병행객체지향 언어에서 행위 방정식을 이용한 상속 변칙

이 호 영*, 이 준**

Inheritance Anomaly using Behavior Equation in Concurrent Object-Oriented Programming Languages

Ho-young Lee*, Joon Lee**

요 약

상속 변칙이 발생하는 기본적인 이유는 병행객체에 대해 동기화 코드가 메소드 코드와 적당하게 분할되지 않을 때, 파생 클래스를 만들어내기 위한 코드의 확장이 슈퍼 클래스에 존재하는 동기화 코드와 메소드 코드를 변경하도록 할 때, 그리고 병행 객체지향 언어에서 상속성과 병행성이 결합할 때 발생한다. 강조할 점은 상속 변칙을 피하는 방법이다. 그래서 본 논문에서는 새로운 모델인 객체 모델을 제안하고 행위 방정식을 사용하여 병행 객체지향 언어에서 나타나는 상속 변칙의 문제를 최소화시키고자 한다.

Abstract

The fundamental reason why inheritance anomaly occurs is that for a concurrent object, when synchronization code is not properly separated from the method code, the extension of code to produce a derived class may force the change of both the synchronization code and the method code in the super class, and inheritance is integrated inheritance in a simple and satisfactory way within a concurrent object-oriented language. The main emphasis on how to avoid or minimize inheritance anomaly. Therefore, in this paper we propose a new model, object model, and will minimizes the problem of inheritance anomaly found in concurrent object-oriented programming languages using Behavior Equation.

※ 이 논문은 1998년도 조선대학교 학술 연구비의 지원을 받아 연구되었음"

* 조선대학교 컴퓨터공학과

** 조선대학교 컴퓨터공학부

접수일자 : 1999년 7월 16일

I. 서 론

지난 10여년 동안, 병행 프로그램에 객체지향성의 개념을 도입하기 위한 많은 연구가 있었다. 상속성은 객체지향 언어의 중요한 특성이다. 그러나 병행성이나 실시간 명세를 포함한 확장된 객체지향 언어에서는 문제가 발생한다. 이러한 문제 중의 하나가 상속 변칙이다. 병행 객체지향 프로그램밍 언어는 객체의 동기화 제약 조건에 대한 표현 기능을 제공한다. 이것은 프로그래머가 full 상태인 큐에 기록하거나 empty 상태인 큐로부터의 판독이 허용되지 않는 표현의 가능성을 요구한다.

상속 변칙은 상속된 메소드의 확장된 재정의가 병행 객체의 동기화 제약조건 유지에 필수적일 때 병행성과 상속성 사이의 충돌이다. 일반적으로 병행 객체지향 프로그래밍 언어에서 일어나는 상속 변칙의 유형은 허용상태의 분할, 허용상태의 과거 민감성, 그리고 허용 상태의 변경으로 분류할 수 있다.

이러한 상속 변칙의 문제점을 최소화하기 위해 본 논문에서는 새로운 객체모델을 제안하고 이 모델을 기반으로 행위 방정식을 사용하여, 어떻게 이 모델이 상속 변칙의 문제를 해결하는가를 보여준다. 2장에서는 일반적인 상속변칙에 대한 기본적인 개념 및 문제점들을 기술하였으며, 3장에서는 2장에서 살펴본 문제점들을 해결하기 위한 제안된 모델을 설계하였으며, 4장에서는 제안된 모델을 기초로 행위 방정식을 사용하여 상속 변칙 문제를 해결하고, 마지막으로 5장에서는 결론으로 끝을 맺는다.

Ⅱ. 상속 변칙

상속 변칙은 병행 프로그래밍에 객체지향의 개념을 도입할 때 병행성과 상속성의 충돌, 동기화 클래스를 보유한 클래스의 선언을 재사용할 경우 모든 형태의 변경을 지원할 동기화 코드가 없기 때문에 발생한다. 즉, 상속 변칙은 상속 계층내의 한 클래스의 메소드 변경이 그 부모 클래스는 물론 자식 클래스의 메소드 재정의를 요구하게 된다. 많은 방법들이 그들 사이의 간섭을 제거하기 위해 제안되었다[1]. 이러한 제안은 두가지 형태의 접근 방법으로 분류된다. 첫 번째 접근 방법은 메소드 guard를 이용한 방법이다. 메소드 guard는 각 메소드에 선행자를 할당하고, 메시지가 만약 그 선행자를 만족하면 받아들여진다. 그러나 accept-set 접근 방식에서, 메소드 코드내에서 받아들여져야 하는 메시지들의 다음 집합은 메소드 코드내에 명시된다. 따라서 병행성과 상속성의 혼합은 객체 지향의 중요한 특징 중의 하나인 캡슐화를 깨뜨리기 때문에 재사용을 불가능하게 한다[2][3].

1. 상속 변칙의 분석

상속 변칙은 허용 상태의 분할, 허용 상태의 과거 민감성, 그리고 허용 상태의 변경 중 하나의 조건하에서 발생한다. 즉, 만약 각각 세 개의 조건하에서, 코드의 상속은 메소드 코드의 어떤 재정의도 필요로 하지 않고 동시에 캡슐화가 보장된다면, 병행 객체 지향 프로그래밍 언어는 대부분의 실제적인 경우에서 상속 변칙 문제가 발생하지 않을 것이다[3][4][5].

상속변칙의 발생은 그림 1과 같이 경계버퍼(bounded buffer)와 같은 경우에 발생한다.[3][4].

그림 1과 같이 경계버퍼는 두 개의 메소드 get()과 put(), 그리고 하나의 인스턴스 변수 num으로 구성되어 있다. put() 메소드는 클래스 buf에 하나의 아이템을 저장하고, 반면에 get() 메소드는 클래스 buf로부터 하나의 아이템을 제거한다. 그리고 인스턴스 변수 num은 현재 클래스 buf 내에 존재하는 전체 아이템의 수를 의미한다. 클래스 gget_buf는 클래스 buf의 서브클래스이다. 그리고 클래스 gget_buf는 세 개의 메소드, 즉 get(), put(), 그리고 gget()와, 하나의 인스턴스 변수 num으로 구성되는데, put()과 get() 메소드, 그리고 인스턴스 변수 num은 수퍼 클래스로부터 상속받는다. 메소드 gget()의 행위는 gget_buf 내에서 연속하는 두개의 아이템을 제거하는 것을 제외하고는 메소드 get()과 비슷하다. 물론, 이러한 메소드 호출을 위하여, 클래스 buf는 버퍼에 적어도 두개의 아이템을 가져야 한다.

상속 변칙을 해결하기 위해서는 병행 프로그램에서 수퍼 클래스로부터 상속받는 메소드는 어떠

```
Class buf: {
   int num = 0, SIZE;
Behaviour:
   Initial_State = {empty};
   empty         = {put};
   partial       = {put, get};
   full          = {get};
Members:
   void  put() {
         num += 1;
         ... ;
         if[num = SIZE}          Become full
         else                    Become partial;
  }
   void  get() {
         num -= 1;
         ... ;
         if[num = 0]             Become empty
         else                    Become partial;
   }

Class gget_buf:buf {
Behaviour:
   Initial_State  = Super(Initial_State);
   empty          = Super(empty);
   one            = Super(partial);
   partial        = Super(partial) + {gget};
   full           = Super(full) + {gget};
Members:
   void  gget() {
         num = num - 2;
         ... ;
         if[num = 0]             Become empty
         elseif [num = 1]        Become one
         else                    Become partial;
   }
   void  put() {
         num += 1;
         ... ;
         if[num = SIZE]          Become full
         elseif [num < SIZE]     Become partial
         else                    Become one;
   }
   void  get() {
         num -= 1;
         ... ;
         if[num = 0]             Become empty
         elseif [num = 1]        Become one
         else                    Become partial;
   }
}
```

그림 1. 경계버퍼에서의 상속 변칙

Fig. 1. Inheritance Anomaly in Bounded Buffer

한 재정의도 없이 수행되어야 한다. 그러나, 행위 추상화의 개념을 사용하는 그림 1의 코드는 상속된 메소드가 파생 클래스에서 재정의되어야 한다는 것을 보여준다. 두개의 클래스 buf와 gget_buf 내에 존재하는 메소드 put()의 정의를 비교하면, 메소드 gget()에 대해, 하나 이상의 state와 하나 이상의 become이 필요하다는 것을 알 수 있다. 즉, 이것은 캡슐화를 만족시키지 못하여 상속 변칙이 발생하였다는 것을 보여준다.

2. 상속 변칙의 유형

상속성은 객체지향 언어의 주요 특징 중의 하나이다. 그러나 병행성이나 실시간 명세를 포함한 확장된 객체지향 언어에서는 문제가 발생한다. 이러한 문제 중의 하나가 상속 변칙이다. 상속변칙은 상속된 메소드의 재정의가 병행 객체의 동기화 제약조건을 유지하는데 필수적일 때 병행성과 상속성 사이의 충돌과 동기화 방식에서 발생한다. 즉 확실한 동기화 방식을 채택하는 병행 객체지향 프로그래밍 언어에서 안전하게 상속되는 클래스일지라도 파생된 클래스의 재정의가 필요하다. 이것은 언어에서 객체 방식 동기화와 상속성의 기술 사이에 의미적인 충돌 문제가 중심이 된다. 더구나 병렬/분산 언어에서 개발된 이전의 기술 또는 시스템은 응용이 가능한지가 불분명하다.

상속 변칙은 일반적으로 가드 메소드 방식에 의한 변칙과 허용 집합 명세에 의한 변칙으로 나눈다. 가드 메소드 방식에 의한 분류는 허용할 수 있는 가드를 충족시키는 메시지인 각 메소드에 부울식인 가드 술어를 할당하는 조건부 동기화 방식이다. 허용 집합 명세에 의한 분류는 자신의 메소드 코드에 허용되기 위한 메시지의 다음 집합을 충족하는 한 객체를 허용할 수 있는가에 따른 분류이다.

마쓰오카(Matsuoka)와 요네자와(Yonezawa)는 병행 객체지향 프로그래밍 언어에서 발생할 수 있는 상속 변칙의 유형을 다음과 같은 세 가지로 분류하였다[5][6].

- 허용 상태의 분할(partitioning of acceptable states)
- 허용 상태의 과거 민감성(history-only sensitiveness of acceptable states)
- 허용 상태의 변경(modification of acceptable states)

A. 허용 상태의 분할

그림 1에서 클래스 buf의 서브 클래스인 gget_buf 클래스는 세 개의 메소드, 즉 get(), put(), 그리고 gget()과 하나의 인스턴스 변수 num으로 구성되는데, 여기서 put()과 get() 메소드, 그리고 인스턴스 변수 num은 슈퍼 클래스로부터 상속받는다. 메소드 gget()의 행위는 gget_buf 클래스의 버퍼 내에서 가장 오래된 원소 두 개를 동시에 제거하는 것을 제외하고는 메소드 get()과 비슷하다. 물론, 이러한 메소드 호출을 위하여, 버퍼는 내부에 적어도 두개의 원소를 가져야 한다. 즉, 클래스 gget_buf라 명명된 buf의 서브클래스에 두 개의 원소를 제거하는 메소드 gget()을 추가하면, accept set partial은 두개의 accept set인 one과 partial이라는 두 개의 상태로 분할되어야 하는데, 이것은 단지 한 개의 원소가 버퍼 상에 존재하는지 또는 존재하지 않는지의 상태를 알아내는 것이 필요하기 때문에 분할되어야 한다. 이러한 이유로 get()과 put() 메소드는 재정의된다. 결과적으로 클래스 buf에서 서브 클래스 gget_buf로 초기화를 제외한 메소드의 어떤 것도 상속되지 않는다. 그래서 허용상태의 분할 변칙이 발생한다.

상속 변칙에 자유로운 병행 프로그램에서, 슈퍼 클래스로부터의 상속받는 메소드는 어떠한 재정의도 없이 수행되어야 한다. 그러나, 행위 추상화의 개념을 사용하는 그림 1의 코드는 상속된 메소드가 파생 클래스에서 재정의 되어야 한다는 것을 보여준다. 만약 두개의 클래스 buf와 gget_buf 내에 존재하는 메소드 put()에 대한 정의와 메소드 get()의 정의를 비교해 보면, 메소드 gget()에 대해, 하나 이상의 상태와 하나 이상의 become 문장이 필요하다는 것을 알 수 있다. 즉, 캡슐화가 깨져, 상태분할 변칙이 발생한다.

B 허용 상태의 과거 민감성

수퍼 클래스 b-buf의 서브클래스인 gb-buf에 단일 메소드 gget()이 추가된 클래스의 정의는 [3]의 그림 4.8과 같다. 그림에서 메소드 gget()의 행위는 메소드 put()의 호출 이후에 즉시 허용될 수 없는 하나의 예외를 포함한 메소드 get()과 거의 동일하다. 호출을 위한 이러한 조건은 method guards와 클래스 b-buf에서만 유효한 인스턴스 변수의 집합을 식별할 수 없다 이것은 여분의 인스턴스 변수 after-put을 필요로 한다. 그 결과 메소드 get()과 put() 모두 재정의 되어야 한다. 결과적으로 허용 집합 기반 방식과 유사한 상황이 발생한다. 이러한 변칙 발생 이유는 클래스 b-buf에서 guard 선언을 포함한 허용된 gget 메시지를 위한 상태를 판단할 수 없다는 것이다. 즉, gget은 b-buf의 인스턴스에 대하여 추적 또는 과거-민감성 메소드이다. 메소드의 동기화 제약은 역사 정보에 의존한다. 만약 gb-buf라 불리는 b-buf의 서브 클래스가 메소드 put()의 요청 후에 즉시 받아질 수 없는 예외를 가진 get() 메소드에 거의 동등한 메소드 gget()을 추가한다면, 'after execution put' 상태는 경계 버퍼로부터 상속된 변수의 기초에 정의될 수 없다. 이런 경우, 새로운 변수의 값을 설정하거나 재설정해야 하기 때문에 새로운 변수인 after_put을 추가하고, 메소드 put()과 get()을 재정의할 필요가 있다. 결과적으로 허용상태의 과거 민감성 변칙이 발생한다.

C. 허용 상태의 변경

새로운 메소드의 추가는 슈퍼클래스로부터 상속된 메소드가 요청할 수 있는 상태 집합을 수정한다. [3]의 그림 4.9처럼 클래스 lock이 메소드 unlock의 요청을 받고, 승낙하고, 실행할 때까지 메소드 lock의 실행을 통해 모든 메시지의 서비스를 중단하는 클래스를 고려하면, 이 클래스는 b-buf 클래스를 새로운 클래스인 Lock에 재정의하기 위해 사용될 수 있다. 이런 경우, lock에 의해 추가되는 동기화 제약이 그들의 객체가 lock될 때는 실행될 수 없기 때문에, get()과 put()의 승낙 가능한 상태의 수정을 야기한다. 결과적으로 b-buf로부터 상속된 메소드를 재정의 할 필요가 있다. 게다가 put()과 get()의 경우, 이미 정의된 메소드들의 동기화 제약 조건(lock과 unlock은)을 변경해야 한다. 결국 b-buf의 method guards는 새로운 제약 조건에 대한 일관성 유지를 위하여 변경되어야 하며, 이로 인해 상태 변경 변칙이 발생한다.

Ⅲ. 객체 모델의 설계

전형적인 객체 모델이 실패한 주요한 이유는 모델이 인터페이스 컨트롤 공간과 객체의 인스턴스 변수에 대한 메소드 코드의 배타적인 작업에 기반을 두고 있기 때문이다. 이런 두 부분 사이에서 종속에 대한 완전한 분할은 설계의 복잡성을 제거하는 방법을 제공한다. 그러나, 이전 모델에서, 이러한 객체의 두 부분들은 완전한 교차되지 않는다. 결과적으로, 이 모델은 상속 변칙의 모든 가능한 상황을 다룰 수 없다[7][8]. 이러한 문제점으로 인하여 발생하는 상속 변칙의 문제점을 해결하기 위해서, 본 논문에서 설계한 객체 모델에서는 동기화 코드와 객체 멤버 코드 사이의 관계 종속성을 최소로 유지하도록 하여 상속 변칙의 문제점을 해결하고자 한다.

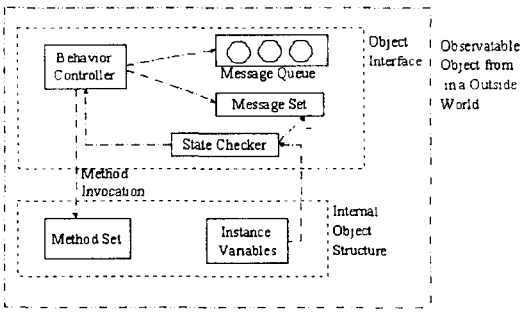

그림 2. 행위 방정식에 기반한 객체 모델
Fig. 2. Object Model Based on Behavior Equation

동기화 코드와 객체 멤버 코드 사이의 관계를 나타내는 제안된 모델은 그림 2와 같다. 이 모델은 두 부분으로 나누어진다.

- 객체 생성과 초기화 : 객체가 생성될 때, 객체는 초기 상태에 놓이게 되고, 메시지 셋은 객체의 초기 상태와 동등하게 된다. 객체의 상태 정보는 상태 검색기(state checker)에 저장된다.
- 객체의 내부 연산 : 메소드들의 실행 후, 인스턴스 변수는 갱신된다. 객체의 다음 상태는 상태 검색기내에 존재하는 인스턴스 변수의 갱신된 값에 의해 얻어진다. 현재 상태의 정보와 인스턴스 변수의 값에 의해, 다음 enabled-set이 얻어진다. 반면에, 상태 검색기는 enabled-set이 다음에 사용할 메시지 셋을 통지한다. 마지막으로, 행위 컨트롤러는 행위 검색기로부터 다음 상태에 대한 정보, 메시지 셋으로부터의 enabled-set, 그리고 메소드가 호출될 지를 결정하기 위해 메시지 큐로부터 다음 메시지 키를 수집한다.

입력 중인 메시지는 메시지 큐에 들어간다. 객체가 생성될 때, 각 객체는 초기 상태에 놓여지게 되고, 메시지 셋은 객체의 초기 상태와 같이 enable된다. 행위 컨트롤러는 객체의 다음 행위를 결정하기 위해 사용된다. 동등한 메소드를 호출하기 전에 행위 컨트롤러는 메시지 큐로부터 입력 메시지를 읽는다. 이 입력 메시지는 단지 enable된 메시지 셋에 포함되는지를 검사한다. 만약 입력 메시지가 유효하면, 동등한 메시지가 호출된다. 그러나 입력 메시지가 유효하지 않다면, 예외 핸들러가 호출된다. 메소드 실행 후, 객체의 인스턴스 변수는 변경된다. 상태 검색기는 이전의 입력 메시지와 선행자에 종속되는 객체의 다음 상태를 찾아낸다. 마지막으로, 상태 검색기는 객체의 다음 상태가 무엇인지를 행위 컨트롤러에 통지한다.

Ⅳ. 행위 방정식을 이용한 상속 변칙의 해결

본 장에서는, 행위 방정식의 개념과 제안된 모델을 기반으로 하여 상속 변칙의 문제를 해결한다. 먼저, 경계 버퍼의 다양한 형태를 증명한다. 이것은 많은 연구자들이 상속 변칙에 대한 해결책을 설명하기 위해 폭넓게 사용되어졌기 때문에 선택되었다. 상속 변칙에 대한 문제점을 기술하고, 또한 어떻게 행위 방정식이 2장에서 기술한 상속 변칙의 다양한 경우를 해결하는지를 증명한다.

1. 행위 방정식

행위 방정식은 모델에 대해 객체의 병행성을 기반으로 하여 다음과 같이 기술된다.

STATE = TERM+ TERM + ... (1)

TERM = A_ACT{[PRED][PRED]...}.OBJ_STATE;
..... (2)

STATE = Based(CLASS.STATE + CLASS.STATE + ...)
{-([TERM]), + ([TERM])};
..... (3)

식 (1)은 객체가 특별한 상태에 있을 때, 객체의 행위를 표현하기 위해 사용된 행위 방정식이다. 식 (2)는 객체의 행위가 어떻게 구성되는가를 나타낸다. 이 방정식에서 TERM은 객체의 행위를 나타낸다. 주어진 메소드 혹은 원자적 실행 후에, 선행자 [PRED]가 검사된다. 만약 [PRED]가 참이면, [ORJ_STATE]는 다음 상태가 된다.

식 (3)은 파생 클래스의 정의에서 나타난다. Based는 Based() 후에 지시된 수퍼 클래스의 모든 행위들의 조건을 의미하는 키워드로, 파생 클래스에 상속된다. 다중 상속의 경우에, 더 많은 수퍼 클래스가 명시될 수 있다. -([TERM])은 수퍼 클래스로부터 상속된 주어진 상태에 속하는 행위들이 무시될 수 있다는 것을 의미한다. 그리고 +([TERM])은 주어진 상태에 속하는 새로운 행위가 도입되었다는 것을 의미한다.[4]

2. 허용상태의 분할

2장에서 기술한 클래스 buf는 경계버퍼에서 객체의 모든 가능한 다음 상태의 행위를 보여주기 위해 행위 방정식과 C++ 코드를 사용하여 그림 3과 같이 재정의된다.

```
CLASS buf{
    int SIZE, num = 0;
BEHAVIOR_CONTROLLER:
    EMPTY   =   put.SOME:                                   (4)
    SOME    =   put[0 < num < SIZE].SOME +
                put[num = SIZE].FULL +
                get[0 < num < SIZE].SOME +
                get[num = 0].EMPTY;                         (5)
    FULL    =   get.SOME;                                   (6)
MEMBERS;
    void  put(int item)
          {num++; ...};   //Put an item into the bounded buffer.
    int get()
          {num--; ...};   //Remove an item from the bounded buffer.
}
```

그림 3. 클래스 buf
Fig. 3. Class buf

그림 3에서 재정의한 클래스 buf는 두개의 주요 부분, 즉 BEHAVIOR_CONTROLLER와 MEMBERS로 구성된다. 이 두 부분은 각각 동기화 코드 및 메소드 코드와 일치한다. 지역 변수 SIZE는 경계버퍼의 크기를 나타낸다.

식 (4)는 객체의 현재 상태가 EMPTY이고 입력 메시지가 put()이면, 다음 상태는 그림 4에 기술된 식 (7)의 상태 ONE이다는 것을 나타낸다. 식 (5)는 현재의 상태가 SOME이므로 put() 메소드와 get() 메소드를 모두 허용한다. 그리고 식 (6)은 현재의 상태가 FULL이므로 get() 메소드만 허용한다.

클래스 buf의 서브클래스인 클래스 gget_buf는 행위 방정식과 C++ 구문을 사용하여 그림 4와 같이 재정의된다. 클래스 gget_buf에는 한번에 두개의 아이템을 획득하는 gget()라 불리는 추가적인 메소드가 추가되었다. 이 클래스의 나머지 행위들은 클래스 buf의 행위와 구별되지 않는다. 그것은 gget_buf가 클래스 buf로부터 상속받기 때문이다. 그러나 버퍼가 단지 하나의 아이템을 가지는 경우를 다루기 위해서는 부가적인 상태가 필요하다. 그렇게 때문에 이런 상태하에서는 gget() 메소드를 호출할 수 없다. 그래서 상태 SOME을 식 (9)와 같이 변경하였다.

```
CLASS gget_buf :buf {
BEHAVIOR_CONTROLLER:
    EMPTY   =   put.ONE;                                    (7)
    ONE     =   put.SOME + get.EMPTY;                       (8)
    SOME    =   put[0 < num < SIZE].SOME +
                put[num = SIZE].FULL +
                get[0 < num < SIZE].SOME +
                get[num = 1].ONE  +
                gget[num = 0].EMPTY  +
                gget[num = 1].ONE  +
                gget[0 < num < SIZE].SOME;                  (9)
    FULL    =   get.SOME + gget.SOME;                       (10)
MEMBERS:
    Int gget();
        {num++;.} //gget will remove two consecutive items
                  //in the gget_buf
}
```

그림 4. 클래스 gget_buf
Fig. 4. Class gget_buf

그림 4에서 상태 ONE을 포함하는 식 (8)은 클

래스 gget_buf 내에서의 새로운 행위 방정식이다. 이 식과 상태 ONE은 클래스 buf에는 정의되어 있지 않은 새로운 상태이다. 결과적으로, 클래스 buf부터 gget_buf까지 식 (8)과 관계된 행위의 상속은 불가능하다.

클래스 buf의 상태 EMPTY에 대한 행위 방정식 즉, 식 (4)와 클래스 gget_buf의 상태 EMPTY에 대한 행위 방정식 즉, 식 (7)을 비교하면, 식 (7)에서의 행위는 put.SOME을 제외하고 식 (4)에서 기술된 행위들이다. 따라서, gget_buf의 SOME 상태는 클래스 buf의 SOME 상태에 기반을 두고 있으며 다른 행위는 무시된다. 더욱이 행위 put.ONE은 gget_buf를 정의하기 위해 행위 방정식 (4)에 추가된다. 결과적으로, gget_buf의 새로운 행위 방정식 EMPTY가 도입된 것이다. 그리고 상태 SOME 즉, 식 (9)에는 연속하는 두 개의 아이템을 제거하기 위하여 메소드 gget() 추가하였다. 다른 행위들도 EMPTY와 비슷한 방법으로 구해진다.

그림 3과 그림 4로부터 어떤 메소드의 재정의도 필요하지 않는다는 것을 알 수 있다. 즉, 메소드 코드가 수퍼 클래스로부터 상속받는다는 것이다. 게다가, 행위 방정식을 사용하여 동기화 코드 재사용 또한 가능하다. 결과적으로 상태 분할시 발생하는 상속 변칙 문제가 해결되었다.

3. 허용상태의 과거 민감성

본 절에서는 허용 가능한 상태 문제의 과거 민감성이라는 상속 변칙을 해결하기 위해 행위 방정식을 사용하여 증명한다. 이러한 변칙을 해결하기 위해 클래스 hsget_buf라는 새로운 병행 객체를 정의한다. 이 객체에서, 새로운 메소드 hsget()는 클래스 buf에 추가된다. 메소드 hsget()의 행위는 메소드 get()의 호출 후 즉시 받아질 수 없는 예외를 가진 get()과 비슷하다. 행위 방정식과 C++ 구문을 사용하여 정의한 클래스 hsget_buf는 그림 5와 같다.

그림 5에서 클래스 hsget_buf의 수퍼 클래스는 클래스 buf이다. hsget_buffer 클래스의 행위 방정식에서, 식 (11)의 EMPTY는 buf 클래스의 식 (4)의 EMPTY와 동일하다. 클래스 hsget_buffer의 SOME 즉 식 (12)는 식 (5)에서 SOME의 get.EMPTY와

```
CLASS hsget_buf : buf {
BEHAVIOR_CONTROLLER :
    EMPTY   =  put.SOME;                                   (11)
    SOME    =  put[0 < num < SIZE].SOME +
               put[num = SIZE].FULL +
               hsget[0 < num < SIZE].SOME +
               hsget[num = 0].EMPTY + get.GET;             (12)
    FULL    =  get.GET + hsget.SOME;                       (13)
    GET     =  get[num = 0].EMPTY +
               get[num < SIZE].GET + put.SOME;             (14)
MEMBERS :
    int  hsget()
        { num++; ··· } // The method hsget() will get an item
                       // excepted that it cannot be accepted
                       // immediately after the method get().
}
```

그림 5. 클래스 hsget_buf
Fig. 5. Class hsget_buf

get.SOME를 제거하고, 새로운 행위인 get.GET, naget.EMPTY 그리고 hsget.SOME을 추가한 것이다. 식 (13)의 FULL도 식 (6)에서의 get.SOME를 제거하고 get.GET를 추가한 것이다. 마지막으로 식 (14)의 상태 GET은 인스턴스 변수 num의 값이 0이면 get.EMPTY 메소드를 호출하거나, num이 SIZE보다 적은 값이면 자신을 재귀 호출하거나, put.PARTIAL을 호출한다.

결과적으로, 그림 5로부터 메소드 코드와 동기화 코드 상에서는 상속 변칙이 발생하지 않는다. 상속된 메소드의 어떤 것도 재 정의를 필요로 하지 않는다. 이것은 동기화 코드 내에서, 더 많은 행위가 제거되고, 다른 행위가 도입되었기 때문이다.

4. 허용상태의 변경

이 절에서는 클래스 lock_buf를 도입하여 허용가능 상태의 변경에 대한 상속 변칙을 해결한다. 행위 방정식과 C++ 코드를 사용한 클래스 lock의 정의는 그림 6과 같다.

그림 6에서, 클래스 lock의 인스턴스 변수 lock_c는 객체의 다중 레벨 lock을 제공하기 위해 객체가 몇번 lock 되었는가의 횟수를 계산하는데 사용된다. 그리고 메소드 lock()은 객체를 lock시키고, 카운터 lock_c의 값을 하나 증가시킨다.

```
CLASS  lock {
   int  lock_c = 0 ;
BEHAVIOR_CONTROLLER :
   LOCK     =  lock.LOCK + unlock[lock_c = 0].UNLOCK
               + unlock[lock_c > 0].LOCK;          (15)
   UNLOCK   =  lock.LOCK;                          (16)
MEMBERS :
   void lock()
        { lock_c++ ; ··· } // locking the object and increasing
                           // the lock counter, lock_c, by one.
   void unlock()
        { lock_c-- ; ··· }  // unlocking the object and decreasing
                           // the lock counter, lock_c, by one.
```

그림 6. 클래스 lock

Fig. 6. Class lock

```
CLASS lock_buf : buf, lock {
STATE_CONTROLLER :
   EMPTY    =  put.SOME + lock.LOCK
   SOME     =  lock.LOCK +
               put[0 < num < SIZE].SOME +
               put[num = SIZE].FULL +
               get[0 < num < SIZE].SOME +
               get[num = 0].EMPTY
   FULL     =  get.SOME + lock.LOCK
   LOCK     =  lock.LOCK +
               unlock[num = 0][lock_c = 0].EMPTY +
               unlock[0 < num < SIZE][lock_c = 0].SOME +
               unlock[num = SIZE][lock_c=0].FULL +
               unlock[lock_c > 0].LOCK
MEMBERS :
```

그림 7. 클래스 lock_buf

Fig. 7. Class lock_buf

그림 7은 클래스 lock_buf를 정의하는 코드이다. 여기서 새로운 클래스인 lock_buf의 수퍼 클래스는 클래스 lock과 클래스 buf이다. 그러나 두 개의 수퍼 클래스인 클래스 lock과 클래스 buf의 동기화 코드의 재사용이 파생 클래스인 lock-buf에서 메소드의 중복을 일으키지 않는다. 그리고 클래스 buf와 클래스 lock에서의 모든 행위가 재사용 되고, 수퍼 클래스의 모든 행위들도 재사용 된다. 결과적으로 행위 지향적이고 객체지향적인 언어가 상속 변칙의 문제를 해결한다.

V. 결 론

병행 객체지향 프로그래밍의 목적은 객체의 병행성을 통한 처리의 극대화, 강력하고 유연한 소프트웨어 설계 등이다. 그러나 병행 객체지향 프로그래밍 언어에서 상속성과 병행성의 결합으로 인하여 발생하는 간섭 현상 즉, 상속 변칙은 분산 환경에서 공유 메소드의 어려움과 상속 변칙으로 인한 캡슐화 파괴 및 코드 재사용을 저해하는 문제점이 생긴다. 이러한 상속 변칙의 문제점을 해결하기 위해 최근까지도 많은 연구자들이 노력하고 있으나, 아직까지 완전하게 상속 변칙을 해결하지는 못하고 있는 실정이다.

본 논문에서는 허용 집합을 토대로 상태 분할 변칙, 허용 상태의 과거-민감성 및 상태 변경 변칙 등의 상속 변칙을 해결하기 위해 새로운 객체 모델을 설계하여 이를 바탕으로 문제가 발생된 상황에 대한 해결로 수퍼 클래스로부터 파생된 각각의 서브 클래스를 정의하고, 이러한 정의를 뒷받침하는 각 메소드의 행위를 행위 방정식을 사용하여 구현하였다. 이러한 구현으로 상속 변칙 현상을 해결하였으며, 또한 메소드 코드와 동기화 코드 사이의 간섭 문제에 대한 해결책을 제공하였다. 결과적으로 상속변칙이 발생한 모든 조건들이 성공적으로 처리되었다.

향후 이런 행위 방정식 모델의 향상을 위해 단일 상속뿐만 아니라 다중 상속의 경우에도 적용 가능한 실제의 다각적인 상황에 대비하여 연구할 계획이다.

참고문헌

[1] J. Meseguer, "Solving the inheritance anomaly in concurrent object-oriented programming", ECOOP'93 Object-Oriented Programming. 7th European Conference Proceedings., pp. xi+531, 220-46, 1993.

[2] C. Tomlinson, V. Singh, "Inheritance and Synchronization with Enables-Sets", OOPSLA'89 Proceedings. , Vol. 24, pp. 103-112, 1989.

[3] G. Agha, P. Wegner, A. Yonezawa, Research Directions in Concurrent Object-Oriented Programming, Massachusetts, MIT Press. , pp. 107-150, 1993.

[4] C. Barry, L. Leung, P. Peter, K. Chiu, "Solution of inheritance anomaly in concurrent object-oriented programming languages", IEEE'96, Proceedings of PDP'96, pp. 360-366, 1996.

[5] L. Thomas, "An Object-Oriented Concurrent Language for Extensibility and Reuse of Synchronization Components", Computers and Artificial Intelligence. , Vol. 15, No. 5, pp. 437-457, 1996.

[6] S. Frölund, "Inheritance of Synchronisation Constraints in Concurrent Object-Oriented Programming languages", Proceedings of ECOOP'92, pp. 185-196, 1992.

[7] D. G. Kafura, K. H. Lee, "Inheritance in Actor Based Concurrent Object-Oriented Language", ECOOP'89, pp. 131-145, 1989.

[8] L. Crinogorac, S. Rao, K. Ramamohanarao, "Inheritance anomaly. A formal treatment", FMOODS'97, Vol. 2, pp. vii+470, 319-34, 1997.

[9] S. E. Mitchell, A. J. Wellings, "Synchronisation, concurrent object-oriented programming and the inheritance anomaly", Comput. Lang. , Vol. 22, No. 1, pp. 15-26, 1996.

[10] M. Karaorman, J. Bruno, "Introducing Concurrency to a sequential language", Communication of the ACM, Vol. 36, No. 9, pp. 103-116, 1993.

[11] D. Caromel, "Toward A Method Of Object-Oriented Concurrent Programming", Communications of the ACM Vol. 36, No. 9, pp. 90-102, 1993.

[12] S. Matsuoks, K. Taura, A. Yonezawa, "Highly Efficient and Encapsulated Re-use of Synchronization Code in Concurrent Object-Oriented Languages", OOPSLA'93, Vol. 28, No. 10, pp. 109-126, 1993.

[13] G. Agha : "Concurrent Object-Oriented Programming", Communications of the ACM, Vol. 33, No. 9, pp. 125-141, 1990.

이 호 영 (Ho-Young Lee)
1988년 2월 조선대학교 전산기공학과 졸업(공학사)
1995년 8월 조선대학교 대학원 컴퓨터공학과 졸업(공학석사)
1999년 2월 조선대학교 대학원 컴퓨터공학과 박사과정 수료
*주관심분야 : 객체지향 프로그래밍, 분산 운영체제, 멀티미디어

이 준 (Joon Lee)
1979년 2월 조선대학교 전자공학과(공학사)
1981년 2월 조선대학교 대학원 전자공학과(공학석사)
1997년 2월 숭실대학교 대학원 전자계산학과(공학박사)
1982년 3월 ~ 현재 조선대학교 공과대학 컴퓨터공학부 교수
*주관심분야 : 분산 운영체제, 병렬처리, 프로그래밍 환경, 컴파일러