연속 메소드 규칙을 이용한 객체간의 행위적 호환성 검증 기법

(An Approach to Verifying Behavioral Compatibility between Objects using Successive Methods Rule)

채 흥 석 † 이 준 상 †† 배 정 호 †††
(Heung Seok Chae) (Joon Sang Lee) (Jung Ho Bae)

요 약 객체지향 시스템의 객체들은 서브타입과 수퍼타입의 관계로 구성될 수 있다. 즉 서브타입은 수퍼타입의 구조와 행위를 상속하고 특화하는 방식의 계층적 구조를 가진다. 기존 시스템의 확장과 진화를 가능케 하기 위해서 수퍼타입과 서브타입간의 행위적 호환성은 둘 간의 대체가능성을 위하여 중요한 문제이다. 본 논문에서는 객체의 동적 행위를 기술하는 범용적인 표현 방법 중 하나인 유한상태기계로 표현된 객체의 동적 행위를 바탕으로 객체간의 행위적 호환성을 검사할 수 있는 기법인 메소드 규칙을 확장한 연속 메소드 규칙을 제시한다. 연속 메소드 규칙은 기존의 메소드 규칙(methods rule)을 바탕으로 두 객체의 자취를 조사함으로써 두 객체가 행위적 호환성이 있음을 보장한다. 또한 연속 메소드 규칙을 이용하여 행위적 호환성을 검증할 수 있는 알고리즘을 제시한다.

키워드 : 행위 호환성, 서브타입

Abstract In object-oriented systems, objects are organized in hierarchies such that subtypes inherit and specialize the structure and the behavior of supertypes. Behavioral compatibility is a very crucial issue to permit the substitution between object types, which supports the extension and evolution of object oriented system. This paper proposes successive methods rule that extending methods rule for checking behavioral compatibility between objects on the basis of their dynamic behaviors expressed in finite state machine which is one of the most frequently used notations for expressing dynamic behaviors of object. Based on the classical methods rule, successive methods rule is used for guarantee behavioral compatibility by checking the traces of two objects. And we describe an algorithm for verifying behavioral compatibility between objects using the successive methods rule.

Key words : Behavioral compatibility, subtype

1. 서 론

객체지향 시스템은 서로 협업하는 수많은 객체로 구성된다. 한 객체의 관점에서 보면, 다른 객체들은 그 객체가 동작하는 환경이라고 볼 수 있다. 즉 객체에게 요구되는 행위는 해당 객체 단독이 아니라 주변의 다른 객체와의 원활한 협업을 통해서만이 가능하다. 시스템 개발 초기 단계에서 각 객체간의 협업은 명확히 정의되고, 구현될 수 있다. 뿐만 아니라, 유지보수 단계에서는 소프트웨어에게 요구되는 기능의 확장 또는 추가를 충족시키기 위해서 이와 관련된 객체가 진화될 수 있다.

이 때 효율적인 유지보수를 위해서는 진화 또는 변화된 객체가 기존 시스템의 관련된 주변 객체들과 여전히 원활한 상호작용을 할 수 있어야 한다. 즉 새롭게 정의된 객체가 기존의 객체 대신에 사용된다 하더라도 기존에 이미 제공되던 시스템의 동작에는 영향을 미치지 않아야 한다. 이를 위해서는 한 객체가 다른 객체 대신에 사용되어 새 객체가 기존 객체의 행위를 안전하게 대체할 수 있는 지에 대한 점검이 필요하다.

이와 같이 두 객체를 행위적인 측면에서 대체가능성을 조사하는 문제를 행위적 호환성(behavioral comp-

· 본 연구는 정보통신부 및 정보통신연구진흥원의 대학 IT연구센터(IITA-2006-(C1090-0603-0032))와 방위사업청과 국방과학연구소의 지원사업의 연구결과로 수행되었음
† 정 회 원 : 부산대학교 컴퓨터공학과 교수
hschae@pusan.ac.kr
†† 정 회 원 : 고려대학교 임베디드 소프트웨어학과 교수
windyscar@korea.ac.kr
††† 학생회원 : 부산대학교 컴퓨터공학과
jhbae83@pusan.ac.kr
논문접수 : 2007년 1월 22일
심사완료 : 2007년 6월 9일

atibility)라고 부른다. 행위적 호환성은 두 객체의 행위에 대한 명세의 일관성을 단순히 점검하는 구문적 호환성(syntactical compatibility)과는 의미상 많은 차이가 있다. 구문적 호환성은 객체의 행위에 대하여 시그너처와 유사한 방식의 명세를 바탕으로 문법적인 일관성만을 조사하는 방법으로서 간단한 정적 분석 기법으로 점검이 가능한 반면에 시스템의 실제 동작 환경에서의 기능상의 호환성을 보장하지 못하는 한계가 있다. 그러나 행위적 호환성은 객체의 동작 환경이 고려되는 방식으로서 구문적 호환성에 비해 보다 정확한 분석 결과를 얻을 수 있다.

행위적 호환성 문제는 객체지향 언어가 제공하는 상속과 유사한 측면이 있다. 즉, 하위클래스는 상위클래스의 구조(structure) 뿐만 아니라 행위(behavior)까지 물려받기 때문에 상위클래스의 객체가 사용될 환경에서 하위클래스의 객체가 대체되어 사용될 수도 있다. 상속 관계는 객체지향언어에서 요구되므로 컴파일러와 같은 정적 분석 도구를 이용하여 구문적 호환성의 준수 여부를 쉽게 조사할 수 있다. 그러나 상속의 사용이 행위적인 측면에서 호환성을 제공한다고 보장할 수는 없다. 즉 하위클래스의 객체가 상위클래스의 객체를 대신하여 사용될 때 주변 환경에 부정적인 영향을 미치지 않으면서 상위클래스의 객체에게 기대되는 동작을 제공할 수 있는 지에 대한 확인이 이루어지지는 않는다. 따라서 상속 관계만으로는 객체 간의 행위적 호환성을 보장할 수는 없는 것이다. 상속과 달리 행위적 호환성을 가정하는 개념으로 서브타이핑(subtyping)이 사용된다. 서브타이핑은 일반적으로 하위타입(subtype)의 객체가 상위타입(supertype)의 객체 대신에 대체 사용될 수 있으며 행위적 호환성도 함께 제공되는 경우를 일컫는다. 따라서 두 객체간의 행위적 호환성 점검은 두 객체의 타입간에 서브타이핑 관계가 있는 지에 대한 점검을 의미하는 것으로 간주될 수 있다[1-5].

상위타입과 하위타입간의 행위적 호환성을 점검하기 위한 기준으로서 Ebert와 Engels[3]는 관찰 일관성(observation consistency)와 호출 일관성(invocation consistency)를 제시하였다. 관찰 일관성은 하위타입의 객체에서 관찰될 수 있는 연산의 연속(sequence of operations)이 대응되는 상위타입의 객체에서도 관찰이 되어야 함을 뜻한다. 반면에 호출 일관성은 상위타입의 객체에서 호출될 수 있는 연산의 연속이 대응되는 하위타입의 객체에서도 같은 방식으로 허용되는 것을 뜻한다. 본 논문에서 다루는 행위적 호환성은 상위타입의 객체 위치에 하위타입의 객체가 사용되는 경우에도 하위타입의 객체가 주변 객체와 여전히 올바른 상호작용을 할 수 있는 특성을 의미한다. 이런 의미에서 Ebert와 Engles가 제시한 두 가지 일관성 중에서 호출 일관성에 해당한다 하겠다.

본 논문에서는 객체의 행위를 나타내기 위한 범용적인 표현 방법인 UML의 상태 다이어그램과 유사한 유한상태기계(Finite State Machine)로 표현된 객체의 행위를 바탕으로 Ebert와 Engel가 제시한 개념인 호출 일관성에 기반하여 상위타입과 하위타입간의 행위적 호환성을 점검하는 방법을 소개한다. 상태기계는 테스팅[6], 정형적 검증, 코드 생성 및 역공학[7-9] 등과 같은 분야에서 쓰이고 있다. 상태기계로부터 연산들의 수행 가능한 모든 순서를 파악할 수 있으므로, 상태기계를 바탕으로 클래스를 테스팅 하는 다양한 연구가 수행되고 있다. 클래스의 객체 모델은 테스트 케이스의 자동 생성[10, 11] 뿐만 아니라, 실시간성[12], 안정성(safety)[13,14] 등의 다양한 정형적 검증의 바탕이 되고 있다[15]. 또한 클래스의 정적 모델을 통해서는 클래스의 속성과 연산에 대한 선언만이 코드로서 생성될 수 있는 반면 상태기계와 같은 동적 모델을 통해서는 연산의 구현 코드를 전체 또는 부분적으로 자동 생성하는 것이 가능하다[16, 17]. 뿐만 아니라, 복잡한 소스코드의 이해를 돕기 위해서 소스코드로부터 동적 모델을 자동으로 추출하는 연구도 진행되고 있다[7-9]. 이와 같이 상태기계는 다양한 분야에서 사용되고 있으며 본 논문에서도 객체간의 행위적 호환성을 검증하는 방법을 사용하기 위한 모델로서 유한 상태기계를 사용한다.

행위적 호환성 또는 서브타이핑을 다루는 대표적인 연구로는 Loskov와 Wing[4]이 제안한 메소드 규칙(methods rule)이 있다. 메소드 규칙은 두 메소드간의 서브타이핑을 점검할 때 하위타입에서 상위타입의 선행조건은 약화되고 후행조건은 강화됨을 뜻한다. 본 논문에서는 Loskov와 Wing에 제시한 메소드 규칙(methods rule)을 확장하여 조건을 가진 유한상태기계로 표현된 두 타입간의 행위적 호환성을 점검하는 방법을 소개한다.

본 논문의 구성은 다음과 같다. 2절에서는 객체의 동적 행위를 표현하기 위한 유한상태기계를 간단히 소개하고, 간단한 예제를 이용하여 호출 일관성을 설명한다. 3절에서는 기존의 메소드 규칙의 단점을 기술하고 이를 확장하여 두 타입간의 호출 일관성을 점검하는 방법을 설명한다. 4절에서는 행위적 호환성과 관련된 기존의 연구를 소개하고 5절에서는 결론과 향후 연구 방향을 기술한다.

2. 연구 배경

2.1 객체의 동적 행위 표현

이 절에서는 객체의 동적 행위를 표현하기 위하여 사용되는 상태 기계 및 관련 몇몇 정의를 소개한다.

정의 1. 객체의 동적 행위는 상태 기계 SM = (S, s_0,

S_Ψ, I, O, σ)로 표현된다.

- S: 상태의 집합으로서 공집합이 아니다.
- I: 입력 이벤트들의 집합으로서 객체가 수신하여 처리할 수 있는 메소드들의 집합에 해당된다.
- O: 출력 이벤트들의 집합이다.
- σ: S X I X O → S. 두 상태간의 전이 함수이다. 전이 t가 (s, i, o, d)로 표현될 때 *source*(t) = s, *input*(t) = i, *output*(t) = o, *sink*(t) = d를 뜻하는 것으로 가정한다. 본 논문에서는 전이의 실행은 결정적(deterministic)임을 가정한다. 즉 동일한 시작 상태에서 동일한 입력 이벤트를 받았을 때 상이한 출력 이벤트를 산출하는 전이는 없는 것으로 가정한다. 즉 $s_1 = s_2 \wedge i_1 = i_2 \wedge o_1 \neq s_2$인 전이 $t_1=(s_1, i_1, o_1, d_1)$과 전이 $t_2=(s_2, i_2, o_2, d_2)$는 없는 것을 가정한다.
- s_0는 S의 원소로서 초기 상태를 뜻한다. 즉, $\forall t \in \sigma \cdot \text{sink}(t) \neq s_0$
- S_Ψ: 최종 상태(final state)로서 S의 부분집합이다. 즉 $\forall s \in S_\Psi \cdot \forall t \in \sigma \cdot \text{source}(t) \neq s$

정의 2. 진입 전이(incoming transition)와 진출 전이(outgoing transition)

*in_trans(s)*는 상태 s로 진입하는 전이들의 집합이다. 즉 in_trans(s) = {t ∈ σ | sink(t) = s}이다. 유사한 방법으로 *out_trans(s)*는 s로부터 진출되는 전이들의 집합이다. 즉 out_trans(s) = {t ∈ σ | source(t) = s}이다.

정의 3. 선행 조건(pre-condition)과 후행 조건(post-condition)

전이 t에는 선행 조건 *pre(t)*와 후행 조건 *post(t)*가 정의되어 있다. 선행 조건은 해당 전이가 발생하기 위해서 충족되어야 할 조건이며 후행 조건은 전이의 발생 후에 만족되어야 하는 조건을 뜻한다.

객체의 행위는 개별적인 행위의 단위인 전이들의 나열로 표현될 수 있다. 그리고 객체의 전체 행위는 이러한 전이 나열의 가능 집합으로 표현될 수 있다. 본 논문에서는 객체가 보여주는 하나의 행위를 자취(trace)로 정의한다.

정의 4. 객체의 자취(trace)

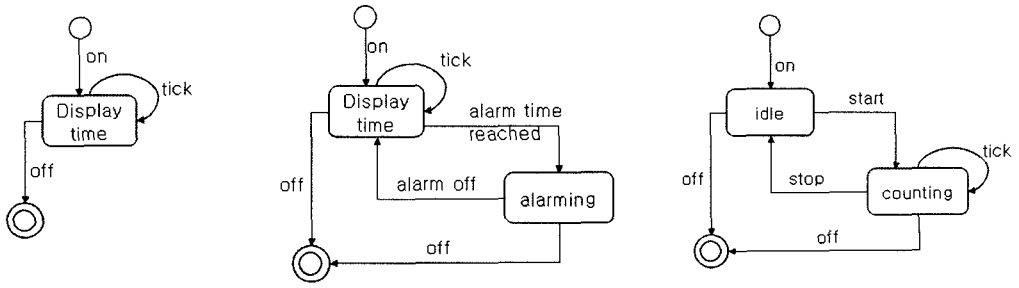
자취(trace)는 객체가 보여주는 한가지 행위로서 일련의 전이에 의해서 정의된다. 객체 타입 *ot*의 자취는 $t_1 \cdot t_2 \cdots \cdot t_n$ ($t_i \in \sigma_{ot}$)로서 표현된다.

2.2 호출 일관성

간단히 말해서 호출 일관성은 상위타입의 객체에서 가능한 모든 자취가 하위 타입의 객체에서도 가능해야 함을 뜻한다. 호출 일관성을 간단한 예를 통해서 설명하겠다. 그림 1은 3가지 종류의 시계에 대한 행위를 유한 상태기계로 보여 주고 있다. 이 예는 호출 일관성을 설명하기 위한 것이므로 조건이 없는 단순한 형태의 유한 상태기계를 이용하여 시계의 행위를 표현하였다.

그림 1(a)는 가장 단순한 행위를 가진 *Clock*으로서 *on* 이벤트 수신 후에 *tick* 이벤트 마다 시간을 보여주다가 *off* 이벤트에 행위를 종료한다. 그림 1(b)는 *Alarm Clock*의 행위로서 *Clock*의 기능을 확장하여 지정된 경보 시간이 되었을 때(*alarm time reached*) 경보가 울리다가 *alarm off* 이벤트에 경보를 중단하는 행위를 보여준다. 그림 1(c)는 스톱워치의 행위를 보여주는 상태기계로서 *start* 이벤트 후에 *stop* 이벤트 또는 *off* 이벤트가 수신될 때까지 *tick* 이벤트 마다 시간 경과를 계산하는 기능을 새롭게 포함한다.

*Clock*과 *Alarm Clock*의 상태 기계를 분석해 보면 *Clock*이 상위타입이고 *Alarm Clock*이 하위타입으로서 두 타입간에 행위적 호환성 즉 호출 일관성이 충족됨을 알 수 있다. 즉 *Clock*에서 가능한 행위가 하위타입인 *Alarm Clock*에서도 가능하다. *Clock*의 행위는 자취로서 표현하면 $on \cdot (tick)^* \cdot off$가 되며 이 자취는 하위타입인 Alarm Clock에서도 수행 가능하다. 반면에 *Clock*의 행위 중에서 일부는 *Stopwatch*에서는 불가능하다. 예를 들어, $on \cdot tick \cdot off$는 *Clock*에서는 가능한 자취이



(a) Clock (b) Alarm Clock (c) Stopwatch

그림 1 호출 일관성의 예

다. 그러나 *Stopwatch*에서는 *tick* 이벤트를 수신하기 위해서는 먼저 *start* 이벤트가 수신되어야 하므로, *on · tick · off* 행위는 *Stopwatch*에서는 가능하지 않다. 따라서 *Stopwatch*는 *Stop*과 호출 일관성을 충족시키지 않으며 행위적 호환성을 가진다고 말할 수 없다. 따라서 세 개의 유사한 객체간의 호출 일관성을 바탕으로 *Alarm Clock* 객체는 *Clock* 객체의 자리에 대신 사용될 수가 있는 반면에 *Stopwatch*는 *Clock* 대신에 사용될 수 없음을 판단할 수 있다.

요약하면 호출 일관성은 상위타입에서 호출 가능한 행위가 하위타입에서도 유지되어야 함을 뜻한다. 이를 자취 개념을 이용하여 정의하면 다음과 같다.

정의 5. (호출 일관성)

T_{ot1}과 T_{ot2}를 객체 타입 ot_1과 ot_2에서 가능한 자취의 집합이라고 할 때, $T_{ot1} \subseteq T_{ot2}$일 때 타입 ot_2는 타입 ot_1과 호출 일관성을 충족시킨다고 말하며 $ot_2 \Rightarrow^{\circ} ot_1$로 표현한다. 즉

$ot_2 \Rightarrow^{\circ} ot_1 \equiv T_{ot1}$의 각 자취에 대해서 대응되는 자취를 T_{ot2}도 *포함한다.*

3. 행위적 호환성의 점검

이 절에서는 2절에서 정의한 기본 개념을 바탕으로 행위적 호환성 즉 호출 일관성을 점검하는 방법을 소개한다. 먼저 유한상태기계로 표현된 두 객체 타입의 호출 일관성 검사를 위해서 기존의 메소드 규칙의 단점을 설명하고 이를 개선한 방법을 소개한다. 그리고 제시된 검사 방법이 호출 일관성을 준수할 수 있음을 간단히 증명한다.

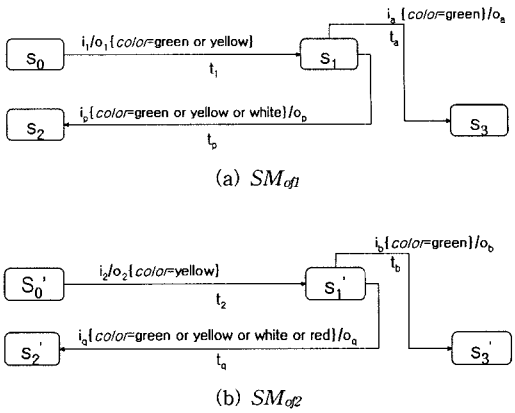
3.1 기본 개념

먼저 두 유한상태기계의 초기 상태간의 대응(correspondence), 전이간의 대응, 상태간의 대응 개념을 소개하고 기존의 메소드 규칙과 이를 개선한 메소드 규칙을 제시한다.

정의 6. 초기상태 간의 대응

두 객체 타입 ot_1과 ot_2의 초기 상태는 상호 대응되는 것으로 정의된다. 즉 $SM_{ot1}.s_0 \Rightarrow SM_{ot2}.s_0$ 이고 반대로 $SM_{ot2}.s_0 \Rightarrow SM_{ot1}.s_0$ 이다. 여기서 $SM_{oti}.s_0$는 객체 타입 ot_i의 초기 상태를 뜻하고 $s_2 \Rightarrow s_1$는 상태 s_2가 상태 s_1에 대응됨을 나타낸다.

그림 2는 두 객체 타입 ot_1과 ot_2의 행위를 표현하는 유한상태기계 SM_{ot1}과 SM_{ot2}를 보여 준다. 이 유한상태기계에서 전이는 '*입력 이벤트 {선행 조건} / 출력 이벤트 {후행 조건}*'의 형식으로 표시된다. 선행 조건 또는 후행 조건의 표시가 누락된 전이는 선행 조건/후행 조건이 참임을 가정한다. 그림에서 전이 하단에 표시된 t_1, t_2, t_a, t_b, t_p, t_q는 각 전이를 고유하게 지칭하기 위하여 사용되는 이름이다. 두 상태기계에서 S_0와 S_0'은 각각 SM_{ot1}과 SM_{ot2}의 초기 상태이다. 따라서 정의1에 의해서 $S_0' \Rightarrow S_0$ 이며 $S_0 \Rightarrow S_0'$ 이다.



(a) SM_{ot1}

(b) SM_{ot2}

그림 2 호출 일관성을 설명하기 위한 예

정의 7. 전이 간의 대응

다음의 두 조건이 만족할 때 전이 $t_2 \in o_{ot2}$는 전이 $t_1 \in o_{ot1}$에 대응되며 $t_2 \Rightarrow t_1$으로 표시한다.

- source(t_2) $\Rightarrow$ source(t_1)
- SM_{ot1}과 SM_{ot2}의 입력 이벤트 집합 I_{ot1}과 I_{ot2}, 출력 이벤트 집합 O_{ot1}과 O_{ot2} 간에 매핑이 존재한다.

그림 2에서 전이 t_2는 전이 t_1에 대응된다. 우선 정의 6에 의해서 source(t_2)= S_0'은 source(t_1)= S_0에 대응된다 즉 $S_0' \Rightarrow S_0$ 이다. 또한 전이 t_2와 t_1의 입/출력 이벤트 집합간에는 (i_2, i_1)과 (o_2, o_1)의 매핑이 가능하다.

정의 8. 상태 간의 대응

만약 상태 s_1의 진입 전이 집합의 각 전이에 대응되는 전이가 상태 s_2의 진입 전이 집합에 있다면 상태 s_2는 상태 s_1에 대응된다고 정의하며 $s_2 \Rightarrow s_1$으로 나타낸다. 즉

$$s_2 \Rightarrow s_1 \equiv \forall t_1 \in \text{in\_trans}(s_1),$$
$$\exists\ t_2 \in \text{in\_trans}(s_2)\ [t_2 \Rightarrow t_1]$$

예를 들어 그림 2에서 상태 s_1'는 상태 s_1에 대응될 수 있다. 상태 s_1은 유일한 진입 전이 t_1을 가지고 있으며, 상태 s_1'은 이에 대응되는 t_2를 진입 전이로서 가지고 있기 때문이다. $t_2 \Rightarrow t_1$에 대해서는 정의 7에서 예제로서 설명을 하였다.

정의 9. 메소드 규칙(조건이 있는 전이 간의 대응)

다음의 두 조건이 만족할 때 전이 $t_2 \in o_{ot2}$는 전이 $t_1 \in o_{ot1}$에 메소드 규칙 관점에서 대응되며 $t_2 \Rightarrow^{m} t_1$으로 표시한다.

- $t_2 \Rightarrow t_1$

$\Rightarrow$ (pre(t_1) $\rightarrow$ pre(t_2)) $\wedge$ (post(t_2) $\rightarrow$ post(t_1))

이 정의는 조건을 가진 두 전이 간의 대응 관계에 해

당된다. 조건을 가진 전이를 비교할 때는 정의 7에서 제시된 단순히 전이의 시작 상태와 입/출력 이벤트의 대응만으로는 부족하며 전이에 부가된 선행/후행 조건간의 관계에 대한 추가적인 고려도 필요하다. 선행/후행 조건에 관련된 기준으로는 Meyer[5]가 제시한 규칙을 활용하고자 한다. Meyer는 하위클래스의 메소드는 상위클래스 메소드의 선행조건은 약화시키고 후행조건은 강화시킬 수 있어야 두 클래스 간에 행위적 호환성이 존재할 수 있다고 주장하였다. Liskov[4]는 Meyer가 제시한 선행조건의 약화 규칙을 선행 조건 규칙(pre-condition rule), 후행조건의 강화 규칙을 후행조건 규칙(post-condition rule)이라고 하였으며, 이 두 규칙을 한꺼번에 메소드 규칙(methods rule)이라고 명명하였다.

본 논문에서는 Meyer가 제시하고 Loskov가 구체화시킨 메소드 규칙을 조건을 가진 전이 간의 대응 관계를 정의할 때 사용한다. 정의 9의 두 번째 조건은 선행조건은 약화되고 후행조건은 강화됨을 의미한다. 예를 들어 정의 7에서 언급된 전이 t_2와 전이 t_1간의 대응을 메소드 규칙을 추가적으로 고려하면 다음과 같다. 정의 7에서 말한 것처럼 $t_2 \Rightarrow t_1$ 은 이미 충족되고 있으므로 선행/후행 조건만을 고려하면 된다. 그림 2에서 볼 수 있듯이 t_1의 선행 조건과 t_2의 선행 조건은 모두 *true*로서 선행조건의 약화 규칙을 준수하는 것으로 간주될 수 있다. 수행조건을 살펴보면 t_1의 후행조건은 *{color = green or yellow}*이고 t_2의 후행조건은 *{color = yellow}* 이므로 후행조건은 t_2에서 강화되고 있다. 따라서 t_2은 메소드 규칙 측면에서도 t_1에 대응된다고 간주될 수 있다. 마찬가지로 전이 t_q는 t_p에 메소드 규칙 면에서 대응될 수 있다. s_1'은 전이 t_q의 출발 상태(source state)로서 t_p의 출발 상태인 s_1에 대응되며, 입/출력 이벤트 집합간에는 (i_p, i_q)와 (o_p, o_q)의 매핑이 존재할 수 있다. 또한 전이 t_p의 선행 조건인 *{color = green or yellow or white}*은 대응 전이 t_q의 선행 조건 *{color = green or yellow or white or red}*에서 약화되고 있다. 두 전이의 후행 조건은 모두 참이므로 마찬가지로 후행조건은 강화된다고 볼 수 있다.

3.2 연속 메소드 규칙

기존의 메소드 규칙을 호출 일관성을 점검하기 위한 방법으로서 사용하는 것을 고려할 수 있다. 즉 SM_{ot1}의 모든 자취 ($t_1 \cdot t_2 \cdots \cdot t_n$)에 대해서 SM_{ot2}에 자취 ($t_1 \cdot t_2 \cdots \cdot t_n$)가 $t_i' \Rightarrow^m t_i$ ($1 \le i \le n$)의 조건을 충족시킨다면 SM_{ot2}이 SM_{ot1}에 대해서 호출 일관성을 충족시키는 것으로 판단할 수 있다. 예를 들어, SM_{ot1}의 자취 $t_1 \cdot t_p$는 SM_{ot2}의 자취 $t_2 \cdot t_q$와 대응될 수 있다. 즉 $t_2 \Rightarrow^m t_1$ 이고 $t_q \Rightarrow^m t_p$ 이다.

그러나 위와 같이 기존의 메소드 규칙을 두 자취를 구성하는 각 전이의 쌍에 적용하는 것만으로는 호출 일관성을 보장할 수가 없다. 예를 들어 또 다른 자취로서 SM_{ot1}의 자취 $t_1 \cdot t_a$가 SM_{ot2}의 자취 $t_2 \cdot t_b$에 대응되는 것으로 생각될 수 있다. 즉 ot_2의 $t_2 \cdot t_b$가 ot_1의 행위인 $t_1 \cdot t_a$를 대체하는 것으로 판단될 수 있다. 그러나 전이 t_2의 후행조건 *{color = yellow}* 과 전이 t_b의 선행조건 *{color = green}*으로부터 $t_2 \cdot t_b$는 실제로는 수행될 수 없는(infeasible) 자취이다. 그러므로 자취를 구성하는 대응되는 전이의 선행/후행조건 규칙만으로는 행위적 호환성을 보장하지 못한다. 본 논문에서는 이러한 단점을 극복하기 위하여 기존의 메소드 규칙을 보완한 *연속 메소드 규칙(successive methods rule)*을 제시하고자 한다.

연속 메소드 규칙은 연속하는 2개의 메소드 쌍을 대상으로 상위의 자취가 실행 가능할 때 하위에서 그에 대응되는 실행 가능한 자취의 존재를 보장하도록 정의되었다. 위에서 기술한 문제 즉 t_2의 후행조건이 t_b의 선행조건을 충족시키지 않아서 $t_2 \cdot t_b$가 수행 불가능한 자취가 되는 것을 방지하기 위해서는 t_2 이후에 t_b의 수행이 보장되어야 한다. $t_2 \cdot t_b$의 보장은 t_2의 후행조건이 t_b의 선행조건을 충족시키면 가능하다. 즉 (post(t_2) → pre(t_b))이 추가적인 조건으로 포함되어야 한다. 하위타입의 $t_2 \cdot t_b$의 보장에 대한 필요성은 근본적으로 상위타입에서 $t_1 \cdot t_a$이 수행 가능할 때만 고려하면 된다. 즉 수행 불가능한 $t_1 \cdot t_a$에 대해서 대응되는 $t_2 \cdot t_b$를 고려할 필요가 없기 때문이다.

또한 상위타입에서 $t_1 \cdot t_a$의 수행 가능성은 전이 t_1와 전이 t_a의 선행/후행 조건에 따라서 다음 두 가지 경우에 $t_1 \cdot t_a$의 수행이 가능하다.

1) 상태 독립적(state independent) 수행 조건: post(t_1) → pre(t_a)

만약 t_1의 후행조건이 t_a의 선행 조건을 충족시킨다면, 각 전이의 선행/후행 조건을 구성하는 변수의 값에 무관하게 항상 전이 t_a는 전이 t_a 뒤에 수행되는 것이 보장될 수 있다. 그림 2(a)에서 전이 t_1의 후행조건 *{color = green or yellow}*와 전이 t_p의 선행조건 *{color = green or yellow or white}*가 이 경우에 해당된다. 즉 전이 t_1의 후행조건이 충족된다면 color의 실제 값에 관계 없이 전이 t_p의 선행조건은 충족된다. 따라서 객체 ot_1의 실제 상태에 독립적으로 $t_1 \cdot t_p$의 자취는 수행 가능하다.

2) 상태 의존적(state dependent) 수행 조건: pre(t_a) → post(t_1)

상태 독립적 수행의 경우와 달리 선행/후행 조건을 구성하는 변수의 값 즉 객체의 상태에 따라서 조건적으로 다음 전이가 수행되는 경우이다. 즉 전이 t_a의 선행조건이 t_1의 후행조건을 충족시키는 경우에는 객체의 상

태에 따라서 t_a가 t_1 이후에 수행될 수도 있고 수행되지 못할 수도 있다. 예를 들어 그림 2(a)에서 $t_1 \cdot t_a$ 전이를 살펴 보면 t_a의 선행조건 *{color = green}*은 t_1의 후행조건 *{color = green or yellow}*를 충족시키고 있다. 이 경우에 전이 t_1 이후에 전이 t_a의 실제 수행 여부는 *color*의 값에 영향을 받는다. *color*의 값이 *green* 인 경우에는 t_a가 수행되겠지만 *color*가 *yellow*인 경우에는 t_a의 수행은 불가능하다.

본 논문에서는 상위타입의 자취 $t_1 \cdot t_a$가 수행 가능한 경우에 상태 독립적/의존적 수행 조건에 관계없이 하위 타입에서 항상 수행 가능한 자취 $t_2 \cdot t_b$를 보장할 수 있도록 기존의 메소드 규칙을 보완한 연속 메소드 규칙을 제안한다.

정의 10. 연속 메소드 규칙(successive methods rule) 다음의 두 조건이 만족할 때 전이 $t_2 \in o_{ot2}$는 전이 $t_1 \in o_{ot1}$에 연속 메소드 규칙 관점에서 대응되며 $t_2 \Rightarrow^s t_1$으로 표시한다.

1) $t_2 \Rightarrow^m t_1$

2) $\forall t_a \in$ out_trans(sink(t_1)), $\exists t_b \in$ out_trans (sink(t_2)) [$t_b \Rightarrow^m t_a$]에 대해서

2.1) 상태 의존적인 경우 : (pre(t_a) → post(t_1)) ∧ (post(t_2) → pre(t_b)) 또는

2.2) 상태 독립적인 경우 : post(t_1) → pre(t_a)

연속 메소드 규칙은 각 메소드 쌍이 우선 기존의 메소드 규칙을 준수해야 하며 추가적으로 위에서 기술한 문제를 보완하기 위한 조건을 추가하였다. 즉 연속 메소드 규칙은 대응 대상이 되는 상위 타입의 전이 t_1과 하위타입의 t_2에 대해서는 기존의 메소드 규칙을 그대로 적용한다 (정의 10의 1) 부분). 그리고, 상위 타입의 자취 상에서 t_1 다음에 수행 가능한 전이 t_a에 대해서 하위 타입에서 t_2 다음에 항상 수행 가능한 메소드 규칙을 준수하는 전이 t_b의 존재를 보장하고 있다 (정의 10의 2) 부분).

정의 10. 2) 부분은 상위 타입의 전이 t_1 이후의 모든 전이 t_a에 대해서 하위 타입에서 대응되는 전이 t_b가 t_2 뒤에 있으면서 t_b가 t_a를 기존의 메소드 규칙을 준수해야 함을 뜻한다. 뿐만 아니라, 상위 타입의 $t_1 \cdot t_a$의 자취에 대해서 하위 타입에서 $t_2 \cdot t_b$를 보장하기 위해서는 추가적으로 2.1)과 2.2)를 요구하고 있다. 기본적으로 $t_2 \cdot t_b$를 보장하기 위해서는 t_2의 후행조건이 t_b의 선행조건을 충족시켜야 한다. 즉 post(t_2) → pre(t_b)가 만족되어야 한다.

- 2.1) 부분은 자취 $t_1 \cdot t_a$가 객체의 상태에 따라서 수행 여부가 결정되는 경우이다. 이 경우에는 상위타입의 자취 $t_1 \cdot t_a$의 수행 여부에 관계 없이 하위타입에서 자취 $t_2 \cdot t_b$가 수행되는 것을 보장하기 위해서 post(t_2) → pre(t_b) 조건이 추가되었다.
- 2.2) 부분은 자취 $t_1 \cdot t_a$가 객체의 상태에 독립적인 경우이다. 즉 상위타입에서 자취 $t_1 \cdot t_a$가 객체의 상태에 독립적으로 수행되는 경우에 하위타입에서 이에 대응되는 자취 $t_2 \cdot t_b$가 post(t_2) → pre(t_b)를 만족시키는 조건이다. 2.1)에서 볼 수 있듯이 $t_1 \cdot t_a$가 상태 독립적인 경우 즉 post(t_1) → pre(t_a)인 경우에는 추가적인 조건이 불필요하며 그 자체만으로 post(t_2) → pre(t_b)가 보장된다. 이를 증명하면 다음과 같다.

· $t_2 \Rightarrow^m t_1$ 이므로 post(t_2) → post(t_1) (1)

· $t_b \Rightarrow^m t_a$ 이므로 pre(t_a) → pre(t_b) (2)

· 정의 10 2.2)에 의해서 post(t_1) → pre(t_a) (3)

(1)과 (3)로부터 post(t_2) → pre(t_a)이다. 그리고 이 조건과 (2)를 조합하면 post(t_2) → pre(t_b)를 얻을 수 있다. 즉 t_2의 수행은 항상 t_b의 수행을 보장한다.

3.3 호출 일관성의 검증

앞에서 제시된 연속 메소드 규칙을 바탕으로 두 객체 타입간의 행위적 호환성 즉 호출 일관성의 정의는 다음과 같다.

정의 11. 두 객체 타입간의 대응관계

객체 타입 ot_1의 모든 전이에 대하여 연속 메소드 규칙을 만족시키는 전이가 객체 타입 ot_2에 존재한다면 ot_2는 ot_1에 대응된다고 정의하며 $ot_2 \Rightarrow^o ot_1$으로 표현한다. 즉

$ot_2 \Rightarrow^o ot_1 \equiv \forall t_1 \in \sigma_{ot1}, \exists t_2 \in \sigma_{ot2}\ [t_2 \Rightarrow^s t_1]$

본 논문에서는 정의 11이 두 타입간의 행위적 호환성 즉 호출 일관성을 점검하는 충분한 조건임을 주장한다. 상위타입의 각 전이에 대해서 확장 메소드 규칙 측면에서 대응되는 전이가 하위타입에 존재한다면 두 타입은 행위적 호환성이 있다고 주장한다. 이 주장을 명제로 표현하면 다음과 같다.

명제 1. $ot_2 \Rightarrow^o ot_1$이면 ot_1의 모든 자취에 대해서 대응되는 자취가 반드시 ot_2에 존재한다.

기본적으로 증명은 자취의 길이에 대한 수학적 귀납적을 이용한다.

Base case: 길이가 1인 자취에 대해서 명제 1이 만족됨을 증명한다. 정의 11에 의해서 $t_2 \Rightarrow^s t_1$인 전이 t_2가 존재한다. 그리고 정의 9로부터 pre(t_1) → pre(t_2)가 만족된다. 따라서 전이 t_1이 ot_1에서 발생한다면 대응되는 전이 t_2가 ot_2에서 수행될 수 있다.

Induction case: ot_1에 길이 n인 자취 tr_1에 대해서 ot_2에 대응되는 자취 tr_2가 있다고 가정한다. t_1과 t_2가 각각 tr_1과 tr_2의 마지막 전이라고 가정한다. 그리고 tr_1에 전이 t_a를 추가된 새로운 자취 tr_1'에 대해서 대응되는 자취 tr_2' (= $tr_2 \cdot t_b$)가 존재함을 증명한다. 이는 자취 $t_1 \cdot t_a$에 대응되는 자취 $t_2 \cdot t_b \in T_{ot2}$가 존재함을 증명

하는 것으로서 자취 $t_1 \cdot t_a$가 상태 독립적인 경우와 상태 의존적인 경우로 나누어서 살펴 본다.

• $t_1 \cdot t_a$가 상태 독립적 자취인 경우:

정의 11에 의해서 $t_2 \Rightarrow^s t_1$ 와 $t_b \Rightarrow^s t_a$이다. $t_1 \cdot t_a$가 상태 독립적 자취이므로 post(t_1) → pre(t_a)이다. 또한 정의 9으로부터 (pre(t_1) → pre(t_2)) ∧ (post(t_2) → post(t_1))을 알 수 있다. 이들로부터 post(t_2) → pre(tb)가 만족된다. 따라서 자취 $t_2 \cdot t_b \in T_{ot2}$가 존재한다.

• $t_1 \cdot t_a$가 상태 의존적 자취인 경우:

정의 11에 의해서 $t_2 \Rightarrow^s t_1$와 $t_b \Rightarrow^s t_a$이다. 그리고 $t_1 \cdot t_a$가 상태 의존적 자취이므로 (pre(t_a) → post(t_1)) ∧ (post(t_2) → pre(t_b))이다. 따라서 자취 $t_2 \cdot t_b \in T_{ot2}$가 존재한다.

위로부터 자취 $t_1 \cdot t_a$에 대응되는 자취 $t_2 \cdot t_b \in T_{ot2}$가 항상 존재함을 알 수 있다. 따라서 tr_2에 대응되는 자취 tr_2'이 존재하게 된다.

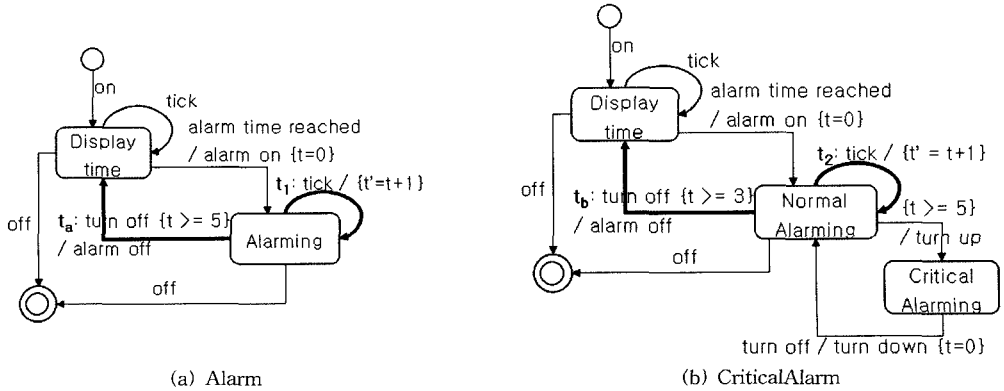
요약하면 상위타입의 각 전이에 대해서 연속 메소드 규칙을 준수하는 전이가 하위타입에 존재하는 경우 상위타입에서 수행 가능한 모든 자취에 대응되는 자취가 하위타입에서도 가능하다.

그림 3은 행위적 호환성의 점검 방법을 소개하기 위한 예제로서 Alarm 객체와 CriticalAlram객체의 행위를 보여준다. Alarm은 on 이벤트를 수신한 후에 tick 이벤트 마다 시간을 표시한다. 설정된 경보시간이 이르면 (alarm time reached), 경보를 작동하기 시작한다 (alarm on). 경보가 작동 후 5초가 경과하면 turn off 이벤트를 통해서 경보 작동을 중단시킬 수가 있다(t_a 전이). CriticalAlarm은 Alarm을 바탕으로 확장된 행위를 가지고 있다. CriticalAlarm은 Alarm과 달리 경보가 작동된 후 3초가 경과하면 turn off 이벤트를 통해서 경보 작동을 중단시킬 수가 있다(t_b 전이). 또한 경보 작동 후 5초가 경과한 후에는 경보의 음량이 높아지고(turn up) CriticalAlarm 상태로 전이된다. 이 상태에서 turn off 이벤트가 Alarm에서처럼 경보를 중단시키는 것이 아니라 기존 음량으로 경보를 울린다(turn down). 또한 높은 음량으로 경보가 울리는 CriticalAlarm 상태에서는 off 이벤트를 통해서 종료 상태로 전이하는 행위도 제공되지 않는다.

일견 CriticalAlarm은 Alarm을 확장시킨 형태로서 행위적 호환성 즉 호출 일관성을 만족시키는 것으로 보일 수가 있다. 그러나, Alarm 객체에서 가능한 자취 중에서 일부는 CriticalAlarm에서 허용되지 않는 것이 있다. 예를 들어, Alarm 객체에서는 경보가 울린 후 5초가 경과하면 turn off 이벤트를 통해서 경보를 중단시킬 수가 있다. 즉 on · alarm time reached · tick · tick · tick · tick · tick · turn off · off 자취가 가능하다. 그러나, 이 자취는 CriticalAlarm에서는 발생할 수가 없다. CriticalAlarm에서는 경보 작동 후 5초가 경과하면 CriticalAlarming 상태가 된다. CriticalAlarm에서 turn off 이벤트가 수신되어야 NormalAlarm 상태로 돌아갈 수 있다. 또한 이때 t가 0으로 초기화되었기 때문에 turn off 이벤트로 경보를 중단시키기 위해서는 3초를 더 기다려야만 한다. 즉 CriticalAlarm 객체는 Alarm 객체와 행위적으로 호환되지 않는다.

Alarm 유한상태기계의 각 전이는 CriticalAlarm에서 메소드 규칙이 준수되고 있다. Alarm의 전이 중에서 turn off 이벤트의 전이를 제외한 나머지 전이는 CriticalAlarm과 동일하다. t_a 전이의 경우에는 선행 조건이 {t>=5}에서 t_b 전이에서 {t>=3}으로 약화되었으므로 기존의 메소드 규칙을 준수하고 있다. 따라서 두 객체간의 행위적 비호환성은 기존의 메소드 규칙만을 통해서는 확인할 수가 없다.

반면에 본 논문에서 제시한 연속 메소드 규칙을 통해서 두 객체간의 행위적 비호환성을 확인하는 것이 가능



(a) Alarm (b) CriticalAlarm

그림 3 Alarm과 CriticalAlarm의 행위 모델

하다. Alarm 객체에서 tick · turn off 자취는 상태 의존적으로 수행될 수 있다. 즉 tick 이벤트 후에 t 값이 5 이상인 경우에만 turn off 이벤트에 의해서 경보가 중단될 수가 있다. 기존 메소드 규칙에 따르면 Alarming 상태에서의 t_1 전이와 t_a 전이는 NormalAlarming 상태에서의 t_2 전이와 t_b 전이에 대응하는 것으로 간주될 수가 있다. 그러나 연속 메소드 규칙에 따르면 두 상태에서의 t_1 전이와 t_2 전이는 대응되지 못한다. 즉 연속 메소드 규칙에 따르면 Alarm 객체에서 t_1 · t_a은 상태 의존적이므로 CriticalAlarm 객체에서 t_2 전이의 후행조건이 t_b 전이의 선행조건을 만족시켜야 만이 t_1 전이와 t_2 전이가 대응된다고 말할 수 있다. 그러나 CriticalAlarm에서 t_2 전이의 후행 조건은 t_b 전이의 선행조건을 충족시키지 않고 있다. 그러므로, Alarm 객체의 t_1전이는 CriticalAlarm에서 대응되는 전이가 없으며 CriticalAlarm 객체는 Alarm 객체와 행위적으로 호환되지 않다고 판단할 수가 있다.

3.4 검증 알고리즘

그림 4는 제시된 연속 메소드 규칙을 이용하여 두 객체의 행위적 호환성을 점검하는 알고리즘을 보여준다.

그림 4의 알고리즘은 2개의 단계로 구성된다. 첫 번째 Finding Corresponding States and Transition 단계는 두 유한상태기계에서 전이의 조건이 없다고 가정하였을 때 서로 대응되는 상태와 전이를 찾아서 대응 상태 집합 CS와 대응 전이 집합 CT에 저장한다. 두 번째 Checking Methods Rule단계는 첫 번째 단계에서 찾은 대응 전이 집합의 조건을 고려하여 각 전이 쌍들이 연속 메소드 규칙을 만족 하는지 검사한다.

- check_behavior_correspondence 함수는 기존의 유한상태기계 SM^1의 상태와 대응하는 상태, 기존의 유한상태기계의 전이와 대응하는 전이가 대체하는 유한상태기계 SM^2에 모두 나타나는지 검사한다. 이 함수는 2단계로 구성된다. 첫 단계는 각 전이의 조건을 고려하지 않고 단지 상태와 전이가 대응하는지를 검사하여 대응 상태/전이 쌍을 찾는다. 기존의 유한상태기계의 상태에 대응하는 상태가 대체하는 유한상태기계에 모두 나타난다면 정의 7,8에 따라서 전이도 모두 대응된다. 그러므로 이 함수에서는 상태의 대응을 모두 검사함으로써 유한상태기계의 대응을 검사한다. 두번째 단계는 첫 단계에서 찾은 전이 쌍이 연속 메소드 규칙을 만족하는 지 검사한다.
- check_state_correspondence 함수 두 상태가 서로 대응하는지 검사한다. 만약 상태가 모두 초기 상태이거나 이미 대응 된다고 판단되었다면 참을 반환한다. 그렇지 않다면 정의 8에 따라서 기존의 유한상태기계의 상태 s_1의 진입 전이가 대체되는 유한상태기계의 상태 s_2진입 전이에 나타나는 지 검사한다. in_trans(s_1) 중 어느 하나라도 in_trans(s_2)와 대응하는 것이 없다면 두 상태는 대응 하지 않는다. s_1과 s_2가 정의 8을 만족한다면 대응 상태 집합CS에 <s_1,s_2>를 추가한다.
- check_transition_correspondence함수는 두 전이가 서로 대응하는지 검사한다. 정의 7의 두 번째 조건에 따라서 두 전이의 입/출력 이벤트 집합의 매핑이 존재하지 않는다면 두 전이는 대응하지 않는다. 그리고 두 전이가 이미 대응된다고 판단되었거나 현재 다른 상태의 대응을 검사하기 위한 과정 중에 이미 들른 전이라면 참을 반환한다. 같은 전이를 두 번 검사한다는 것은 현재의 유한상태기계가 사이클을 가진다는 것을 뜻한다. 이때는 기존의 유한상태기계의 임의의 상태의 진입 전이들이 대체되는 유한상태기계의 진입 전이들과 대응하거나 현재 검사 중에 있는 전이들로 이루어져 있을 때는 대응하는 상태이다.

  정의 7의 첫 번째 조건에 따라서 source(t_2) ⇒ source(t_1)를 만족하면 두 전이는 대응하므로 <t_1,t_2>를 대응 전이 집합에 추가한다.
- 두 번째 단계인 check_successive_methods_rule함수는 대응하는 모든 전이가 연속 메소드 규칙을 준수하는지 검사한다. 정의 10의 첫 번째 조건에 따라서 전이 t_1과 t_2가 메소드 규칙을 준수하지 않으면 연속 메소드 규칙을 준수 하지 않는다. 그리고 정의 10의 2.1 조건에 따라서 상태 의존적 수행 조건일 경우 즉, pre(t_a) → post(t_1)를 만족할 경우 post(t_2) → pre(t_b)를 만족하면 t_1은 연속 메소드 규칙을 준수한다. 그리고 정의 10의 2.2조건에 따라서 상태 독립적 수행 조건일 경우 즉, post(t_1) → pre(t_a)을 만족하면 t_1은 연속 메소드 규칙을 준수한다.

그림 3에서 Display time상태를 예로 알고리즘을 설명하고자 한다. 조건을 고려하지 않은 상황에서 그림 3의 *Alarm*과 *CriticalAlarm*의 *Display time*상태가 서로 대응하기 위해서는 <on^1,on^2>[1], <$tick^1$,$tick^2$>, <*ta*,*tb*>가 모두 대응하면 된다. 여기서 <on^1,on^2>은 source 상태가 초기 상태이므로 정의 7에 따라서 서로 대응 하는 전이다. <$tick^1$,$tick^2$>가 서로 대응하기 위해서는 <$tick^1$, $tick^2$>의 source상태 쌍인 <*Display time*^{1}, *Display time*^{2}>이 대응해야 한다. 이때 <*Display time*^{1}, *Display time*^{2}>의 진입 전이 중 <on^1,on^2>은 대응하는 전이 쌍이고, <$tick^1$,$tick^2$>는 현재 고려 중인 전이 쌍이므로 <*ta(turn off/ alarm off)*,*tb(turn off/alarm off)*>가 대응하면 된다. <*ta*,*tb*>가 대응하기 위해서는 <ta,tb>의

1) 이하 전이와 상태의 ^{1}은 *Alarm*의 요소를 뜻하고 ^{2}는 *CriticalAlarm*의 요소를 뜻한다.

```
Let SM^1 = (S^1, s^1_0, S^1ψ, I^1, O^1, σ^1 )
Let SM^2 = (S^2, s^2_0, S^2ψ, I^2, O^2, σ^2)
CS is a set of corresponding states
CT is a set of corresponding transitions
PCT is a set of partially corresponding transitions

boolean check_behavior_correspondence(SM^1, SM^2) begin

//Phase 1 Finding Corresponding States and Transitions

        let S^1 ∈ SM^1
        let S^2 ∈ SM^2
        for each s_1 ∈ S^1 begin
                let is_corespondence be false
                for each s_2 ∈ S^2 begin
                        PCT = 0
                        if check_state_correspondence(s_1, s_2) is true begin
                                set is_corespondence true
                                break
                        end if
                end for
                if is_corespondence is false then return false
        end for

//Phase 2 Checking Successive Methods Rule

        for each <t_1,t_2> ∈ PCT begin
                if check_successive_methods_rule( t_1, t_2 ) is false return false
        end for
        return true
end


boolean check_state_correspondence (s_1∈S^1, s_2∈S^2) begin
        if  (s_1 = S^1_0 ∧ s_2 = S^2_0) ∨ <s_1, s_2> ∈ CS then return true
        T_1 ⊆ in_trans(s_1)
        T_2 ⊆ in_trans(s_2)
        for each t_1 ∈ T_1 begin
                find_correspondence = false
                for each t_2 ∈ T_2 such that there is a mapping t_1 to t_2 begin
                        if check_transition_correspondence(t_1, t_2) is true begin
                                find_correspondence = true
                                break
                        end if
                end for
                if not find_correspondence then return false
        end for
        CS = CS ∪ {<s_1,s_2>}
        return true
end

boolean check_transition_correspondence(t_1∈σ^1, t_2∈σ^2) begin
        if <t_1,t_2> ∈ CT ∨ <t_1,t_2> ∈ PCT then return true
        PCT = PCT ∪ {<t_1,t_2>}
        if check_state_correspondence( source(t_1), source(t_2) ) is false then return false
        CT = CT ∪ {<t_1,t_2>}
        return true
end

boolean check_successive_methods_rule(t_1∈σ^1, t_2∈σ^2) begin
        if not ( pre(t_1) → pre(t_2) ) ∧ ( post(t_2) → post(t_1) ) then return false
        for each t_a ∈ out_trans(sink(t_1)) begin
                let t_b such that <t_a, t_b> ∈ CT
                if pre(t_a) → post(t_1) begin //state dependent
                        if not post(t_2) → pre(t_b) return false
                else if not post(t_1) → pre(t_a) then return false //state not independent
                end if
        end for
        return true
end
```

그림 4 행위적 호환성 점검 알고리즘

source 상태인 <*Alarming, Normal Alarming*>이 대응해야 한다. <*Alarming, Normal Alarming*>이 대응하기 위해서는 Alarming의 진입 전이인 *t1(tick)*, *alarm time reached*와 대응하는 전이가 *CriticalAlarm*에 존재해야 한다. *Normal Alarming*의 진입 전이 중, *Alarming*의 진입전이 *t1*과 입/출력 이벤트가 매칭하는 전이는 *t2(tick)*이다. 그리고 *Alarm*의 *alarm time reached*전이와 입/출력 이벤트가 매칭하는 전이는 *CriticalAlarm*의 *alarm time reached* 전이다. 그러므로 <*t1,t2*>, <*alarm time reached, alarm time reached*>가 서로 대응하면 <*Alarming, Normal Alarming*>은 서로 대응한다. <*t1,t2*>가 대응하기 위해서는 각 전이의 source 상태 쌍인 <*Alarming, Normal Alarming*>가 서로 대응해야 한다. <*Alarming, Normal Alarming*>의 진입 전이 쌍 중 <*t1,t2*>는 현재 고려중인 전이 쌍이므로 <*alarm time reached, alarm time reached*>가 대응해야 한다. <*alarm time reached, alarm time reached*>가 대응하기 위해서는 각각의 source 상태 쌍인 <*Display time*^{1}, *Display time*^{2}>가 대응하면 된다. <*Display time*^{1}, *Display time*^{2}>의 진입 전이 쌍 중 <*on*^{1}, *on*^{2}>은 서로 대응하고 <*tick*^{1},*tick*^{2}>와 <*ta,tb*>는 고려 중인 전이이므로 <*Display time*^{1}, *Display time*^{2}>는 서로 대응한다. 그러므로 <*alarm time reached, alarm time reached*>, <*Alarming, Normal Alarming*>, <*t1, t2*>, <*ta, tb*>, <*tick*^{1},*tick*^{2}>은 순차적으로 대응한다.

Checking Methods Rule단계에서 모든 대응 전이 쌍에 대해서 연속 메소드 규칙을 만족하는지 검사한다. 이때 <*t1,t2*>는 메소드 규칙을 만족한다. 그리고 t1, tb가 pre(t_a) → post(t_1)의 관계를 만족하므로 연속 메소드 규칙을 만족하기 위해서는 post(t_2) → pre(t_b)를 만족해야 한다. 하지만 그림 3의 *CriticalAlarm*에서는 post(t_2) → pre(t_b) 조건을 만족하지 못한다. 그러므로 *CriticalAlarm*은 *Alarm*과 호환성이 없다.

4. 관련 연구

객체 타입간의 행위적 호환성을 점검하기 위한 많은 연구가 있었다. 기존에 프로그래밍 언어 분야에서 제시된 contra/covariance 기법을 메소드의 인자와 반환 타입에 적용하는 연구가 대표적으로 이에 속한다[1,18]. 즉 인자의 타입은 일반화되고 반환 타입은 구체화하는 방향으로 메소드를 재정의해야 한다고 주장하였다. Meyer는 contra/covariance 기법을 메소드의 선행/후행 조건에 적용하였다[5]. 즉 상위메소드의 선행조건을 약화시키고 반대로 후행조건은 강화시키는 방식으로 하위메소드를 재 정의하면 하위클래스의 객체가 상위클래스의 객체 대신에 사용될 때 안전할 수 있다고 주장하였다. Loskov와 Wing[4]은 위에서 언급한 메소드 시그너처에 대한 contra/covariance 규칙과 Meyer의 선행/후행조건을 포함하여 불변식 규칙을 추가한 형태로서 두 타입간의 서브타입 관계를 정의하였다.

Ebert와 Engels[3] 그리고 Saake 등[19]은 상태 다이어그램을 바탕으로 행위적 서브타이핑을 연구하였다. 그들은 기존에 이미 수행한 연구[20,21]를 바탕으로 그래프 동형 기법을 이용하여 상위타입과 하위타입간의 매핑을 정의하려고 하였다. 그들은 조건이 있는 전이를 고려하였지만 호출 일관성이 아니라 관찰 일관성을 다루고 있다. 호출 일관성과 달리 관찰 일관성은 하위타입의 자취에 해당하는 자취가 상위타입에 있는 것을 뜻하며 상위타입의 객체 대신에 하위타입의 객체가 대신 사용될 수 있는 행위적 호환성 측면을 점검하기 위한 용도로는 부족한 규칙이다. 또한 그들은 두 상태 다이어그램간에 대응되는 개별적인 전이만을 비교하였으며 본 논문과 같이 자취의 존재여부를 다루고 있지는 않다.

Schrefl과 Stumptner[22-24]는 페트리 넷과 객체 행위 다이어그램(object behavior diagram)을 정의하고 이를 바탕으로 관찰 일관성과 호출 일관성을 점검하는 방법을 제시하였다. 또한 그들은 호출 일관성을 약 호출 일관성과 강 호출 일관성으로 구분하고 있으며 약 호출 일관성이 본 논문의 호출 일관성에 해당된다. 그들은 객체 생명 주기(object life cycle) 개념을 바탕으로 일관성 규칙을 정의하였으며 객체 생명 주기는 본 논문에서 바탕으로 한 자취와 유사하다. 그들의 연구는 일관성 규칙을 명확하게 정의하고는 있지만 본 논문에서 다루는 유한상태기계(UML의 상태다이어그램과 유사)과 같이 범용적인 표현법은 아니므로 제시된 기법을 실용적으로 적용하거나 이를 도구화하는데 보다 큰 어려움이 있을 수 있다. 또한 그들의 연구에서는 조건을 가진 전이를 고려한 일관성 규칙을 제시하고 있지는 않다.

Fisher와 Wehrheim[25,26]은 CSP [27]를 바탕으로 분산 시스템 하에서 4가지 유형의 서브타이핑 관계를 제안하였다. 최근에 Wehrheim[28]은 상태와 행위를 통합한 관점에서 서브타이핑을 제시하려고 하였다. 이 연구를 바탕으로 Olderog와 Wehrhem[29]은 CSP와 Object-Z의 혼합 언어인 CSP-OZ 언어[13,30]에서 상속의 개념을 연구하였다.

5. 결론 및 향후 연구 방향

본 논문에서는 두 객체 타입간의 행위적 호환성을 판단하기 위한 규칙으로서 호출 일관성을 점검하는 방법을 제안하였다. 객체의 동적 행위를 기술하는 데 일반적으로 널리 사용되는 유한상태기계를 바탕으로 기존의

전통적인 메소드 규칙을 보완한 연속 메소드 규칙을 제시하였다. 그리고 연속 메소드 규칙을 준수하는 전이가 존재한다면 호출 일관성 즉 행위적 호환성이 있음을 주장하였다. 그리고 본 논문에서 제시한 연속 메소드 규칙을 이용하여 객체간의 호환성을 검증하기 위한 알고리즘을 제시하였다.

본 논문에서는 제시된 행위적 호환성을 점검하기 위한 규칙을 명확히 소개는 하였지만 이를 실제 적용해 보기 위해서는 우선 적절한 도구의 지원이 필요하다. 우선 상태기계로 객체의 행위를 표현하는 것은 Eclipse의 프로젝트 중 하나인 Model Development Tools(MDT) 프로젝트[31]에서 Eclipse의 플러그인 형태로 제공하는 StateMachine Diagram뿐만 아니라 기존의 UML 모델링 도구를 충분히 활용할 수 있다. 그리고 전이의 선행/후행 조건의 표현은 OCL(Object Constraint Language)의 사용을 고려하고 있다. 또한 선행/후행 조건의 포함관계를 알 수 있는 방법도 연구 되어야 한다. 앞으로는 본 논문에서 제시된 행위적 호환성 점검 방법을 MDT에서 제공하는 StateMachine Diagram을 추가/확장하여 본 논문에서 소개한 연속 메소드 규칙 검증 알고리즘을 적용하여 구현 할 계획이다.

참 고 문 헌

[1] Wegner, P. and S. Zdonik. 1988. Inheritance as an incremental modification mechanism or what like is and isn't like. In *ECOOP '88*, pp. 55-77.

[2] America, P. 1991. Designing an Object-Oriented Programming Language with Behavioral Subtyping, In *Proceedings of the REX School/Workshop on Foundations of Object-Oriented Languages*. London, UK: Springer-Verlag, pp. 60-90.

[3] Ebert, J. and G Engels. 1994. Observable or invocable behavior. You have to choose. Technical report Universitat Koblenz, Koblenz, Germany.

[4] Liskov, B. and J. Wing. 1994. A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems* 15(6), pp. 1911-1841.

[5] Meyer, B. 1992. \Design by Contract," *IEEE Computer* 25(10), p. 4051.

[6] Chow, T., S. 1978. Testing software design modeled by finite-state machines. *IEEE Trans. Software Eng* 4(3), pp. 178-187.

[7] Knor, R. Trausmuth, G. and Weidl, J. 1998. Reengineering c/c++ source code by transforming state machines. In *ESPRIT ARES Workshop*, pp. 97-105.

[8] Corbett, J., C. Dwyer, M., B. Hatcliff, J. Laubach, S. Pasareanu, C., S. Robby, and Zheng, H. 2000. Bandera:extracting finite-state models from java source code. In *ICSE*, pp. 439-448.

[9] N. Pywes and P. Rehmet. 1996. Recovery of software design, state-machines, and speci-cations from source code. *In ICECCS*, pp. 279-288.

[10] Chevalley, P. and Thevenod-Fosse, P. 2001. Automated generation of statistical test cases from uml state diagrams. In *COMPSAC*, pp. 205-214.

[11] Kim, Y. Hong, H. Bae, D. and Cha. S. Test cases generation from uml state diagrams.

[12] David, A. Moller, M., O. and Yi, W. 2002. Formal verification of uml statecharts with real-time extensions. In *FASE*, pp. 218-232.

[13] van Katwijk, J. Toetenel, H. Sahraoui, A.-E.-K. Anderson, E. and Zalewski, J. 2000. Specification and verification of a safety shell with statecharts and extended timed graphs. In *SAFECOMP*, pp. 37-52.

[14] Kang, K., C. and Ko, K.-I. 1996. Formalization and veri-cation of safety properties of statechart specifications. In *APSEC*, pp. 16-,

[15] Pinter, G. and Majzik, I. 2004. Runtime verification of statechart implementations. In *WADS*, pp, 148-172.

[16] Khler, H., J. Nickel, U. Niere, J. and Zndorf, A. 2000. Integrating uml diagrams for production control systems. In *ICSE '00*: Proceedings of the 22nd international conference on Software engineering, pp. 241-251.

[17] Niaz, I. and Tanaka, J. 2003. Code generation from uml statecharts. In *SEA '03: Proceedings of international conference on Software Engineering and Applications.*

[18] Canning, P., W. Cook, W. Hill and W. Olthoff. 1989. Interfaces for strongly-typed object-oriented programming. In *OOPSLA '89*, pp. 457-467.

[19] Saake, G., R. Jungclaus, R. Wieringa and R. Feenstra. 1994. Inheritance Conditions for Object Life Cycle Diagrams. In *Proceedings of EMISA*. pp. 79-88.

[20] Ehrich, H.-D., J. Goguen and A. Sernadas. 1990. A Categorical Theory of Objects as Observed Processes. In *Proceedings of Foundations of Object-Oriented Languages(REX School/Workshop)*. pp. 203-228.

[21] Sernadas, A. and H.-D. Ehrich. 1991. What is an Object, after all? In *Proceedings of IFIP WG 2.6 Working Conference on Object-oriented Databases: Analysis, Design and Construction*. pp. 39-70.

[22] Schrefl, M. and M. Stumptner. 1995. Behavior Consistent Extension of Object Life Cycles. In *Proceedings of OOER'95*. pp. 133-145.

[23] Schrefl, M. and M. Stumptner. 1997. Behavior Consistent Refinement of Object Life Cycles. In *Proceedings of ER'97*. pp. 155-168.

[24] Schrefl, M. and M. Stumptner. 2002. Behavior-consistent Specialization of Object Life Cycles,"

ACM Transactions on Software Engineering and Methodology 11(1), pp. 92-148.
[25] Fischer, C. and H. Wehrheim. 2000. Behavioral Subtyping Relations for Object-Oriented Formalisms," *Lecture Notes in Computer Science* 1816, pp. 469-484.
[26] Wehrheim, H. 2003. Behavioral Subtyping Relations for Active Objects," *Form. Methods Syst. Des.* 23(2), pp. 143-170.
[27] Hoare, C. 1985. *Communicating Sequential Process.* Prentice Hall.
[28] Wehrheim, H. 2002. Behavioral Subtyping in Object-oriented Specification Formalisms."
[29] Olderog, E.-R. and H. Wehrheim. 2005. Specification and (property) inheritance in CSP-OZ," *Sci. Comput. Program.* 55(1-3), pp. 227-257.
[30] Fischer, C. 1997. CSP-OZ: a combination of Object-Z and CSP. In *Proc. 2nd IFIP Workshop on Formal Methods for Open Object-Based Distributed Systems (FMOODS),*
[31] Model Development Tools (MDT) version 1.0, 2007. Available at http://www.eclipse.org/modeling/mdt/?project=uml2

채 홍 석
정보과학회논문지 : 소프트웨어 및 응용
제 34 권 제 7 호 참조

이 준 상
1997년 동국대학교 컴퓨터공학과(공학사). 1999년 한국과학기술원 전산학과 전공(공학석사). 2003년 한국과학기술원 전산학과 전공(공학박사). 2003년 2월~2006년 6월 LG전자 디지털 미디어 연구소, HPS 그룹 선임 연구원. 2006년 7월~2007년 4월 LG전자 소프트웨어 & 솔루션 센터, 책임연구원. 2007년 5월~현재 고려대학교 대학원 임베디드 소프트웨어학과, 연구교수. 관심분야는 소프트웨어공학, 임베디드 소프트웨어, 소프트웨어 아키텍처 등

배 정 호
2007년 부산대학교 컴퓨터공학과(공학사). 2007년 3월~현재 부산대학교 대학원 석사과정. 관심분야는 소프트웨어공학, 소프트웨어 아키텍처, 객체지향 개발방법론 등