인터페이스 명세에서 효과적인 프레임 방법

천윤식, 김흥남
한국전자통신연구원 컴퓨터·소프트웨어기술연구소 소프트웨어공학연구부
e-mail : cheon@etri.re.kr

A Practical Approach to Framing In Interface Specifications

Yoonsik Cheon and Heung-Nam Kim
Software Engineering Department
ETRI Computer & Software Technology Laboratories

요 약

인터페이스 명세에서 프레임문제(frame problem)란 "특정부분 이외의 모든 프로그램 상태는 변화되지 않는다"는 것을 기술하는 문제이다. 본 논문에서는 프레임문제를 상속을 지원하는 객체지향명세에서 재조명하고 이에 대한 현실적인 접근방법을 제시한다. 먼저 프레임문제를 정형적으로 다루기 위하여 프레임관계(frame relation)라는 개념을 도입한다. 프레임관계는 연산자와 변수간의 관계로 특정 연산자가 어떤 변수를 수정할 수 있는가를 나타낸다. 슈퍼클래스의 프레임관계는 앵커관계(anchoring relation)를 통하여 서브클래스로 확장되는데, 앵커관계는 서브클래스 변수에서 슈퍼클래스 변수로의 사상이다. 앵커관계는 프레임문제 해결의 핵심적인 역할을 한다. 즉, 연산자가 슈퍼클래스의 특정 변수를 수정할 수 있다는 것은 그 변수에 앵커 된 모든 서브클래스 변수를 수정할 수 있다는 것을 의미한다. 앵커관계는 프레임문제의 현실적이고 효과적인 접근방법으로, 널리 보급된 선·후조건문 방식의 인터페이스명세에 잘 접목된다는 부가적인 장점을 가진다.

1. 개요

인터페이스 명세에서 프레임문제(frame problem)는 특정 연산자에 의해 어떤 변수가 수정될 수 있고 어떤 변수가 수정될 수 없는가에 대한 명세로 귀결된다. 특정부분 이외의 모든 프로그램 상태는 변화되지 않는다는 것을 직접 명세하는 것은 인터페이스 명세를 길게 만들뿐만 아니라, 결과적으로 명세의 이해와 수정을 어렵게 한다. 또한 수정 가능한 변수를 일일이 나열하여야 하므로 부주의로 인한 오류 가능성을 높이고 명세의 상속과 객체지향 방식에 적용하기 어렵다. 따라서 대부분의 인터페이스 명세언어는 문맥에 따라 달리 해석되는 구문적 약어를 제공한다. 예를 들면, **modifies** 절이 그것이다 [9]. 이는 해당 연산자가 수정할 수 있는 모든 변수를 나열하는데, 그 밖의 모든 변수는 수정될 수 없음을 간접적 명세 한다. 한 예로서 변수 x, y 를 접근할 수 있는 연산자를 가정하자. 만약 연산자 명세에 "**modifies** x" 절이 포함되었다면, 이 연산자는 단지 변수 x 만을 수정할 수 있다. 특히 또 다른 변수 z 가 존재하는 문맥에서 이 연산자가 호출되었다 할지라도, 단지 변수 x 만 수정될 수 있다. 즉, 호출의 결과로 변수 y 와 z 의 값은 변화되어서는 안 된다. 예에서와 같이 **modifies** 절은 특정 연산자가 볼 수 있는 프로그램 상태에 테두리를 치는 것과 같다. 테두리 안의 상태는 해당 연산자에 의해 접근되고 수정될 수 있으나 테두리 밖의 상태는 동결, 즉 변화될 수 없다. 프레임문제라 불리는 이유도 이 때문이다.

본 논문에서는 객체지향 인터페이스 명세에서 프레임문제에 대해서 자세히 설명하고 이에 대한 효과적인 해결방안을 제시한다. 프레임문제를 정형적으로 다루기 위하여 먼저 '프레임관계(frame relation)'라는 개념을 도입한다. 프레임관계는 클래스에 정의된 연산자와 개체변수 간의 다 대 다 관계로 특정 연산자가 어떤 변수를 수정할 수 있는가를 수학적으로 묘사한다. 프레임관계를 서브클래스를 통한 상속에 적용하기 위하여, '앵커관계(anchoring relation)'라는 개념을 도입한다. 앵커관계는 서브클래스의 개체변수에서 슈퍼클래스의 개체변수로의 사상이다. 앵커관계는 상속을 지원

하는 객체지향 명세에서 프레임문제 해결의 핵심적인 역할을 한다. 즉, 임의의 연산자가 슈퍼클래스의 한 변수에 대한 수정권리를 가지면 그 변수에 앵커 된 모든 서브클래스 변수에 대한 수정권리가 주어진다. 이는 변수 수정권리가 앵커관계를 따라 서브클래스로 전파됨을 의미한다. 앵커관계를 사용한 접근방법은 객체지향 명세의 프레임문제를 매우 효과적으로 해결할 뿐만 아니라 인터페이스 명세에 널리 사용되는 선.후조건문 방식에 잘 적용된다는 현실적인 장점이 있다.

본 논문은 다음과 같이 구성된다. 먼저 제 2 절에서는 프레임문제를 객체지향과 상속이라는 측면에서 재조명하고, 상속이 허용될 때 프레임문제는 더욱 복잡해짐을 간단한 예를 사용하여 설명한다. 제 3 절에서는 프레임문제를 수학적으로 접근하기 위하여 변수에 대한 연산자의 수정권리를 프레임 관계로 모형화 한다. 본 논문에서 가장 중요한 부분인 제 4 절에서는 슈퍼클래스와 서브클래스 간의 자료정제를 앵커관계로 모형화하고 이를 사용하여 직관적이고 효과적인 프레임문제의 해결방법을 제시한다. 또한 앵커관계의 수학적인 특성도 분석한다. 다음으로 제 5 절에서 주요 관련연구를 살펴보고 마지막으로 제 6 절에서 결론을 맺는다.

2. 프레임 문제

프레임문제를 설명하기 위하여 간단한 컴퓨터 게임을 예로 사용한다 [10]. 게임의 주체는 Sprite 라 불리는 개체이다. Sprite 는 위치와 색깔을 나타내는 변수를 가지며 이들을 수정하기 위한 연산자를 제공한다. 게임의 주 프로그램은 매 비디오 프레임 마다 각 Sprite 에 대하여 먼저 update 연산자를 호출하여 위치와 색깔정보를 수정하고, 그 다음 draw 연산자를 통하여 해당 Sprite 를 화면에 출력하도록 한다.

클래스 Sprite 는 다음과 같이 정의된다. 편의상 **modifies** 절을 제외한 모든 행위명세, 예를 들면 선·후조건문은 생략되었다. 명세는 //@로 시작하는 특수한 형태의 코멘트로 프로그램에 첨부된다 [4].

```
class Sprite {
   int x, y;
   int color;
   void updatePosition {
      //@ modifies x,y;
   }
   void updateColor() {
      //@ modifies color;
   }
   void update() {
      //@ modifies x,y,color;
      updatePosition();
      updateColor();
   }
   void draw() {
      //@ modifies nothing;
   }
};
```

연산자 updatePosition 과 updateColor 의 디폴트 구현은 아무런 작업도 하지 않는다. 하지만 서브클래스에 의해 정제될 수 있도록 하기 위하여, 각각 개체변수 x, y 와 color 의 수정이 허용되었다. 연산자 update 는 연산자 updatePosition 과 updateColor 를 순차적으로 호출한다. 따라서 변수 x, y, color 를 수정할 수 있도록 명세 되었다.

게임의 영웅을 표시하는 Sprite 의 서브클래스 Hero 를 고려해 보자. Hero 개체는 이리 저리 움직일 수 있다. 따라서 연산자 updatePosition 는 디폴트 구현을 상속 받지 않고 독자 구현을 제공한다. Hero 객체의 다음 위치는 개체변수로 표현되는 속도와 가속도로부터 계산된다.

```
class Hero: public Sprite{
   int dx, ddx;
   int dy, ddy;
   void updatePosition() {
      x += dx + ddx / 2;
      y += dy + ddy / 2;
      dx += ddx;
      dy += ddy;
   }
};
```

클래스 Hero 의 updatePosition 구현은 위치를 나타내는 변수 x 와 y 를 적당한 양 만큼 증가 시킨다. 속도도 현재의 가속도에 따라 증가된다. 위 구현에서 보는 바와 같이 새로운 위치를 계산하는 연산자에서 속도를 수정하는 것은 자연스럽다. 하지만 (슈퍼클래스 Sprite 로부터 상속 받은) 연산자 updatePosition 의 명세는 오직 변수 x, y 만의 수정을 허용한다. 변수 dx 와 dy 는 수정될 수 없을 뿐만 아니라 이들은 클래스 Sprite 에서는 선언되지 조차도 않았다. 따라서 새로운 updatePosition 구현은 슈퍼클래스에서 명세를 수정하지 않는 이상 합법적인 구현이 될 수 없다.[1] 요약하면, 명세상속이 허용되면 슈퍼클래스에 정의된 **modifies** 절과 같은 프레임공리는 명세작성 시 고려되지조차도 않은 서브클래스의 개체변수에 대해서도 적용될 수 있어야 한다 [2]. 그러므로 객체지향명세에서 프레임문제는 더욱 복잡해 진다.

문헌에서 가장 자주 접하게 되는 객체지향 프레임문제의 해결책으로는 "**modifies also**" 절이 있다 [4]. 그 의도는 서브클래스에서 번복(override)되는 연산자의 프레임 공리는 서브클래스에서 추가하자는 것이다.

[1] 사실상 슈퍼클래스에서 수정할 수도 없는 상황이다. 서브클래스에서 추가된 개체변수는 슈퍼클래스에서는 알 수 없기 때문이다.

예를 들어, 연산자 updatePosition 의 경우 Hero 클래스에서 "**modifies also** dx, dy, ddx, ddy;" 절을 추가하면 된다. 이는 Hero 클래스의 updatePosition 구현이 상속받은 변수 x, y 뿐만 아니라 추가된 변수 dx, dy, ddx, ddy 도 수정할 수 있도록 한다. 모든 문제가 해결된 것처럼 보인다. 하지만 사실은 그렇지 않다. 예를 들어, 새로운 게임 레벨이 시작될 때 Hero 객체의 startNewLevel 이라는 연산자가 호출된다고 가정하자. 이 연산자는 새로운 게임 레벨을 시작하기 위하여 일부 특성은 유지하고 나머지는 초기화한다.

```
void startNewLevel() {
   //@ modifies x,y,color,dx,dy,ddx,ddy;
   //@ ensures dx_post = 0 ∧ dy_post = 0;
   dx = 0;
   dy = 0;
   update();
}
```

연산자 startNewLevel 의 구현은 그 명세를 만족하는 것처럼 보인다. 실제로 이를 수학적으로 간단하게 증명할 수 있다. 먼저 dx, dy 의 값이 0 으로 설정되었고 그 다음에 update 가 호출되었다. 연산자 update 는 dx, dy, color 변수만 수정할 수 있으므로 (클래스 Sprite 에 주어진 update 명세 참조) **ensures** 절에 명시된 후조건문은 만족된다.[2] 하지만 불행하게도 실제 코드실행은 dx, dy 를 0 이 아닌 전혀 다른 값으로 설정할 수 있다. 연산자 update 의 호출은 클래스 Sprite 에 정의된 구현을 실행한다. 따라서 연산자 updatePosition 과 updateColor 이 순차적으로 호출되는데, 동적호출(dynamic dispatch)에 의하여 클래스 Hero 에 정의된 updatePosition 이 호출된다. 클래스 Hero 의 구현은 변수 dx, dy 를 수정할 수 있다. 즉, 초기에 설정된 0 값이 유지되지 않을 수 있다. 결국, 제안된 명세 프레임워크에 논리적 모순이 있다는 이야기다. 문제의 원인은 **modifies also** 절이 관련된 두 연산자(예: updatePosition, update)가 서로 다르게 확장되는 것을 허용하는데 있다. 즉, updatePosition 는 추가변수의 수정이 허용되었지만 update 의 경우 updatePosition 을 호출함에도 불구하고 그렇지 않았다. 아래에서는 명세상속의 개념하에서 프레임문제를 해결할 수 있는 새롭고 효과적인 방법을 제시한다.

3. 프레임의 수학적 모형

본 절에서는 먼저 프레임문제를 좀더 정형적으로 다루기 위하여 프레임이 무엇인가를 수학적으로 모형화 한다.

임의의 클래스 T 에 대하여 O 와 V 를 각각 T 에 정의된 연산자와 개체변수(instance variable)의 집합이라고 하자. 클래스 T 의 프레임은 임의의 $o \in O$ 에 대하여 어떤 $v \in V$ 를 수정할 수 있는가를 나타낸다. 수학적인 측면에서 보면, 이는 O 와 V 사이의 관계(relation)를 정의한다. 이 관계 $r: O \leftrightarrow V$ 를 '프레임관계'라 부른다.[3]

정의 1. 연산자와 개체변수의 집합 O 와 V 를 갖는 클래스 T 에 대하여, '프레임관계(frame relation)', $r: O \leftrightarrow V$ 는 다음과 같이 정의된다. 모든 $o \in O$, $v \in V$ 에 대하여, 연산자 o 가 변수 v 를 수정할 수 있을 때만, $(o,v) \in r$ 이다.

인터페이스 명세에서 클래스 T 의 프레임관계는 T 에 정의된 연산자명세의 **modifies** 절에 의해 주어진다. 특정 연산자의 **modifies** 절은 해당 연산자를 위한 프레임관계의 원소를 정의하고, 모든 **modifies** 절은 프레임관계를 완전하게 정의한다. 예를 들면, 클래스 Sprite 의 프레임관계는 다음과 같다.

{(updatePosition,x), (updatePosition,y),(updateColor,color),
(update,x), (update,y), (update,color)}

임의의 연산자 $o \in O$ 에 대하여 프레임관계 r 하에서 연산자 o 의 상(image) $r(|o|)$는 연산자 o 가 수정할 수 있는 모든 변수의 집합을 나타낸다. 예를 들면, $r(|updatePosition|)$은 {x,y}이다. 따라서 이러한 프레임워크 하에서 클래스의 프레임을 명세하는 것은 프레임관계를 정의하는 것을 의미한다.

프레임관계는 서브클래스와 어떻게 상호작용하는가? S 를 클래스 T 의 임의의 서브클래스라 하고, ΔV 를 S 에 추가된 (새롭게 정의된) 개체변수라고 하자. 즉, ΔV 는 T 에서 상속된 것을 제외한 S 의 새로운 개체변수를 나타낸다. 편의상 V 와 ΔV 가 유별하다고 가정하자 (즉, $\Delta V \cap V = \varnothing$). 클래스 S 가 클래스 T 의 모든 연산자와 변수를 상속 받으므로 S 의 프레임관계는 T 의 프레임관계를 포함하며, 추가로 O 와 ΔV 사이의 정보를 포함하여야 한다. 클래스 S 에 추가된 새로운 연산자를 무시한다면 S 의 프레임관계 r_S 는 O 와 $V \cup \Delta V$ 사이의 관계, 즉 $r_S: O \leftrightarrow (V \cup \Delta V)$이다. 관계 r_S 는 서브클래스에 의해 확장된 프로그램 상태에 대한 O 의 수정권리를 나타낸다. 다음 절에서 자세히 설명되겠지만, r_S는 T 의 프레임관계 r_T와 추가상태의 프레임관계 $\delta r : O \leftrightarrow \Delta V$ 의 합성으로 정의될 수 있다. 따라서 객체지향 명세에서 프레임문제는 어떻게 δr 정의를 효과적이고 모듈성을 지니게 하느냐 하는 문제로 귀결된다.

4. 변수 앵커

4.1 앵커관계

[2] 참고로 증명의 모듈화를 위하여 연산자호출 시 연산자의 구현코드가 아니라 명세가 사용된다 [6].

[3] 본 논문에서는 편의상 Z 표기법을 사용한다. 예를 들면, 관계(relation) $O \leftrightarrow V$ 는 멱집합 $P(O \times V)$를 나타낸다 [8].

서브클래스에서 프로그램 상태를 확장한다는 것은 슈퍼클래스의 추상화를 특정 측면에서 정제하는 것이다. 예를 들면, Sprite 개체의 위치에 관한 정보가 서브클래스 Hero 에서는 구체화되어 조작된다. 클래스 Sprite 는 위치정보를 개체변수 x, y 로 표현하고 조작하지만 서브클래스 Hero 는 개체변수 x, y, dx, dy, ddx, ddy 로 정제한다. 즉, 고정된 좌표뿐만 아니라 이동속도와 가속도까지 표현한다. 따라서 확장 전 상태에서 변수 x, y 를 수정할 수 있는 연산자가 확장된 상태에서 변수 x, y 뿐만 아니라 dx, dy, ddx, ddy 를 수정할 수 있도록 하는 것은 당연하다. 즉, 슈퍼클래스의 추상화 정도에서 어떤 변수를 수정할 수 있다면 서브클래스의 추상화 정도에서는 이 변수를 정제한 모든 변수를 수정할 수 있어야 한다는 뜻이다.

이런 가정하에서 프레임 명세를 위한 새로운 방법을 제안한다. 집합 V 를 슈퍼클래스 T 에 정의된 개체변수라고 하고, ΔV 를 T 의 서브클래스 S 에서 추가로 정의된 개체변수라 하자. 클래스 S 의 프레임관계를 직접 기술하지 않고, S 의 추가변수와 T 의 개체변수 사이의 관계와 T 의 프레임관계를 사용하여 간접적으로 명세할 수 있다. S 의 추가변수 ΔV 와 T 의 변수 사이의 관계를 '앵커관계'라 부른다.

정의 2. 앵커관계(anchoring relation)란 서브클래스에 정의된 추가 개체변수에서 슈퍼클래스의 개체변수로의 사상(mapping)을 말한다.

ΔV 와 V 사이의 앵커관계를 $a: \Delta V \leftrightarrow V$ 로 나타내자. 만약 앵커관계 a 에 의해 변수 x 가 변수 y 와 상관관계를 가진다면 (즉, $(x,y) \in a$), "변수 x 가 변수 y 에 앵커 된다"라고 말한다.

앵커관계는 서브클래스의 확장상태는 슈퍼클래스 상태를 특정 측면에서 정제한 혹은 연관이 있다는 개념을 표현한 것이다. 임의의 연산자가 슈퍼클래스의 특정 변수를 수정할 수 있다면 그 연산자는 해당 변수에 앵커 된 모든 서브클래스 변수를 수정할 수 있다. 따라서 서브클래스의 프레임관계는 다음과 같이 정의된다.

정의 3. 클래스 T 의 프레임관계를 $r_T: O \leftrightarrow V$ 라 하고, $a: \Delta V \leftrightarrow V$ 를 T 의 서브클래스 S 와 T 사이의 앵커관계라 하자. 서브클래스 S 의 프레임관계 $r_s: O \leftrightarrow (V \cup \Delta V)$는 다음과 같이 정의된다.

$$(o,v) \in r_S \Leftrightarrow (o,v) \in r_T \vee (v,v_1) \in a \wedge (o,v_1) \in r_T, \text{ for some } v_1$$

위 정의에서 우변의 첫 째 줄은 슈퍼클래스의 프레임관계는 서브클래스에 상속되고, 둘 째 줄은 추가상태에 대한 수정권리는 슈퍼클래스의 프레임 관계와 앵커관계로부터 간접적으로 주어짐을 나타낸다.

인터페이스 명세에서 앵커관계는 **anchored to** 주석을 사용하여 명세 한다. 예를 들면, 클래스 Hero 의 명세는 다음과 같이 작성될 수 있다.

```
class Hero: public Sprite {
  int dx, ddx; //@ anchored to x;
  int dy, ddy; //@ anchored to y;
  // the rest of def …
}
```

서브클래스의 추가변수 dx, ddx 는 Sprite 의 변수 x 에 앵커 되었고 dy, ddy 는 y 에 앵커 되었다. 따라서 클래스 Hero 와 클래스 Sprite 의 앵커관계 a: {dx,ddx,dy,ddy} ↔ {x,y,color}는 다음과 같이 정의된다.

a = {(dx,x), (ddx,x), (dy,y), (ddy,y)}

그림 1 은 이를 부가적인 정보와 함께 묘사하고 있다. 그림에서 보는 바와 같이 앵커관계는 함수적 관계일 필요가 없다. 그림 1 은 또한 위의 앵커관계와 클래스 T 의 프레임관계로부터 유추할 수 있는 클래스 S 의 프레임관계를 보여주고 있다 (아래 4.2 절 설명 참조).

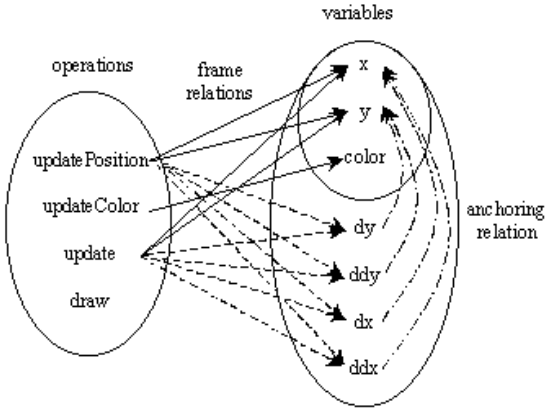

그림 1. 프레임관계와 앵커관계의 예

4.2 연산자에 의해 수정 가능한 변수의 계산

임의의 연산자가 수정할 수 있는 변수의 집합은 어떻게 정의되는가? 만약 o 를 임의의 연산자라 할 때, o 에 의해 수정될 수 있는 모든 변수의 집합 M(o)를 '수정가능집합(modifiable set)'이라 부른다, 수정가능집합 M(o)는 다음과 같이 재귀적으로 정의된다.

(1) 연산자 o 의 **modifies** 절에 나열된 모든 변수는 M(o)에 포함된다.
(2) 만약 변수 x 가 M(o)에 포함되었다면 직접 혹은 간접적으로 x 에 앵커 된 모든 변수는 M(o)에 포함된다.
(3) 이 밖의 변수는 M(o)에 포함되지 않는다.

예를 들면, 제 2 절에 주어진 클래스 Sprite 에 대해 연산자 updatePosition 의 수정가능집합은 {x, y}이다. 하지만 서브클래스 Hero 가 추가되었을 때는 {x, y, dx, dy, ddx, ddy}가 된다. 이는 서브클래스가 추가되면 연산자의 수정가능 집합이 변화될 수 있음을 의미한다. 즉, 수정가능 집합은 확장될 수 있다. 하지만 결코 축소되지는 않는다.

4.3 앵커관계의 특성

앵커관계는 함수관계일 필요가 없다. 서브클래스 변수는 하나 이상의 슈퍼클래스 변수에 앵커 될 수 있다. 즉, 서브클래스 변수는 하나 이상의 슈퍼클래스 연산자에 의해서 수정될 수 있다. 앵커관계는 전칭(total)이어야 하는가? 그렇지 않다. 서브클래스 변수는 아무런 슈퍼클래스 변수에 앵커 되지 않을 수 있다. 이는 상속된 모든 슈퍼클래스 연산자가 해당 변수를 수정할 수 없다는 뜻이다. 앵커 되지 않는 변수는 **anchored to** 주석을 생략함으로써 명세 한다. 혹은 앵커 되지 않는다는 사실을 강조하기 위하여 아래와 같이 **anchored to none** 주석을 사용할 수도 있다.

int x; //@ **anchored to none**;

반면, 슈퍼클래스의 모든 변수에 앵커 되는 서브클래스 변수는 **anchored to all** 주석으로 줄여 쓸 수 있다.

앵커관계는 이행적(transitive)이다. 예를 들면, 클래스 S 의 변수 x 가 S 의 슈퍼클래스 T 의 변수 y 에 앵커 되고, 변수 y 가 T 의 슈퍼클래스 U 의 변수 z 에 앵커 된다면, 변수 x 는 y 에 간접적으로 앵커 된다. 이는 앵커관계가 서브클래스 체인을 통하여 상위클래스로 전파됨을 의미한다.

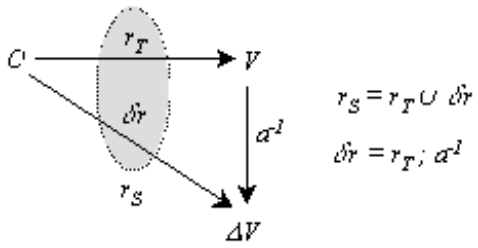

그림 2. 프레임과 앵커의 관계

프레임관계와 앵커관계는 아래 정리 1 과 같은 연관성을 지니고 있다.

정리 1. 임의의 클래스 T 에 대하여 O 와 V 를 각각 T 의 연산자와 변수집합이라고 하고, S 를 추가변수 ΔV 를 갖는 T 의 서브클래스라 하자. 만약 클래스 S 가 추가연산자를 정의하지 않는다면, 클래스 S 의 프레임관계 r_S 는 $r_T \cup \delta r$ 과 같다. 여기서 r_T 는 T 의 프레임관계이고 $\delta r = r_T ; a^{-1}$, 즉 T 의 프레임관계 r_T 와 S 와 T 의 앵커관계 a 의 역과의 관계적 합성을 나타낸다 (그림 2 참조).

정리 1 의 증명은 정의 3 를 적용하면 간단히 할 수 있다.

5. 관련연구

일반적으로 프레임문제에 대한 해결방법은 구문적 접근방법과 의미적 접근방법의 두 범주로 분류할 수 있다. 구문적 접근방법에서는 “그 밖의 모든 것은 변화되지 않는다”와 같은 공리를 명세할 수 있는 구문적 약어를 명세언어가 제공한다. 구문적 접근방법의 예는 VDM-SL 의 외부변수 접근모드(ext wr) [8], Z 의 Δ 표기법 [11], Larch 의 **modifies** 절을 들 수 있다 [3] [4] [5] [6]. 의미적 접근방법에서는 문맥에 따른 프로그램 상태의 변화여부를 공리로 정의한다. 일반적으로 이러한 프레임공리는 연산자의 행위명세인 선·후조건문과 별개로 존재하게 된다. 의미적 접근방법의 대표적 예는 [1]에서 찾아 볼 수 있다.

VDM-SL 은 전통적인 형태의 프레임 명세기법을 제공한다 [8]. 연산자 명세에서 모든 외부변수는 접근모드(**rd**, **wr**)와 함께 선언된다. 명세 된 연산자는 쓰기모드(**wr**)로 선언된 변수만 수정할 수 있다. Z 표기법도 Δ, Ξ와 같은 유사한 관습을 사용하고 있다 [11]. Larch 인터페이스 명세언어는 나열된 변수 이외의 모든 변수는 수정될 수 없음을 나타내는 **modifies** 절을 사용하는데, 이는 후조건문의 일부로 해석된다 [4]. Larch/C++를 비롯한 몇몇 객체지향 명세언어는 상속하에서 프레임문제를 접근하기 위하여 **modifies also** 과 같은 특수 구문을 도입하였다 [6]. 하지만 제 2 절에서 설명한 것과 같은 모듈화 문제를 해결하지 못하고 있다.

Leino 는 프레임문제의 해결방안으로 ‘자료군(data group)’이라는 개념을 도입하였다 [10]. 자료군이란 개체변수의 집합을 나타내는 가상의 변수로 **modifies** 절에 나열 될 수 있다. 자료군에 대한 수정권리는 해당 자료군에 속하는 모든 변수에 대한 수정권리를 의미한다. 자료군 방식은 서브클래스 변수를 특정 슈퍼클래스 변수에 연관시키지 않고 임의적으로 묶을 수 있다는 장점이 있다. 반면 단점은 명세자가 자료군 변수를 명시적으로 선언해야 하며 이들이 개체변수와 뒤섞여 명세의 이해를 어렵게 한다. 게다가 슈퍼클래스 작성시 향후 서브클래스에서 사용될 자료군들을 미리 예측하고 선언해야 한다는 치명적인 단점이 있다. 본 논문에서 제안된 앵커기반의 접근방법은 슈퍼클래스는 아무런 작업이 필요 없다. 또한, 필요하다면 슈퍼클래스에 부가변수를 선언함으로써 자료군 방식과 동일한 효과를 얻을 수 도 있다.

일차논리(first-order predicate logic)만 사용하여 프레임문제를 해결하기 위한 다소 혁신적인 방법이 [10]에서 제안되었다. 이 방법에서는 **modifies** 절과 같은 특수구문을 사용하지 않고, 인터페이스 명세에 '변화공리(change axiom)'라 불리는 프레임공리의 집합이 추가된다. 변화공리는 각각의 추상함수에 대하여 프로그램 상태가 변화될 수 있는 모든 경우를 정의한다. 여기서 추상 함수란 선·후조건문에 사용되는 논리문이나 기타 수학함수를 말한다. 이러한 프레임워크 하에서 서브클래스를 추가하는 것은 서브클래스의 확장상태를 반영하기 위하여 새로운 변화공리를 추가하는 것을 의미한다. 상속된 연산자 명세는 수정할 필요가 없다. 따라서 프레임공리를 명세하는 의무는 연산자 명세에서 자료명세로 이관된다. 이 방법은 명세합성(예: Z 의 스키마 계산법)을 잘 지원하나 연산자 명세와 프레임 명세의 이분화에 따른 복잡성과 모든 추상함수에 대하여 변화공리를 작성해야 한다는 부담이 있다. 또한 이론적으로 잘 정립된 장점이 있으나 널리 보급된 델타방식(Δ, **modifies** 절)과 전혀 다른 방식을 취하므로 실질적인 적용에는 한계가 있다.

6. 맺음말

본 논문에서는 서브클래스와 상속하에서 프레임문제를 다루었다. 프레임문제는 프로그램 상태의 특정 부분이, 특히 서브클래스에 의해서 상태가 확장되었을 때도 변화되지 않고 유지된다는 것을 기술하는 문제이다. 확장된 상태에 대한 프레임공리를 직접명시하지 않고 **anchored to** 주석을 사용하여 확장상태를 개념적으로 원래상태에 연관시킴으로써 간접적으로 명세 한다. 연산자가 슈퍼클래스의 변수에 대하여 수정권리를 갖는다면 해당 변수에 앵커 된 모든 서브클래스 변수에 대해서도 수정권리를 보장 받는다. 그 밖의 서브클래스 변수는 수정되지 않아야 한다. 제안된 방법은 수학적인 바탕 위에서 도입되었다. 먼저 연산자의 수정권리를 프레임관계(frame relation)로 모형화 하였고 서브클래스의 확장상태와 슈퍼클래스의 원 상태간의 개념적 관계는 앵커관계(anchoring relation)로 추상화 하였다. 프레임과 앵커관계에 대한 수학적 특성도 증명하였다. 앵커기반의 접근방식은 프레임문제에 대한 직관적, 실질적, 효과적인 해결책으로 Hoare 방식의 선·후조건문 명세기법과 잘 접목된다.

참고문헌

[1] A. Borgida, J. Mylopoulos, and R. Reiter. On the frame problem in procedure specifications. *IEEE Transactions on Software Engineering*, 21(10):785-798, October 1995.

[2] Y. Cheon. Inheritance in Larch Interface Specification Languages, Its Semantic Foundation and Formal Semantics. In Jim Grundy, Martin Schwenke, and Trevor Vickers, editors, *Proceedings of International Refinement Workshop and Formal Methods Pacific (IRW/FMP) '98, 29 September - 2 October 1998, Canberra, Australia*, pages 81-99. Springer-Verlag, September, 1998.

[3] Y. Cheon and G.T. Leavens. The Larch/Smalltalk Interface Specification Language. ACM Transactions on Software Engineering and Methodology, 3(3):221-253, July 1994.

[4] Y. Cheon and G.T. Leavens. A quick overview of Larch/C++, Journal of Object-Oriented Programming, 7(6):39-49, October 1994.

[5] Y. Cheon and H.-N. Kim. Sequence Operators: Specifying Behavioral Interface of Smalltalk Blocks. In *Proceedings of Asia-Pacific Software Engineering Conference (APSEC '99), Takamatsu, Japan, December 7-10, 1999*, pages 468-475. IEEE Computer Society, 1999.

[6] J.V. Guttag and J.J. Horning. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, New York, N.Y., 1993.

[7] C.A.R. Hoare. An axiomatic basis of computer programming. Communications of ACM, 12(10):576-583, October 1969.

[8] C.B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall, Inc., Englewood Cliffs, N.J., second edition, 1990.

[9] G.T. Leavens. Modular specification and verification of object-oriented programs. *IEEE Software*, 8(4):72-80, July 1991.

[10] K.R.M. Leino. Data groups: Specifying the modification of extended state. *ACM SIGLPAN Notices*, 33(10):144-153, October 1998. OOSPLA '98 Conference Proceedings.

[11] J.M. Spivey. *Understanding Z: A Specification Language and its Formal Semantics*. Cambridge University Press, 1988.