 2]의 경우는 최약 선조건이 논리합으로 연결되어 있을 경우 적용할 수 있는 분배 법칙을 정의한다. [정리 2] 역시 [정리 1]과 유사한 과정을 거쳐 증명할 수 있으므로 증명과정은 생략한다.

[정리 3] 하나의 컴포넌트 서비스 서비스 C_{S1}에 대해 독립적으로 존재하는 두 개의 최약선조건 $wp(C_{S1}, post\text{-}O_1)$ 와 $wp(C_{S1}, post\text{-}O_2)$ 사이에는 다음과 같은 함축의 관계가 성립한다.

$(post\text{-}O_1 \Rightarrow post\text{-}O_2) \Rightarrow (wp(C_{S1}, post\text{-}O_1) \Rightarrow wp(C_{S1}, post\text{-}O_2))$

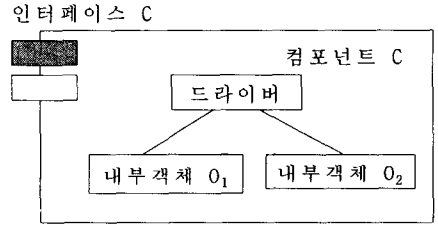
[정리 3]의 경우는 서비스 C_{S1}의 실행이 $post\text{-}O_1$을 성립시키고, $post\text{-}O_1 \Rightarrow post\text{-}O_2$관계가 존재한다면, 컴포넌트 서비스 C_{S1} 은 $post\text{-}O_2$ 역시 성립시킬 수 있다는 의미다. 이것은 wp 논리식에서 단조성(monotonicity)이 일어날 수 있음을 뜻하며 이 성질은 [정리 1]을 적용하여 쉽게 증명할 수 있다.

컴포넌트의 기능 분할 혹은 컴포넌트의 통합에 관한 검증을 수행하기 위해서는 규칙에 기반한 단계 확인이 제시되어야 하며 이를 위한 선행 작업으로 앞서 살펴본 최약선조건의 정의를 바탕으로 컴포넌트 제약사항 표현에 적합하도록 분할규칙을 정의할 필요가 있다. 이 경우의 관계는 다음과 같이 정의한다.

$$pre\text{-}C \Leftrightarrow pre\text{-}O_1 \vee pre\text{-}O_2$$

$$post\text{-}C \Leftrightarrow (pre\text{-}O_1 \Rightarrow post\text{-}O_1) \vee (pre\text{-}O_2 \Rightarrow post\text{-}O_2)$$

컴포넌트 서비스에 관한 가장 간단한 연결 관계는 순차관계이다. 순차관계란 어떤 컴포넌트의 서비스를 구성하는 방법으로서 하나 이상의 내부 객체 서비스가 순차적으로 연결되는 관계를 말한다.



<그림 1> 인터페이스의 순차구성

예로서 <그림 1>에서와 같이 컴포넌트 인터페이스C는 내부객체 O_1과 O_2 그리고 이들 서비스를 순차적으로 실행시키는 드라이버를 포함하는 구조를 가졌다고 정의하자. 이 경우 컴포넌트의 인터페이스C가 선조건 $pre\text{-}O$와 후조건 $post\text{-}O$에 의해 표현되며 내부 객체 O_1과 O_2 의 명세조건은 각각 $pre\text{-}O_1$과 $post\text{-}O_1$ 그리고 $pre\text{-}O_2$와 $post\text{-}O_2$로 표현된다고 할 때 이들 사이에 생기는 관계는 다음의 [정리4]에 의해 표현된다.

[정리 4] 컴포넌트 C와 후조건 $post\text{-}O$에 대해 내부 객체 O_1과 O_2가 순차 실행 관계 ";"으로 연결된다면 객체 O_1, O_2와 후조건 $post\text{-}O$ 사이에는 다음 관계가 성립한다

$wp("O_1; O_2", post\text{-}O) \Leftrightarrow wp(O_1, wp(O_2, post\text{-}O))$

이상에서 설명한 조건들에 대해 본 논문은 컴포넌트를 합성하거나 분해할 경우 원래의 컴포넌트를 명세할 때 사용되었던 선조건과 후조건들에 대한 결합 관계를 정의한다.

4.테스트케이스의생성

상태기반 명세로부터 테스트케이스를 생성하기 위해서는 명세에 대한 테스트 기준이 먼저 설정되어야 한다. 상태기반 테스트는 속성상 모듈에 관한 블랙박스 테스트이다. 테스트케이스의 생성에 근거가 되는 정보는 명세에 포함된 선조건과 후조건이다. 즉 명세에 대한 설계가 있다면 이들 설계는 명세 내에 포함된 선조건과 후조건을 만족해야지만 올바른 설계라 할 수 있다. 이러한 예로서 " 0 보다 큰 임의의 자연수 x를 반지름으로 하는 원의 면적을 구하는 함수 *circle*이 참값이라면 면적 *size*의 근사치 범위가 0.00001보다 작아야한다" 라는 조건에 대해 간접명세를 하면 다음과 같다.

circle(x:int, size:real) result:bool

pre x > 0

post result ↔ (x*x* π - size <0.00001)

간접명세로부터 유도되는 테스트케이스는 직접명세 혹은 직접명세로부터 구현된 구현코드가 올바르게 작동함을 보일 때 쓰일 수 있는 블랙박스 테스트의 기준을 제공해 준다.

블랙박스 테스트의 경우 가장 많이 사용되는 테스트케이스 생성방법은 동치분해(equivalent class partitioning) 방법이다. 동치분해는 일반적으로 정상적인 값의 클래스와 비정상적인 값의 클래스로 구분한다. 상태기반의 명세로부터 테스트케이스를 생성시키기 위해서 제일 적당한 방법은 조건이 가질 수 있는 모든 경우를 포함하는 조건(condition) 커버리지를 생성하는 것이다. 조건 커버리지의 대상은 선조건과 후조건에 포함된 모든 조건들이다. 선조건과 후조건에 대한 조건 커버리지가 일반적인 조건 커버리지와 다른 점은 이들 조건은 결과 값이 참인 조건에 한해 고려한다는 점이다. 그 이유는 선조건, 후조건의 정의상 파라메터들에 대해 참인 조건들만을 명시하고 있기 때문에 위 예의 경우 x의 값이 0 혹은 음수인 경우는 선조건을 거짓으로 만들기 때문에 의미가 없어지게 된다. 따라서 어떠한 경우에도 테스트케이스로 생성된 값은 x에는 0 혹은 0보다 작은 값이 할당되어서는 안 되며 결과적으로 선조건은 하나의 동치클래스인 x>0만을 가진다. 후조건에 대해서는 result∧ (x*x* π - size <0.00001), ¬result ∧(x*x* π - size ≥ 0.00001)과 같은 2개의 경우가 생성된다.

본 논문에서는 테스트케이스 생성을 위한 규칙들을 정의함에 있어 다음 사항을 정의한다. 첫째, 논리식을 평가(evaluation)하는 함수로 evalT와 evalF를 도입한다. 함수 evalT는 단항 연산자로서 논리식 e에 대해 평가하여 참 값만을 반환하며, 함수 evalF 는 e에 대한 거짓 값을 반환한다. 이들 함수에 대한 관계는 식 (1) ~(4)에 정의된다.

$$evalT(e) = \{e\} \quad (1)$$

$$evalF(e) = \{\neg e\} \quad (2)$$

$$evalT(\neg e) = evalF(e) \quad (3)$$

$$evalF(\neg e) = evalT(e) \quad (4)$$

다음의 식 (5)~ (12)는 논리 연결자는 ¬, ∧, ∨, ⇒, ⇔, =, ≠에 대한 관계를 정의한 것으로서 식들 간의 가능한 모든 논리곱을 표현할 경우는 편의상 카테지안 곱(×)을 사용하며 논리곱과 카테지안 곱은 서로 호환적인 기호이다.

$$evalT(e1 \wedge e2) = evalT(e1) \times evalT(e2) \quad (5)$$

$$evalF(e1 \wedge e2) = \begin{Bmatrix} evalTrue(e1) \wedge evalFalse(e2) \\ evalFalse(e1) \wedge evalTrue(e2) \\ evalFalse(e1) \wedge evalFalse(e2) \end{Bmatrix} \quad (6)$$

$$evalT(e1 \vee e2) = \begin{Bmatrix} evalTrue(e1) \wedge evalTrue(e2) \\ evalTrue(e1) \wedge evalFalse(e2) \\ evalFalse(e1) \wedge evalTrue(e2) \end{Bmatrix} \quad (7)$$

$$evalF(e1 \vee e2) = evalF(e1) \times evalF(e2) \quad (8)$$

$$evalT(e1 \rightarrow e2) = \begin{Bmatrix} evalTrue(e1) \wedge evalTrue(e2) \\ evalFalse(e1) \wedge evalTrue(e2) \\ evalFalse(e1) \wedge evalFalse(e2) \end{Bmatrix} \quad (9)$$

$$evalF(e1 \rightarrow e2) = evalT(e1) \times evalF(e2) \quad (10)$$

$$evalT(e1 \leftrightarrow e2) =$$

$$\begin{Bmatrix} evalTrue(e1) \wedge evalTrue(e2) \\ evalFalse(e1) \wedge evalFalse(e2) \end{Bmatrix} \quad (11)$$

evalF(e1↔e2) =

$$\begin{Bmatrix} evalTrue(e1) \wedge evalFalse(e2) \\ evalFalse(e1) \wedge evalTrue(e2) \end{Bmatrix} \quad (12)$$

evalF(e1∧e2)의 경우는 논리식 e1∧e2를 평가하는데 있어 거짓 값만을 반환하게 되므로 위의 식(6)과 같은 3가지 경우를 테스트하여야 한다. 다음은 식 (13)~(24)는 논리식의 비교와 관계된 규칙들을 포함한다.

evalT(e1<e2) =

$$evalDef(e1) \times evalDef(e2) \times \begin{Bmatrix} e1 + 1 = e2 \\ e1 + 1 < e2 \end{Bmatrix} \quad (13)$$

evalF(e1<e2) = evalT(e1≥e2) (14)

evalT(e1>e2) =

$$evalDef(e1) \times evalDef(e2) \times \begin{Bmatrix} e1 = e2 + 1 \\ e1 > e2 + 1 \end{Bmatrix} \quad (15)$$

evalF(e1>e2) = evalT(e1≤e2) (16)

evalT(e1≤e2) =

$$evalDef(e1) \times evalDef(e2) \times \begin{Bmatrix} e1 = e2 \\ e1 < e2 \end{Bmatrix} \quad (17)$$

evalF(e1≤e2) = evalT(e1>e2) (18)

evalT(e1≥e2) =

$$evalDef(e1) \times evalDef(e2) \times \begin{Bmatrix} e1 = e2 \\ e1 > e2 \end{Bmatrix} \quad (19)$$

evalF(e1≥e2) = evalT(e1<e2) (20)

evalT(e1=e2) =

$$evalDef(e1) \times evalDef(e2) \times \{e1=e2\} \quad (21)$$

evalF(e1=e2) = evalT(e1≠e2) (22)

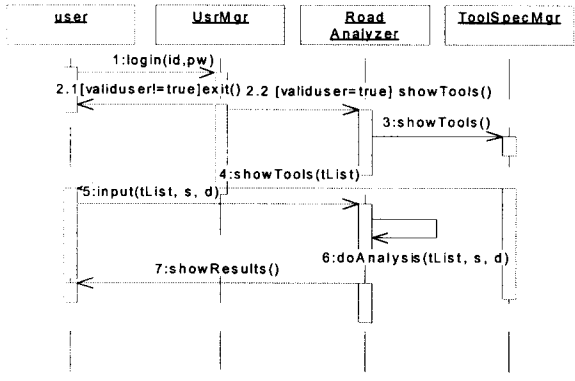
evalT(e1≠e2) =

$$evalDef(e1) \times evalDef(e2) \times \begin{Bmatrix} e1 = e2 + 1 \\ e1 > e2 + 1 \\ e1 + 1 = e2 \\ e1 + 1 < e2 \end{Bmatrix} \quad (23)$$

evalF(e1≠e2) = evalT(e1=e2) (24)

evalDef(e) = true, 단, evalDef는 산술 수식값을 계산하며 참 값을 반환한다.

이상의 설명을 바탕으로 C4I 기능 중 지도상에 표현된 지형과 도로망 관리 및 분석을 수행하는 도로분석기 기능에 적용해본다. 도로분석기 컴포넌트가 제공하는핵심 오퍼레이션 중 하나인 *startMapAnalysis*는 사용이 허가된 사용자에 한해 입력을 받으며 사용자는 출발지와 도착지, 그리고 도로분석에 사용될 방법을 입력하며 화면을 통해 결과물을 표현하는 기능을 갖는다. 각각의 기능들은 이를 담당하는 내부 클래스가 제공하는 내부 오퍼레이션의 순차적인 수행에 의해 이루어지며 내부 클래스로는 사용자 관리기(UsrMger), 분석기(RoadAnalyzer), 그리고 도구관리기(ToolSpecMgr) 등으로 구성된다. *startMapAnalysis*의 작동순서는 <그림 2>와 같은 UML의 순차 다이어그램으로 표현된다.



<그림 2> 도로망 분석 순차도

*startMapAnalysis*오퍼레이션의 명세를 살펴보면 다음과 같다.

startRoadMapAnalyzer : User × TList × Map → Bool

ext rd validusr : ID × Passwd → Bool

pre *∃(id,pw) ∈(ID,Passwd), ∃tList∈TList, ∃s, d ∈Loc • validusr(id, passwd) ∧{ tList} ≠{} ∧ {s}∩{d}={}*

post *∀tList∈TList, ∀s, d∈Map • doAnalysis(tList, s,d)=OK ∧ {tList} = {$\overrightarrow{tList}$} ∧{s} = {$\vec{s}$ } ∧{d} = {$\vec{d}$}*

위의 명세는 VDM 스타일의 명세로서 선조건부 ***pre***와 후조건부 ***post*** 절로 구성된다. 선조건부에서 정의하는 조건 중 *validusr(id, passwd)*는 암호와 아이디로 인증된 사용자에 한해 허용함을, *tList ≠{}*는 등록된 도구 리스트 중 반드시 하나 이상의 요소를 선택함을, *{s}∩{d}={}*는 출발점 s와 도착점 d는 서로 동일하지 않음을 조건으로 함을 의미한다. 후조건부의 내용은 함수 *startMapAnalysis*가 올바로 수행을 종료됨을 보장하며, *{tList} = {$\overrightarrow{tList}$}* 조건은 수행전의 *tList* 내용과 수행후의 tList 내용은 동일해야 함을 보장한다. (단, 장식기호 → 는 함수 수행전의 자료상태를 의미한

다) 마찬가지로 $\{s\} = \{\vec{s}\} \wedge \{d\} = \{\vec{d}\}$는 각각 출발점과 도착점의 내용은 오퍼레이션의 수행 이후에도 변경이 없어야 함을 서술한다.

도로분석기 컴포넌트와 유사한 기능을 갖는 장애물 분석기 컴포넌트는 선택된 지형에 존재하는 여러 가지 장애물의 배치를 알려주는 오퍼레이션인 *startObstacleAnalyzer*을 제공한다. 이 서비스는 도로분석기능이 갖는 시나리오 중 계절 요소를 추가로 입력한다는 측면에서 도로분석기와 차별되며 나머지는 유사한 과정을 거친다. 다음은 *startObstacleAnalyzer*에 대한 명세이다.

startObstacleAnalyzer : User × TList × Map → Bool

ext rd validusr : ID × Passwd → Bool

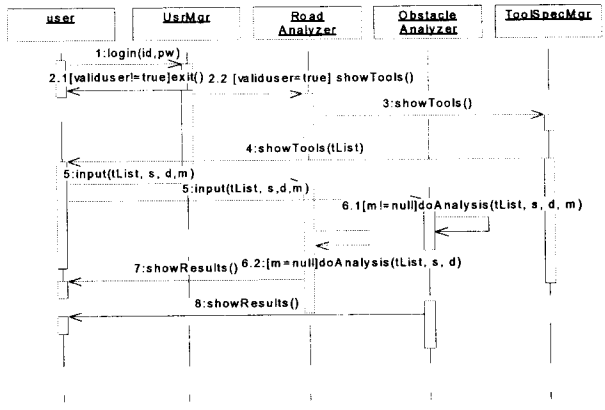
pre $\exists(id,pw) \in (ID,Passwd), \exists tList \in TList, \exists s, d \in Loc, \exists m \in Month \bullet validusr(id, passwd) \wedge \{tList\} \neq \{\} \wedge \{s\} \cap \{d\} = \{\} \wedge \{m\} \neq \{\}$

post $\forall \exists tList \in TList, \forall s, d \in Map \bullet$

$doAnalysis(tList,s,d,m) = OK \wedge \{tList\} = \{\overrightarrow{tList}\} \wedge \{s\} = \{\vec{s}\} \wedge \{d\} = \{\vec{d}\} \wedge \{m\} = \{\vec{m}\}$

startObstacleAnalyzer 명세는 도로분석기 명세와 유사하며 선조건절에 포함된 조건 $m \neq \{\}$은 반드시 1개 이상의 달을 선택해야 함을 의미한다.

만일 위에서 서술된 두 개의 오퍼레이션 *startMapAnalysis*과 *startObstacleAnalyzer*를 통합한 지형분석기 컴포넌트를 개발한다고 가정하자. 개발자는 도로분석 기능과 장애물분석 기능의 시나리오의 분석을 바탕으로 컴포넌트를 합성할 경우 각각의 순차도에 표현된 오퍼레이션들의 범위를 설정한 뒤 <그림 3>과 같은 순차 다이어그램을 작성한다. 이 순차도는 도구선택과 계절선택, 그리고 객체 *ObstacleAnalyzer*와 *RoadAnalyzer* 내부에 정의된 오퍼레이션인 각각의 doAnalysis()가 선택적으로 호출되도록 표현하고 있다. 따라서 두 컴포넌트는 <그림 1>과 [정리4]에서와 같은 순차적인 관계로 연결될 수 있는 컴포넌트임을 알 수 있다. 이 경우 결과적으로 생성되는 컴포넌트 *RoadObstacleAnalyzer*의 인터페이스 서비스 *roadObstacleAnalysis*라 할 때 이는 다음과 같은 추가정보를 갖는다.



<그림 3> 도로장애물분석기 순차도

roadObstacleAnalysis : User × TList × Map → Bool

ext rd validusr : ID × Passwd → Bool

pre $\exists(id,pw) \in (ID,Passwd), \exists tList \in TList, \exists s, d \in Loc, \exists m \in Month \bullet validusr(id, passwd) \wedge \{tList\} \neq \{\} \wedge \{s\} \cap \{d\} = \{\} \wedge \{m\} \neq \{\}$

post $\forall tList \in TList, \forall s,d \in Map \bullet$

$doRoadObstacleAnalysis(s,d,tList,m) = OK \wedge \{tList\} = \{\overrightarrow{tList}\} \wedge ((\{s\} = \{\vec{s}\} \wedge \{d\} = \{\vec{d}\}) \vee \{m\} = \{\vec{m}\})$

roadObstacleAnalysis 명세에서 정의된 선조건과 후조건은 각각 [정의3]과 [정의4]에서 제시된 원리에 기반하여 *roadObstacleAnalysis* 오퍼레리션에서 표현된 선조건, 후조건들이 충실히 보전되도록 정제되어야 한다 이후 단계의 작업은 이렇게 합성된 오퍼레이션 *roadObstacleAnalysis*에 대해 테스트케이스를 생성시키는 작업으로서 컴포넌트 내부에 표현된 선조건과 후조건의 조건분석을 통해 이루어진다. 먼저, 선조건과 후조건에 논리곱 (∧)으로 연결된 조건들을 분해하여 이를 평가할 수 있는 함수 evalT의 정의로 선언한다.

evalT(e1):= $validusr(id, pw)$ = OK

$evalT(e2) := tList \neq \{\}$

$evalT(e3) := \{s\} \cap \{d\} = \{\}$

$evalT(e4) := m \neq \{\}$

$evalT(e5) := doRoadObstacleAnalysis(s,d,tList,m) =$ *OK*

$evalT(e6) := \{tList\} = \{\overrightarrow{tList}\}$

$evalT(e7) := \{s\} = \{\vec{s}\}$

$evalT(e8) := \{d\} = \{\vec{d}\}$

$evalT(e9) := \{m\} = \{\overrightarrow{m}\}$

이를 바탕으로 *roadObstacleAnalysis* 의 선조건에 표현된 조건식에 대한 테스트케이스를 유도한다. *roadObstacleAnalysis* 의 선조건은 논리곱으로만 연결된 단순한 형태이므로 식 (5)에 의해 다음과 같은 카테지안 곱의 관계를 갖는다.

$evalT(e1) \times evalT(e2) \times evalT(e3) \times evalT(e4)$ (25)

$= evalT(e1) \wedge eval\Pi(e2) \wedge eval\Pi(e3) \wedge eval\Pi(e4)$ (26)

$= validusr(id,passwd) \wedge tList \neq \{\} \wedge \{s\} \cap \{d\} = \{\} \wedge \{m\} \neq \{\}$ (27)

후조건부의 논리식을 축약하여 표현하면 식(28)과 같으며 이 식에 대해 식(1)~(12) 중 적절한 규칙을 적용하여 정제를 하면 다음과 같은 과정을 거쳐 식(31)을 유도한다.

$evalT(e5) \times evalT(e6) \times evalT((e7 \wedge e8) \vee e9)$ (28)

$= evalT(e5) \times evalT(e6) \times$

$$\begin{Bmatrix} evalTrue(e7 \wedge e8) \wedge evalTrue(e9) \\ evalTrue(e7 \wedge e8) \wedge evalFalse(e9) \\ evalFalse(e7 \wedge e8) \wedge evalTrue(e9) \end{Bmatrix}$$

(29)

$= evalT(e5) \times evalT(e6) \times$

$$\begin{Bmatrix} evalTrue(e7) \wedge evalTrue(e8) \wedge evalTrue(e9) \\ evalTrue(e7) \wedge evalTrue(e8) \wedge evalFalse(e9) \\ evalTrue(e7) \wedge evalFalse(e8) \wedge evalTrue(e9) \\ evalFalse(e7) \wedge evalTrue(e8) \wedge evalTrue(e9) \\ evalFalse(e7) \wedge evalFalse(e8) \wedge evalTrue(e9) \end{Bmatrix}$$

(30)

$$= \begin{cases} evalT(e5) \wedge eval1T(e6) \wedge eval1T(e7) \wedge eval1T(e8) \wedge eval1T(e9) \\ evalT(e5) \wedge eval1T(e6) \wedge eval1T(e7) \wedge eval1T(e8) \wedge eval1F(e9) \\ evalT(e5) \wedge eval1T(e6) \wedge eval1T(e7) \wedge eval1F(e8) \wedge eval1T(e9) \\ evalT(e5) \wedge eval1T(e6) \wedge eval1F(e7) \wedge eval1T(e8) \wedge eval1T(e9) \\ evalT(e5) \wedge eval1T(e6) \wedge eval1F(e7) \wedge eval1F(e8) \wedge eval1T(e9) \end{cases}$$ (31)

선조건과 후조건으로부터 각각 유도한 식(26)와 식(31)에 대해 카테지안 곱하면 다음과 같다.

$\{evalT(e1) \wedge eval1T(e2) \wedge eval1T(e3) \wedge eval1T(e4)\} \times$ 식 (31) (32)

식 (32)에는 중복되는 조건이 존재한다. 예를 들어 e2의 경우 $tList \neq \{\}$을 의미하지만 이 관계는 이미 공백이 아닌 리스트 간의 변경여부를 확인하는 식 (e6) 즉, $\{tList\} = \{\overrightarrow{tList}\}$ 에 의해 함축되므로 제거할 수 있다. 최종적으로 테스트케이스 생성에 사용되는 관계는 다음의 5가지 조건으로 축약된다.

$$\begin{cases} evalT(e1) \wedge eval1T(e3) \wedge eval1T(e3) \wedge eval1T(e5) \wedge eval1T(e6) \wedge evalT(e7) \wedge evalT(e8) \wedge eval1T(e9) \\ evalT(e1) \wedge eval1T(e3) \wedge eval1T(e3) \wedge eval1T(e5) \wedge eval1T(e6) \wedge evalT(e7) \wedge eval1T(e8) \wedge eval1F(e9) \\ evalT(e1) \wedge eval1T(e3) \wedge eval1T(e3) \wedge eval1T(e5) \wedge eval1T(e6) \wedge evalT(e7) \wedge eval1F(e8) \wedge eval1T(e9) \\ evalT(e1) \wedge eval1T(e3) \wedge eval1T(e3) \wedge eval1T(e5) \wedge eval1T(e6) \wedge evalF(e7) \wedge eval1T(e8) \wedge eval1T(e9) \\ evalT(e1) \wedge eval1T(e3) \wedge eval1T(e3) \wedge eval1T(e5) \wedge eval1T(e6) \wedge evalF(e7) \wedge eval1F(e8) \wedge eval1T(e9) \end{cases}$$ (33)

따라서 개발자는 컴포넌트 *RoadObstacleAnalyzer* 의 구현 시 이상에서 유도된 5 가지의 조건들을 테스트 하여 *roadObstacleAnalysis*명세에서 제시된 조건들이 충실히 구현되었는가를 테스트 할 수 있다.

5평가

본 절에서는 상태 조건식 분해에 의한 테스트케이스 유도결과와 Briand[8]가 제시한 테스트케이스 유도 방법을 비교함으로써 본 논문의 결과를 평가한다. Briand는 사용사례로부터 시스템 테스트를 위한 테스트케이스를 유도하는 방법을 제시했다. Briand의 테스트케이스의 유도 과정을 간략히 설명하면, 1) 순차도에 나타난 메시지 흐름의 분석을 통해 오퍼레이션의 시퀀스를 파악하여 정규수식 (regular expression)으로 표현한다 2) 테스트케이스 생성을 위해 정규수식을 논리곱의

합 형태로 변환한다 3) 순차도에 나타난 조건분기를 분석하여 테스트 드라이버가 수행할 조건의 경로를 파악한다 4) 각 조건의 경로상에 위치하는 오퍼레이션 시퀀스를 찾아 이를 반영하는 의사결정표를 작성한다.

이러한 절차를 바탕으로 <그림 2>에 표현된 오퍼레이션 *roadObstacleAnalysis*의 순차도를 바탕으로 테스트케이스를 생성해 보면 다음과 같다. 단, 오퍼레이션 시퀀스의 표기에 있어 오퍼레이션의 구분을 위해 해당 객체명을 첨자로 함께 표기하며 연결은 도트(.)기호에 의해 구분한다. <그림 2>의 오퍼레이션 2.1, 2.2와 같은 가드(guard) 조건 오퍼레이션들은 합(+)기호에 의해 연결된다. 초기 오퍼레이션 시퀀스로부터 정제를 거쳐 결과적으로 얻어진 시퀀스 집합은 다음과 같다. (간략한 표현을 위해 오퍼레이션의 파라메터는 생략함)

$$
\begin{aligned}
&login_{UsrMgr}.\ exit_{Usr}\\
&+\\
&login_{UsrMgr}.showTools_{RoadAnalyser}.showTools_{ToolSpecMgr}.\\
&showList_{Usr}.input_{RoadAnalyser}.input_{ObstacleAnalyser}.\\
&doAnalysis_{ObstacleAnalyser}.\ showResult_{ObstacleAnalyser}\\
&+\\
&login_{UsrMgr}.showTools_{RoadAnalyser}.showTools_{ToolSpecMgr}.\\
&showList_{Usr}.input_{RoadAnalyser}.input_{ObstacleAnalyser}.\\
&doAnalysis_{RoadAnalyser}.\ showResult_{RoadAnalyser}
\end{aligned}
\tag{34}
$$

위 식이 의미하는 것은 *roadObstacleAnalysis* 순차도는 + 로 연결된 세 개의 오퍼레이션 시퀀스로 분해될 수 있으며 이들은 시퀀스 집합은 다음 <그림2>의 가드조건 2.1, 2.2와 이들로부터 유도된 다음의 조건들에 대한 테스트케이스로 사용된다.

조건 A: UsrMgr.validuser -> exist(u: User |u = true)
조건 B: ObstacleAnalyser.month->
select(m:Month| m != null) (35)

식(34)로부터 유도되는 오퍼레이션의 인스턴스들은 다음과 같다.

- 테스트 시나리오 1(t1): $login_{UsrMgr}.\ exit_{Usr}$
- 테스트 시나리오 2(t2):

$$
\begin{aligned}
&login_{UsrMgr}.showTools_{RoadAnalyser}.showTools_{ToolSpecMgr}.\\
&\quad showList_{Usr}.input_{RoadAnalyser}.input_{ObstacleAnalyser}.\\
&\quad doAnalysis_{ObstacleAnalyser}.\ showResult_{ObstacleAnalyser}
\end{aligned}
$$

- 테스트 시나리오 3(t3):

$$
\begin{aligned}
&login_{UsrMgr}.showTools_{RoadAnalyser}.showTools_{ToolSpecMgr}.\\
&\quad showList_{Usr}.input_{RoadAnalyser}.input_{ObstacleAnalyser}.\\
&\quad doAnalysis_{RoadAnalyser}.\ showResult_{RoadAnalyser}
\end{aligned}
$$

마지막으로 각 조건과 테스트 시나리오에 대한 반응결과인 엑션을 정의해야 한다. <그림 2>의 경우 객체 User가 엑터의 역할을 하므로 User에 대한 응답 및 반응이 엑션의 대상이 되며 따라서 다음과 같이 구분된다.

I: showError;exit()
II: showToolList
III: $showResult()_{ObstacleAnalyser}$
IV: $showResult()_{RoadAnalyser}$

이상의 내용을 바탕으로 테스트에 대한 의사결정표를 구성하면 <표 1>과 같다.

경우	조건		엑션				상태 변경
	A	B	I	II	III	IV	
t1	No	–	Yes	No	No	No	No
t2	Yes	Yes	No	Yes	Yes	Yes	No
t3	Yes	No	No	Yes	No	Yes	No

<표 1> Briand 테스트방법의 의사결정표

본 논문에서 제시한 상태 조건분해 방법에 의해 유도된 테스트케이스를 비교해보도록 한다. 먼저 상태조건분해 방법은 선조건과 후조건의 분해를 통해 테스트 조건을 유도함으로써 <표1>에 나타난 조건보다 더 포괄적인 조건을 산출한다. 식(33)으로부터 생성된 조건들을 살펴보면 다음과 같다.

- 조건I:

$$
\begin{aligned}
&validusr(id,pw)=OK \text{ and } \{s\}\cap\{d\}=\{\} \text{ and } m\neq\{\} \text{ and}\\
&doRoadObstacleAnalysis(tList,s,d,m)=OK \text{ and } \{tList\}=\\
&\{\overrightarrow{tList}\} \text{ and } \{s\}=\{\vec{s}\} \text{ and } \{d\}=\{\vec{d}\} \text{ and } \{m\}=\{\overrightarrow{m}\}
\end{aligned}
$$

- 조건 II:

$$
\begin{aligned}
&validusr(id,pw)=OK \text{ and } \{s\}\cap\{d\}=\{\} \text{ and } m\neq\{\} \text{ and}\\
&doRoadObstacleAnalysis(tList,s,d,m)=OK \text{ and } \{tList\}=\\
&\{\overrightarrow{tList}\} \text{ and } \{s\}=\{\vec{s}\} \text{ and } \{d\}=\{\vec{d}\} \text{ and } \{m\}\neq\\
&\{\overrightarrow{m}\}
\end{aligned}
$$

- 조건III:

$$
\begin{aligned}
&validusr(id,pw)=OK \text{ and } \{s\}\cap\{d\}=\{\} \text{ and } m\neq\{\} \text{ and}\\
&doRoadObstacleAnalysis(tList,s,d,m)=OK \text{ and } \{tList\}=\\
&\{\overrightarrow{tList}\} \text{ and } \{s\}=\{\vec{s}\} \text{ and } \{d\}\neq\{\vec{d}\} \text{ and } \{m\}=
\end{aligned}
$$

$\{\overrightarrow{m}\}$

- 조건IV:

validusr(id,pw)=OK and {s}∩{d}={} and m ≠{} and doRoadObstacleAnalysis(tList,s,d,m)=OK and {tList}= $\{\overrightarrow{tList}\}$ and {s} ≠ $\{\overrightarrow{s}\}$ and {d} = $\{\overrightarrow{d}\}$ and {m} = $\{\overrightarrow{m}\}$

- 조건V:

validusr(id,pw)=OK and {s}∩{d}={} and m ≠{} and doRoadObstacleAnalysis(tList,s,d,m)=OK and {tList}= $\{\overrightarrow{tList}\}$ and {s} ≠ $\{\overrightarrow{s}\}$ and {d} ≠ $\{\overrightarrow{d}\}$ and {m} = $\{\overrightarrow{m}\}$

실제 조건 I에 포함된 논리절 중 evalT(e1), evalT(e4), evalT(e9)은 식 (35)의 조건 A, 조건 B를 포함한다. 즉, 다음의 관계

evalT(e1):= $\exists(id,pw) \in(ID,Passwd)\bullet$ *validusr(id,pw)=OK*

evalT(e4)∧eval(e9):= $\exists m\in Month \bullet m \neq\{\}\wedge\{m\} = \{\overrightarrow{m}\}$

에 의해 조건 A, 조건 B가 함축될 수 있음을 알 수 있다.

선조건과 후조건의 보장을 테스트하기 위해서는 <표 1>에서 나타나지 않은 추가의 조건들이 파악되어야 하며 따라서 이를 테스트하기 위해서는 더 많은 오퍼레이션의 시퀀스를 동원하는 테스트가 이루어져야 한다. <표 2>는 이러한 관찰을 바탕으로 테스트에 대한 의사결정표를 구성한 것이다. (단, t9의 경우 식 *validusr(id,pw)=OK*를 부정함으로써 얻어질 수 있다)

경우	조건					엑 션			
	I	II	III	IV	V	I	II	III	IV
t4	Yes	No	No	No	No	No	Yes	Yes	Yes
t5	No	Yes	No	No	No	No	Yes	No	Yes
t6	No	No	Yes	No	No	No	No	No	No
t7	No	No	No	Yes	No	No	No	No	No
t8	No	No	No	No	Yes	No	No	No	No
t9	No	No	No	No	No	Yes	No	No	No

<표 2> 상태조건분해에 의한 의사결정표

<표 2>를 살펴보면 경우 t4, t5와 t9은 <표1>의 경우와 엑션에 대해 동일한 결과값를 요구함을 알 수 있다. 이와 더불어 t6~8과 같은 추가의 테스트 경우를 명시함으로써 더욱 정교한 테스트를 할 수 있도록 한다. 이러한 이유는 오퍼레이션의 선조건과 후조건에 이미 엑션들이 가져야 할 조건이 모두 포함되었으며, 또한 유스케이스의 시나리오에는 누락되었지만 파라메터들이 만족해야 할 조건들이 추가로 선조건과 후조건에 서술됨으로 써 더욱 엄밀한 경우의 테스트케이스를 생성할 수 있다.

6결론

정확성과 신뢰성이 요구되는 시스템의 개발에 있어 정형기법의 도입은 검증 가능한 시스템 개발을 가능하도록 한다. 특히 컴포넌트 기반 개발 방법에 입각하여 시스템을 개발할 경우 추상화된 수준의 명세로부터 시작하여 구체적인 명세가 나오도록 분해를 하는 활동을 수행한다. 이러한 개발활동이 올바르다는 것을 보이기 위해서는 적절한 수준의 명세간의 일치성을 보여주는 검증을 필요로 한다.

본 논문에서는 정형적 정제활동이 요구분석뿐 아니라 설계단계에 까지 일관되게 적용된다는 점에 착안하여 요구분석 단계에서 산출된 상태기반 정형명세로부터 테스트케이스를 생성하는 방법을 제시하였다. 또한 이 과정에서 기존의 유스케이스 시나리오에 기반한 테스트케이스 생성방법과 비교하여 상태조건에 기반한 테스트케이스케이스 생성방법의 결과를 도로장애물 분석기의 예를 통해 비교하였다.

상태조건에 기반한 테스트케이스 생성의 경우 유스케이스에 의한 방법보다 정교한 테스트케이스를 생성한다. 그러나 컴포넌트 내부의 논리흐름에 대한 커버리지의 확인이 테스트의 목적일 경우는 메시지 시퀀스의 추적을 통한 테스트를 수행하는 유스케이스 기반의 테스트가 효과가 있고, 블랙박스 테스트일 경우 상태조건에 기반한 테스트가 더욱 효과적이라 판단된다. 따라서 이 두 방법은 목적에 따라 서로 보완적으로 사용될 수 있을 것이다.

기능의 명세를 함에 있어 본 본문이 취한 접근방법은 상태모형에 기반한 선조건과 후조건의 논리절 분석이다. 만일 명세가 페트리넷, 혹은 CSP와 같은 프로세스 기반으로 서술되었을 경우 이들 명세는 프로세스 대수에 이론적 근거를 두고 있으므로 직접적인 선조건과 후조건의 유

도가 곤란한 경우가 대부분이다. 따라서 이 경우는 본 논문에서와 같은 테스트케이스 유도 방법은 직접적으로 적용하기 곤란한 한계를 가진다. 향후에는 프로세스 모형과 상태기반 모형 사이의 조건 일치성 관계를 파악함으로써 테스트케이스 생성기법을 확장시킬 예정이다.

참고문헌

[1] 송영재, 김귀정 외, 객체지향모델링과 CBD 중심의 소프트웨어공학, p.12, 이한출판사, 2004

[2] Brooks T,. Fitzgerald J., Larson P., Formal and informal specifications of a security System Component: Final Results in a comparative study, FME 96, LNCS 1051, pp214-227, Springer, 1996

[3] Alencar A., Goguen J., OOZE : An Object-Oriented Z Environment, ECOOP '91 Proceedings, LNCS 512, pp.180-199, Springer-Verlag, 1991.

[4] Durr E., VDM++ Language Reference Manual, Afrodite Report AFRO/CG/ED/LRM/V11, 1995.

[5] Jones C. B., Systematic Software Development using VDM, 2nd ed. Prentice Hall, 1990

[6] Woodcock J., Using Z-Specification, refinement and Proof, Oxford University Press, 1990

[7] Binder R., Testing Object-oriented Systems, Addison Wesley, 2000

[8] Briand L., Labiche Y., A UML-based Approach to System Testing, Carleton University, TESCR-01-01, 2002

[9] Meudec C., Automatic Generation of Software Test Cases from Formal Specifications, PhD thesis, Queen's University of Belfast, 1998

[10] Gries D., The Science of Programming, Springer-Verlag, p.109 1981

[11] Stavely A., Toward Zero-defect Programming, Addison-Wesley, 1999

[12] Hinchey M., Bowen J., Application of Formal Methods, Prentice-Hall, 1995

서 동 수

1986 중앙대학교 컴퓨터공학과 (이학사)
1990 영국 맨체스터 이공대학 전산학과 (이학석사)
1994 영국 맨체스터 이공대학 전산학과 (공학박사)
1994~1998 한국전자통신연구원 선임연구원
1998~현재 성신여자대학교 컴퓨터정보학부 부교수
관심분야: 소프트웨어공학, 정형기법, 정보보호
Email: dseo@sungshin.ac.kr