3

명세의 정제와 시뮬레이션 이론

성신여자대학교 ▌서 동 수*

1. 서 론

시스템을 개발할 때 개발자가 가장 어려움을 겪는 부분 중 하나는 사용자의 문제를 명확히 표현하고 이를 검증하는 일이다. 모델링 기법은 사용자의 문제 영역에 존재하는 다양한 개체의 구조, 관계, 그리고 제약을 모델이라는 추상적인 표현 통해 나타낼 수 있게 한다. 모델은 모델링 작업의 결과물이며 동시에 설계 활동과 코딩 작업의 근거로 사용된다.

모델은 모델 명세 언어에 의해 표현된다. 바람직한 모델 명세 언어가 갖추어야 할 특징을 살펴보면 첫째, 사용자의 문제를 자유롭게 표현할 수 있도록 높은 표현력을 가져야 하며 둘째, 개발자와 사용자 모두 명확한 이해를 할 수 있도록 높은 명료성을 제공해야 하며 마지막으로 잘 정의된 구문과 의미를 가지고 있어야 한다. 현재 통신 시스템, 실시간 시스템의 모델링 언어로 사용되는 Petri-net, MSC, 그리고 제어 시스템이나 신뢰할 수 있는(dependable) 소프트웨어를 개발할 때 널리 사용되는 Z[1], B, HOL과 같은 언어는 이러한 특징을 잘 가지고 있는 모델링 언어라 볼 수 있다.

물론, 높은 수준의 표현력 제공을 목적으로 하는 모델링 언어는 컴퓨터 실행을 목적으로 하는 프로그래밍 언어와 확연히 구별된다. 일반적으로 모델링 언어는 높은 표현력을 제공하는 대가로 기계 종속적인 표현이나 분기, 루프 제어와 같은 제어 정보를 표현할 수 있는 수단을 제공하지 않고 있다. 이러한 이유로 모델은 코드처럼 실행 가능한 프로그램 형태를 갖지는 못한다.

모델이 실행가능하지 못하다는 것은 모델이 정확한지를 판단하는데 큰 걸림돌로 작용한다. 이 문제를 완화시키기 위해 개발자들은 모델에 좀 더 구체적인 정보들을 추가하여 애니메이션, 기호실행(symbolic execution), 혹은 시뮬레이션 기법을 도입하여 실행을 대체할 수 있는 수단으로 활용한다.

통신 시스템, 임베디드 시스템, 안전중요(safety critical) 시스템과 같은 엄밀한 시스템을 개발할 때 모델의 검증은 반드시 필요한 활동이다. 실제 항공관련 시스템의 품질을 규정하는 RTCA DO-178B[2]나 정보보호 제품의 인증수준을 규정하는 공통평가기준[3]의 일부 고등급 수준은 모델의 정확성, 일관성, 완전성과 같은 항목에 대한 증명을 요구한다.

본 논문은 모델로부터 시스템을 개발하는 과정에서 생기는 모델의 정제와 시뮬레이션 개념을 정형기법의 관점에서 설명한다. 여러 패러다임 중에 Z나 B와 같은 정형 명세 언어를 사용하는 상태 기반 모델링 기법에서의 정제와 시뮬레이션 개념을 살펴봄으로써 수학적 검증이 활용되는 엄밀한 개발 기법의 특성을 이해하는데 도움을 주고자 한다.

2. 상태기반 명세의 개념

컴퓨터의 특성을 설명하는 가장 고전적인 견해 중 하나는 상태의 관점에서 설명하는 것이다. 프로그램은 컴퓨터로 하여금 초기 상태에서 종료 상태로 움직이도록 한다. 여기서 말하는 상태(state)란 의미 있는 값을 갖는 변수들의 집합이며, 프로그램을 구성하는 오퍼레이션의 수행에 의해 변화된다[4, p4]. 이것은 현재 대부분의 명령적인(imperative) 컴퓨터 언어들이 취하는 방식으로서 입력 자료들을 초기 상태에 배치하고, 종료 상태에 도달하면 출력 자료를 읽음으로써 프로그램의 수행을 종료한다.

프로그램과 상태 사이의 관계는 동전의 앞뒷면과 같다. 누군가가 프로그램을 작성했다면 그 프로그램에 의해 생성되는 상태 집합을 추출할 수 있다. 반대로 누군가가 상태 집합과 이들의 관계에 대한 모델을 정의한다면 이를 표현할 수 있는 프로그램 역시 고안해 낼 수 있다. 그렇다면 프로그램을 먼저 개발할 것인가 혹은 상태에 관한 모델을 먼저 만들 것인가의 결정은 서로 가역적이므로 그 순서가 중요할 것 같지는 않지만 생명주기 모델에 의한 개발을 할 경우

* 정회원

명확한 모델을 만드는 것이 선행되어야한다.

프로그램과 상태의 관계를 예를 통해 살펴보겠다. 프로그램 FindQandR0(x,y)는 두 정수 x, y에 대해 x를 y로 나눈 몫 q와 나머지 r을 구하는 일을 수행한다. 이 프로그램은 다음과 같은 구성을 갖는다고 하자.

```
FindQandR0(x,y){
 r:=x;  q:=0;
 while (r > y) {
    r:= r-y;
    q:= q+1; }
}
```

프로그램이 올바로 작성되었는지 확인하고 싶다면 프로그램의 수행에 의해 얻을 수 있는 상태가 올바른가를 보고 판단할 수 있다. 만일 FindQandR0(10, 4)인 경우 이 프로그램의 수행을 통해 생성되는 상태는 다음과 같다.

(상태 1) (x=10∧y=4) // 시작 전 상태
(상태 2) (x=10∧y=4∧r=10∧q=0) // 루프 앞
(상태 3) (x=10∧y=4∧r=6∧q=1) // 첫 루프 종료
(상태 4) (x=10∧y=4∧r=2∧q=2) // 둘째 루프종료
(상태 5) (x=10∧y=4∧r=2∧q=2) // 종료 후 상태

이 프로그램은 시작 전 상태인 (상태 1)과 종료 후의 상태로 (상태 5)를 가지며 이들 사이에는 3 개의 중간 상태를 갖는다. 이들 모든 상태는 원하는 값들을 정확히 가지고 있으므로 올바른 수행을 했다고 판단한다. 그러나 이러한 분석 방식은 입력 변수의 범위가 큰 경우 (예를 들면 $x=10^{10}$, y=4인 경우) 프로그램이 만들어 내는 상태가 급격히 늘어나게 되므로 그리 좋은 방법은 아니다.

이와는 반대의 접근으로 상태에 관한 모델을 먼저 만들고 이를 근거로 코딩을 하는 것을 생각해볼 수 있다. 이 방법은 프로그램이 바람직하게 작동되었을 때 얻을 수 있는 시작조건과 종료조건을 먼저 명세한 후 코드 내부에서 만족시켜야 할 조건을 추가시킨다. 이러한 방식을 상태기반(state-based) 명세라 하며 상태기반 명세에서는 프로그램에 대한 선조건(pre-condition)과 후조건(post-condition)을 명세의 근간으로 이용한다. 선조건은 프로그램 시작 전의 상태를 표현하며, 후조건은 프로그램 종료 후 만족해야 하는 조건을 표현한다. 이들은 술어조건 형식을 가지며 따라서 참, 거짓 값만을 갖는다. 선조건을 pre, 후조건을 post라 할 때 명세 프레임은 다음의 구조를 갖는다.

[pre] 코드 [post]

프로그램 FindQandR0에 대해 제수 y는 0보다 큰 양의 정수로, r, q는 x=y*q+r인 관계를 만족하는 수로 정의할 경우 프레임으로 표현하면 다음과 같다.

[y>0] FindQandR0(x,y) [x=y*q+r]

프레임 내의 선조건과 후조건이 항상 올바른 조건을 포함해야 하는 것은 정확한 프로그램이 되기 위한 요구사항이다. 사실 위의 선조건과 후조건은 오류를 포함한다. 만일 입력 값이 x= -1이거나 혹은 (x=6, y=3)이 입력된 경우 프로그램 FindQandR0은 선조건과 후조건을 만족함에도 불구하고 원하는 결과를 얻을 수 없음을 알 수 있다. 이러한 오류를 방지하기 위해서는 프레임 내의 선조건과 후조건을 다음과 같이 강화시킨 후 FindQandR0의 구조가 올바른지 확인할 필요가 있다.

[0≤x∧y>0] FindQandR1(x,y) [0≤r<y∧x=y*q+r]

강화된 조건을 만족하는 프로그램을 FindQandR1(x,y)라 할 때 이 프로그램의 내부 조건은 다음과 같다.

```
[0≤x ∧ 0<y ]
FindQandR1(x,y){
// r:=x; q:=0;
[0≤r ∧ 0<y ∧ x=y*q+r ]
    //while (r ≥ y) {
       [0≤r ∧ 0<y≤r ∧ x=y*q+r ]
         // r:= r-y;
         // q:=q+1;
       [0≤r ∧ 0<y ∧ x=y*q+r ]
    // }
// }
[0≤r<y ∧ x=y*q+r ]
```

선조건과 후조건은 프로그램이 가져야 할 바람직한 상태가 표현한다는 점에서 중요하다. 어떤 프로그램의 입력 자료나 출력 결과가 이들 조건을 참으로 만족시키지 못할 경우 우리는 프로그램의 수행 결과 역시 올바름을 보장할 수 없게 된다. 그러한 예로 만일 x 값이 음수인 경우 선조건 [0≤x ∧ 0<y]은 거짓이 되므로 초기 상태가 존재하지 않는다. 따라서 이 경우 프로그램이 종료 상태에 도달한다는 보장을 할 수 없게 된다.

3. 모델의 정제

요구모델에는 구현에 관한 정보가 포함되어있지 않다. 그러나 개발자가 모델의 구현에 관한 정보들, 예를 들면 구체적인 자료구조나 실행제어에 관한 정보

를 추가한다면 이 모델은 구현에 가까운 모델, 혹은 상세한 모델로 변경될 수 있다. 이와 같이 어떤 모델에 구체적인 정보를 추가함으로써 상세한 모델로 만드는 작업을 모델의 정제(refinement)라 부른다.

의미적으로 동일한 두 모델 M_1, M_2에 대해 M_2가 M_1에 비해 구현 정보를 더 많이 포함하고 있다면 M_2는 M_1을 정제한다고 말하며 기호로는 $M_1 \sqsubseteq M_2$로 표현한다. 정제의 개념을 시스템 개발 과정으로 확장시키면 다음과 같은 설명이 가능하다. 먼저 최상위 수준의 추상(abstract)모델 A에 대해 일련의 중간 정제 과정을 거치면 결국 코드 C를 얻을 수 있다.

$$A \sqsubseteq M_0 \sqsubseteq M_1 \sqsubseteq \ldots \sqsubseteq M_{n-1} \sqsubseteq M_n \sqsubseteq C$$

선조건과 후조건은 명세를 정제하는 중요한 근거로 사용된다. 정제는 다음 두 가지 방식으로 진행된다. 먼저 개발자는 후조건을 강화함으로써 명세를 정제시킬 수 있다.

(방법 1) 후조건의 강화: 모델 M과 선조건 pre, 후조건 post, post'에 대해 만일 post' $\Rightarrow$post 이면,

$$[pre]\ M\ [post] \sqsubseteq [pre]\ M\ [post']$$

이 방법은 새로운 후조건 post'를 기존의 후조건 post에 대한 충분조건으로 강화시킴으로써 명세를 정제할 수 있다는 의미이다. 예를 들면 다음의 M_1, M_2에 대해 $M_1 \sqsubseteq M_2$인 관계가 성립한다.

M_1: $[0 \le x \wedge y > 0]$ FindQandR(x,y) $[x = y*q+r]$
M_2: $[0 \le x \wedge y > 0]$ FindQandR(x,y) $[0 \le r < y \wedge x = y*q+r]$

후조건 강화가 좋은 이유는 후조건에 제한을 가함으로써 개발자는 프로그램 수행 결과에 대해 더 자세한 정보를 알 수 있기 때문이다.

또 다른 방법으로 선조건을 완화시킴으로써 명세를 정제하는 것을 생각할 수 있다.

(방법 2) 선조건의 완화: 모델 M과 선조건 pre, pre'와 후조건 post에 대해 만일 pre $\Rightarrow$pre' 이면,

$$[pre]\ M\ [post] \sqsubseteq [pre']\ M\ [post]$$

이 방법은 새로운 선조건을 완화시킴으로써 예전 선조건이 새로운 선조건에 포함되도록 만드는 방법이다. 예를 들어 입력 x의 범위를 $0 \le x < 10$하는 명세보다는 $0 \le x$로 확장함으로써 조건을 완화시킨다. 따라서 다음의 명세 M_1과 M_2 사이에는 $M_1 \sqsubseteq M_2$인 관계가 성립한다.

M_1: $[0 \le x < 10 \wedge y > 0]$FindQandR(x,y)$[0 \le r < y \wedge x = y*q+r]$
M_2: $[0 \le x \wedge y > 0]$FindQandR(x,y)$[0 \le r < y \wedge x = y*q+r]$

정형 개발은 다른 방식의 개발에 비해 엄밀성의 정도가 높은 정제 작업을 수행한다. 일반 개발 방식의 경우 구현은 명세의 이해로부터 출발한다는 점에서 동일하지만 설계나 코딩 과정의 여러 판단은 개발자의 경험과 직관에 의해 선택되어진다. 하지만 정형 개발에 있어서의 이 과정은 정제 활동으로 대체되며, 정제 전의 모델과 정제 후의 모델 간에는 수학적인 매핑 과정이 정의되어야 한다. 추상 수준이 다른 두 개의 모델 간에 수학적인 관계가 있다는 것은 이들 사이에 올바른 정제가 수행되었다는 것을 증명할 수 있는 메커니즘이 존재한다는 것을 의미한다.

프로그램이 오퍼레이션에 의해 초기 상태로부터 종료 상태로 이동한다는 것을 정형적으로 표현하면 다음과 같다. 프로그램 P를 구성하는 요소로 자료집합 D와 오퍼레이션 집합 O가 있다고 하자. 자료형 D는 초기화 연산 $d_{init} \in D \leftrightarrow D$ (단, 관계 기호 $D \leftrightarrow D$는 D의 카테지안 곱의 멱집합 $P(D \times D)$을 의미한다), 종료연산 $d_{final} \in D \leftrightarrow D$을 가지며, 오퍼레이션 O는 인덱스된 오퍼레이션 op_i의 집합으로 구성되며 이들은 연산자 결합(기호로 ⨾를 사용한다)에 의해 순서지어진다. 만일 오퍼레이션 $op_1, .., op_n \in D \leftrightarrow D$인 경우 프로그램 P(D)의 한 가지 가능한 구성은 다음과 같다.

$$P(D) = d_{init} \fatsemi op_1 \fatsemi .. \fatsemi op_n \fatsemi d_{final}$$

프로그램의 정제는 얼마나 구현에 가까운 자료형을 사용하는가에 영향을 받는다. 개발자가 선택한 자료형 D와 D'가 서로 다른 수준의 자료형이라 할 때 이들을 사용하는 프로그램 사이에는 정제 관계가 존재한다. 만일 프로그램 P(D)와 P(D')사이에 다음의 관계가 성립한다고 하자.

$$P(D) \sqsubseteq P(D')$$

이 경우 D'에 적용되는 모든 인덱스된 오퍼레이션 단계는 D내의 인덱스된 단계에 대해 시뮬레이션 할 수 있다. 만일 $P(D) \sqsubseteq P(D')$인 관계를 증명하려 한다면 다음 연산 사이에 존재하는 시뮬레이션 관계를 보여야 한다.

$$d_{init} \fatsemi d_{final} \sqsubseteq d'_{init} \fatsemi d'_{final}$$
$$d_{init} \fatsemi op_1 \fatsemi d_{final} \sqsubseteq d'_{init} \fatsemi op'_1 \fatsemi d'_{final}$$
$$d_{init} \fatsemi op_1 \fatsemi .. \fatsemi op_n \fatsemi d_{final} \sqsubseteq d'_{init} \fatsemi op'_1 \fatsemi .. \fatsemi op'_n \fatsemi d'_{final}$$

4. 모델 시뮬레이션

주어진 두 프로그램 P(D), P(D') 사이에는 정제 관계가 존재하는지, 만일 그렇다면 누가 상대방을 정제

하는 것인지를 증명하는 것은 그리 간단하지 않다. 예를 들어 시스템 자원의 배분에 관한 모델링을 할 경우 추상 명세 수준에서는 자원과 프로세스의 요소를 집합으로 표현한다. 반면에 구체 모델에서 이들은 큐 혹은 리스트 형태로 표현된다.

이 과정에서 개발자는 리스트 표현이 상위 단계 명세에서 나타난 집합 모델의 올바른 반영이라는 점을 시뮬레이션을 통해 확인해야 한다. 따라서 우리는 구체 명세가 과연 추상 명세의 올바른 정제인지를 검증하지 않은 상태에서 구현에 들어간다면 문제가 생길 수 있다고 본다.

명세 P(D)와 P(D')의 사이에 P(D) ⊑ P(D') 관계가 있다면 이들을 구성하는 오퍼레이션의 시퀀스 사이에도 정제 관계가 보전되어야 함은 앞서 설명한바가 있다.

그림 1의 윗 부분은 d_{init} ⨾ op_1 ⨾ .. ⨾ op_n ⨾ d_{final}을, 아랫부분은 d'_{init} ⨾ op'_1 ⨾ .. ⨾ op'_n ⨾ d_{final}을, 그리고 가운데 관계 f는 자료형 D, D'간에 동일한 인덱스 집합을 갖는 경우 정의 가능한 각 인덱스 단계의 매핑을 보여준다.

이 과정에서 각 op_i와 op'_i 사이의 매핑을 설명해주는 관계 $f \in D \leftrightarrow D'$가 존재하여야 한다. 시뮬레이션 과정에서 관계 f의 존재는 중요하다. 만일 명세의 정제 과정에서 관계 f가 존재함을 보일 수 없다면 개발자는 올바른 정제를 수행하였는지를 직관적으로 판단할 수 밖에 없다. 나아가 코드가 설계 명세의 정확한 정제인지, 또는 설계 명세가 요구 명세의 정확한 정제인지를 논리적으로 증명해야 할 경우 이를 입증할 수 없는 문제가 발생한다.

그림 1과 같은 관계에서 다음 조건이 만족된다면 관계 f를 전진 시뮬레이션(forward simulation)이라 한다.

(조건 1) d_{init} ⨾ f ⊑ d'_{init}

(조건 2) f ⨾ d'_{final} ⊑ d_{final}

(조건 3) op_i ⨾ f ⊑ f ⨾ op'_i, 단 i는 모든 인덱스를 포함한다.

이들 세 가지 조건의 의미는 그림 2에 나타난다. (조건 1)과 (조건 2)는 그림 내의 (1), (2)에서와 같이 자

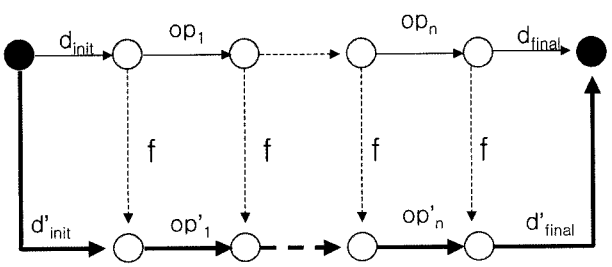

그림 1 D와 D' 사이의 관계

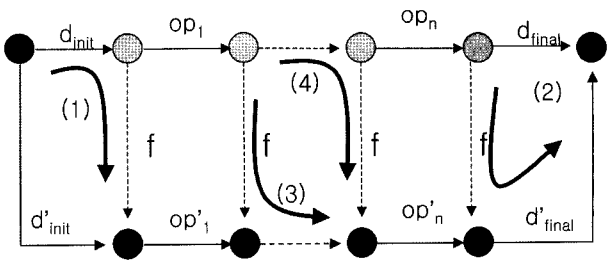

그림 2 전진 시뮬레이션

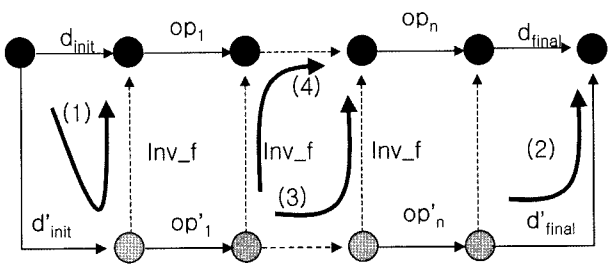

그림 3 후진 시뮬레이션

료에 대한 초기화 연산과 종료 연산이 각각의 보전되어야 함을 나타내며, (조건 3)은 그림 2의 (3), (4)와 같이 구체 명세의 오퍼레이션으로 매핑이 일어나야 함을 보인다.

일반적인 경우는 명세 P(D)로부터 유도된 P(D')가 올바르다는 점을 확인하기 위해 전진 시뮬레이션을 하지만 경우에 따라 P(D)⊑P(D')인지 혹은 P(D')⊑P(D)인지 확인하기 힘든 경우가 있다. 이 경우는 다음과 같은 후진 시뮬레이션을 적용하는 것이 효과적다.

그림 3의 경우 여기서 나타나는 관계 inv_f는 관계 f에 대한 역함수로서 구체 명세에서 추상 명세로 돌아갈 수 있는 관계가 정의된다는 것을 보여준다. 만일 inv_f가 정의될 수 있는 관계라면 다음의 조건을 만족하는 inv_f를 후진 시뮬레이션(backward simulation)이라 한다.

(조건 4) d_{init} ⊑ d'_{init} ⨾ inv_f

(조건 5) inv_f ⨾ d_{final} ⊑ d'_{final}

(조건 6) inv_f ⨾ op_i ⊑ op'_i ⨾ inv_f, i는 모든 인덱스를 포함한다.

앞의 예와 동일하게 초기 연산에 관한 (조건 4), 종료연산에 관한 (조건 5)의 내용은 그림 3의 (1), (2)에, 연산 시퀀스의 관계에 관한 (조건 6)은 (3),(4)에 나타난다.

5. 시뮬레이션 검증 케이스스터디

전진 시뮬레이션의 예로서 메모리에 프로세스를 적재하고 제거하는 Z 명세인 Memory0를 살펴보자. 집합 [Process]는 프로세스를 원소로 가지는 자유타입이며

Memory0는 적재되는 프로세스 집합의 총 개수가 max 이하가 되도록 제한한다.

$[Process]$

$$Memory0 \mathrel{\widehat=} [pset : \mathbb{P}\, Process \mid \#pset \leq max]$$

MemoryInit0는 초기화 오퍼레이션에 의해 pset을 공집합 상태로 초기화한다. 단, Memory0'와 pset'은 변경 후의 변수를 의미하는 장식으로 오퍼레이션 수행 결과 값이 변경되는 경우와 그 이전을 구분하기 위해 사용된다.

$$MemoryInit0 \mathrel{\widehat=} [Memory0' \mid pset' = \varnothing]$$

Z는 스키마라 불리는 구조를 명세에 이용한다. 스키마는 선언부와 술어부로 구성되며 이들은 각각 변수 선언과 이들에 대한 제약을 담고 있다.

schema name

declaration
predicate

스키마를 사용하는 적재와 제거 오퍼레이션을 살펴보면 다음과 같다.

Load0

$\Delta Memory0$ $i? : Process$
$\#pset < max$ $i? \notin pset$ $pset' = pset \cup \{i?\}$

Unload0

$\Delta Memory0$ $i? : Process$
$i? \in pset$ $pset' = pset \setminus \{i?\}$

Load0는 입력 프로세스 i가 이미 적재된 프로세스 집합의 원소가 아니어야 하고 (i? ∉ pset, 단 ?는 입력 변수임을 나타내는 장식이다), 변화 전의 pset의 개수는 max 미만이어야 하며, pset은 적재가 요청된 프로세스 i를 적재 집합에 포함해야 함을 조건으로 담고 있다.

Unload0는 제거할 프로세스 i가 이미 적재된 프로세스 집합의 요소이어야 하며 (i? ∈ pset), 오퍼레이션 수행 결과 pset'은 pset에서 원소 i가 제거된 상태와 동일해야 함을 선언한다(단, 기호 \ 는 집합제거 연산자이다).

Memory0의 정제된 명세로 다음과 같은 명세 Memory를 살펴보자. 주목할 점은 Memory0가 집합을 자료형으로 사용한 것임에 비해 이 명세는 원소 간의 순서가 부여된 시퀀스(seq)를 자료형으로 사용한다.

$$Memory \mathrel{\widehat=} [lseq : \mathrm{seq}\, Process \mid \#lseq \leq max]$$

$$MemoryInit \mathrel{\widehat=} [Memory' \mid lseq' = \langle\rangle]$$

초기화 오퍼레이션 MemoryInit는 시퀀스 내에 원소가 없도록 정의한다.

Load

$\Delta Memory$ $i? : Process$
$\#l < max$ $i? \notin \mathrm{ran}\, lseq$ $lseq' = lseq \frown \langle i? \rangle$

Unload

$\Delta Memory$ $i? : Process$
$i? \in \mathrm{ran}\, lseq$ $lseq' = lseq \upharpoonright (Process \setminus \{i?\})$

Load 오퍼레이션은 요청된 프로세스 i가 기존의 시퀀스에 있는 요소가 아닌지 확인(i? ∉ ran lseq)한다. 이 과정에서 치역 연산자 ran을 사용하여 확인하는 이유는 자료형 시퀀스의 정의상 $1 \mapsto pset_1$, $2 \mapsto pset_1, \ldots$, $n \mapsto pset_n$과 같은 매핑에 대한 치역을 추출해야 하기 때문이다. 또한 명세는 lseq에 대해 적재가 요청된 프로세스 i를 새로운 시퀀스에 포함해야 함을 규정한다. 유사한 방식으로 Unload 오퍼레이션은 시퀀스 필터 연산자 ↾ 를 사용하여 i만 삭제된 시퀀스를 생성하도록 제약한다.

명세 Memory0와 Memory의 차이는 프로세스를 표현하는 방법으로 집합과 시퀀스를 이용하는 것이고 이에 따라 각각에 적용할 수 있는 별개의 연산자를 사용한다는 점이다. 따라서 이들을 사용하는 오퍼레이션 Load0와 Load, 그리고 Unload0와 Unload 사이에는 시뮬레이션 관계가 성립하는지 확인해야 한다. 이를 가능하게 하기 위해서는 적절한 시뮬레이션 함수 f가 정의되어야 한다. 시뮬레이션 함수 f는 다음과 같이 정의할 수 있다.

f

$\Delta Memory0$ $\Delta Memory$
$pset = \mathrm{ran}\, lseq$

스키마 f의 술어부는 lseq의 치역 연산자 ran에 의해 얻어진 집합은 다름 아닌 pset 집합과 동일해야 한다는 것을 명시한다. 과연 f가 올바른 시뮬레이션 함수이어서 Memory가 Memory0를 정제하고 있는지를 확인하기 위해서는 몇 가지 정리가 증명되어야 한다.

앞서 설명된 (조건 1)과 (조건 3)의 내용은 다음과 같은 술어 정리 (초기화 규칙)과 (관계 규칙)으로 대체하여 사용할 수 있다(이 대체 과정은 증명이 필요하며 자세한 내용은 [2]에서 찾을 수 있다).

(1) 초기화 규칙:

$$\forall D' \bullet d'_{init} \Rightarrow d_{init} \wedge f$$

(2) 관계 규칙:

$$\forall D;D' \bullet pre\ op_i \wedge f \Rightarrow pre\ op'_i$$

$$\forall D;D' \bullet pre\ op_i \wedge f \wedge op_i \Rightarrow \exists D \bullet op_i \wedge f$$

이들 두 규칙에 근거하여 Memory0와 Memory 시스템에 적용하면 얻을 수 있는 정리는 다음과 같다.

(정리 1) $\forall$ Memory' • MemoryInit $\Rightarrow$
($\exists$ Memory0' • MemoryInit0 $\Rightarrow$ f')

(정리 2) $\forall$ Memory0; Memory •
pre Load0 $\wedge$ f $\Rightarrow$ pre Load

(정리 3) $\forall$ Memory0;Memory •
pre Unload0 $\wedge$ f$\Longrightarrow$ pre Unload

(정리 4) $\forall$ Memory0; Memory; Memory' •
pre Load0 $\wedge$ f $\wedge$ Load $\Rightarrow$
($\exists$ Memory0' • f' $\wedge$ load0)

(정리 5) $\forall$ Memory0; Memory; Memory' •
pre Unload0 $\wedge$ f $\wedge$ Unload $\Rightarrow$
($\exists$ Memory0' • f' $\wedge$ Unload0)

개발자는 이상에서 파악된 5가지 증명 규칙이 참임을 증명함으로써 명세 Memory는 Memory0의 올바른 정제라는 것을 증명할 수 있다.

6. 맺음말

애니메이션이나 기호 실행, 혹은 시각화 기법을 사용하여 시뮬레이션을 하는 방식은 개발자들이 널리 선호하는 방식이다. 그러나 고신뢰 시스템이나 보안 시스템, 통신 시스템과 같은 엄밀성이 요구되는 분야는 모델의 엄밀한 검증을 중요시한다. 엄밀성의 정도에 차이는 있지만 증명 수준의 엄밀성이 요구될 경우 기존의 시뮬레이션 기법은 한계가 있다.

개발 과정 초기에 올바른 모델을 확보하는 것은 중요한 일이다. 그러나 이 작업 못지않게 중요한 것은 모델 정제를 통해 구체적인 정보가 포함된 구현모델을 개발하는 일이다.

본 논문에서는 모델의 정제 개념과 이 과정에서 적용될 수 있는 전진 시뮬레이션과 후진 시뮬레이션 이론을 간단히 살펴보았다.

정형적인 시뮬레이션을 통한 검증은 그 복잡성으로 인해 증명기와 같은 지원도구의 도움을 필수로 한다. 일부 증명 과정은 기계적인 반복적용이 가능한 부분이므로 축약하거나 생략할 수 있지만 실제적인 크기의 문제에 적용하는 것은 아직도 복잡하고 방대한 작업임에는 틀림없다.

국외의 경우 정형적 정제와 시뮬레이션이 우주항공, 국방, 자동차 제어 등 다양한 시스템 개발에 시도되고 있지만 국내에는 이러한 방식의 개발 기법이나 도구의 보급이 초보 수준에 머무르고 있다. 적용 가능한 분야의 발굴과 이를 활용할 수 있는 전문가 양성이 절실히 필요한 때이다.

참고문헌

[1] J Woodcock, J, Davis, Using Z Specification, Refinement, and Proof, Prentice Hall, 1996

[2] RTCA DO 178-B, Software Consideration in Airborne Systems and Equipment Certification, 1992

[3] 정보시스템 공통평가기준, 정보통신부 2002

[4] C Morgan, Programming from Specification, Prentice Hall, 1990

서 동 수

1987 중앙대학교 컴퓨터공학과(학사)
1989 중앙대학교 컴퓨터공학과(석사)
1994 Univ. of Manchester, Dept. of Computation (석사, 박사)
1994~1998 전자통신연구원, 선임연구원
1998~현재 성신여자대학교 컴퓨터정보학부 부교수

관심분야 : 소프트웨어공학, 정형기법, 정보보호기술
E-mail : dseo@sungshin.ac.kr