韓國 OA學會 論文誌
第6卷 第4號, 2001. 12.

2001-6-4-1-6

상속성과 병행성 통합에서 오는 상속 변칙 분석

오 승 재*

Analysis of Inheritance Anomaly due to Integrated of Inheritance and Concurrency

Seung-jae Oh*

요 약

병행성(concurrency)과 객체지향 패러다임을 통합하는 문제가 여러 해 동안 많은 사람들의 목표가 되어 왔다. 병행 프로그래밍과 객체지향 프로그래밍 기법을 결합한 병행 객체지향 프로그래밍 언어는 병행 응용 프로그램을 개발하는데 여러 가지 이점을 얻을 수 있다. 그러나 병행성과 상속성은 서로 충돌하는 특성을 가지기 때문에 캡슐화를 심각하게 파괴하지 않고는 동시에 사용하는 것이 어렵게 된다. 이러한 상속변칙 현상은 응용성을 제한하게 되고, 동기화가 발생하는 모든 문제를 처리하지 못하게 된다.

본 논문에서는 상속성과 객체 기반의 병행성 사이의 충돌 현상 해결에 행위 기술 방정식을 도입하였다. 그 결과, 여러 가지 상속 변칙 문제를 해결할 수 있고 메소드의 재정의가 불필요하게 되었다. 따라서 동기화와 함수 코드의 재사용을 인하여 유연성을 가진 병행성 모델을 제공할 수 있다.

Abstract

The integration of concurrency and object-oriented programming paradigm has been a goal of researchers for over a decade. The concurrent object-oriented programming languages used for various concurrent applications development. However, concurrency and inheritance have conflicting characteristics, thereby inhibiting their simultaneous use without heavy breakage of encapsulation. This conflicting phenomenon is known as inheritance anomalies, have been limited in applicability and have not addressed all the issues that synchronization raises.

In this paper, the behavior description equation is introduced for the solution for the conflicting phenomenon, between inheritance and object-based concurrency. As a result, it can solve the several inheritance anomalies and is not necessary to redefine method. Therefor, it offers flexible concurrency model with an emphasis on the reuse of both synchronization and function code.

* 순천청암대학 컴퓨터정보과학과 조교수

Ⅰ. 서론

병행 프로그래밍을 위해 객체(Object)의 병행성(Concurrency)을 지원할 뿐만 아니라 객체의 상속성(inheritance)과 재사용성(reusability), 캡슐화(encapsulation) 등 객체지향 프로그래밍 언어의 특성도 동시에 지원할 수 있는 언어를 병행 객체지향(Concurrent Object-Oriented) 프로그래밍 언어라고 한다. 병행성과 객체지향 개념을 통합하게 되면 병행성과 상속성 사이에 서로 충돌하는 현상이 발생한다. 병행 객체의 통합성을 유지하기 위해서 반드시 필요한 상속 메소드의 재정의가 요구되는 현상을 상속변칙이라고 한다. 상속변칙이 일어나는 정확한 원인과 유형을 파악하기가 쉽지 않아서 상속변칙 해결에 많은 어려움이 따른다[1][2].

본 연구에서는 허용 집합 내에서 객체의 다음 상태의 행위에 바탕을 둔 행위 기술 방정식 모델을 설계하고, 이 모델을 이용하여 여러 가지 상속 변칙의 문제를 해결하여, 상속성과 객체 기반의 병행성 사이의 충돌 문제 해결과 코드 재사용을 개선하고자 한다.

Ⅱ. 병행 객체지향 언어의 개념

1. 병행 객체지향 언어에서 병행성과 상속성

병행 객체지향 언어는 병행 프로그래밍과 객체지향 개념을 결합한 언어로써, 병행적으로 존재하는 여러 객체들 사이에 객체들의 행위가 동시에 실행되도록 허용하거나, 한 객체의 내부에서 다수의 스레드가 동시에 실행되도록 허용할 수 있다. 병행 객체지향 프로그래밍의 목적은 객체들의 병행적인 처리를 통하여 처리기(processor)의 처리 능력을 증가시키는 데 있다[1][3].

병행성이란 병행 객체들 사이에 동시에 발생하는 모든 독립적인 활동을 말한다. 서로 독립적인 프로세스들이 병행 수행되기 위해서는 병행 수행되는 프로세스들 사이의 상호 작용 문제가 해결되어야 한다. 병행 객체가 병행 상태이면 내부의 무결성(integrity)을 유지하기 위해서 메시지 집합의 일부를 허용하게 되는데, 이 때 허용 가능한 메시지 집합에 제한성을 둔다[3][4].

객체지향 프로그래밍 언어에서 캡슐화(encapsulation)란 관련된 데이터와 프로세스를 결합하여 캡슐로 만들어서 객체들의 내부 정보를 숨기려는 특성이다. 또 상속성이란 클래스 간의 계층 경로를 따라 부모 클래스(parent class)로부터 자식 클래스(child class)에 속성들을 물려주는 개념이다. 상속성에 의해 부모 클래스에서 새로운 자식 클래스를 생성하게 되면 객체들 간에 공통된 속성을 갖는 메소드나 데이터는 자동적으로 공유하게 되고, 현재 클래스의 새로운 부분을 추가하여 확장함으로써 생성된다. 그 결과, 강력한 타입 검사와 효율적인 코드 재사용을 지원하게 되어 소프트웨어의 재사용과 유지·보수성, 유연성, 동적 특성 등을 향상시킬 수 있다.

2. 병행성 제어

객체들에 대한 병행 수행이 일어나면 객체의 상태에 맞는 동기화를 요구한다. 한 객체의 내부의 상태는 메소드 호출에 의해서 접근이 가능하다. 따라서 병행적으로 활동하는 객체들의 병행성 제어 기술은 객체의 내부에서 구현하게 된다. 병행성 제어는 집중 제어 방식과 분산 제어 방식으로 구분할 수 있다[6].

집중 제어 방식은 단일 절차 상에 병행성 제어를 집중화시키는 제어 방식으로, POOL-T 언어의 body 구조나 Extended Eiffel 언어의 Live 구조와 같은 병행 제어를 말한다. 집중 제어 방식을 사용하는 언어의 경우에 메시지 수신은 가드(guard)나 SELECT 구조의 사용에 의해 명시적인 표현을 사용한다. 또 병행성과 상속성을 결합하게 되면 집중화된 절차로 명세한 동기화 제약 조건이 서브 클래스에 상속될 수 없다. 따라서 집중 제어 방식을 사용하는 언어에서는 재사용되는 동기화 코드의 상속이 허용되지 않는다.

분산 제어 방식은 집중화된 절차가 없는 메소드 사이의 분산된 병행성 제어이다. 분산 제어 방식을 사용하는 언어로는 Actor 모델을 기반으로 한 언어인 Act++, Rosette, Hybrid 언어가 있다.

Ⅲ. 상속 변칙의 개념과 원인

1. 동기화와 가용집합

병행 객체지향 프로그래밍 언어는 병행 객체의 동기화를 표현할 수 있는 기능이 제공되어야 한다. 병행 접근에 대한 객체들의 상태의 보호를 위해서 동기화(synchronization) 코드가 필요하다. 동기화 코드는 객체의 행위의 동기화 제어를 위해서 객체의 행위로부터 메소드 코드를 분리한 것이기 때문에 반드시 동기화 제약 조건과 일치해야 하며, 병행 객체의 동기화 제약 조건을 유지하려면 상속 메소드의 확장을 위한 재정의가 요구된다. 동기화를 표현하기 위해 호출되는 메소드의 집합이 가용집합(enabled set)이며, 병행 객체들이 수신할 수 있는 객체들의 상태에 달려 있다[8][9].

가용집합(enabled set)이란 동기화 제약 조건의 분해의 한 형태이며, 동기화에 대한 강력한 접근의 한 형태이다. 즉 가용집합은 새로운 상태에서 실행될 메시지를 정의하는 집합이다. 어떤 유형의 요구는 서브 클래스의 변경을 일으키지 않고 슈퍼 클래스에 추가된 새로운 메소드를 충족한다. 또는 슈퍼 클래스에서 관련없이 상속되는 연산을 위해 동기화 제약 조건을 손상시키지 않는 서브 클래스를 추가한다. 따라서 객체의 다음 상태의 명세는 가용 집합을 수반하게 된다. 객체의 다음 상태가 가용 집합 내에서 결정되면 unlock인 한 객체를 재호출하게 되며, 이 때 동기화 제약 조건은 객체들이 적용하는 상태에 직접 결합이 가능한 가용 집합 속으로 분해된다.

2. 상속 변칙의 개념

상속성과 병행성을 통합하게 되면 공유 프로토콜의 통합이 어렵게 되거나, 상속성에 간섭을 일으키거나 하게 된다. 상속 변칙은 병행성과 상속성을 함께 사용할 때 병행성과 상속성 사이의 충돌로 인하여 발생하는 현상으로써, 부모 메소드가 전혀 상속되지 않을 수 있는 순차 객체지향 프로그래밍 언어에서 더 심각하다. 즉 상속 변칙은 동기화 코드가 메소드 코드에서 적절하게 분리되지 않거나, 서브 클래스의 생성을 위해 코드를 확장할 때 슈퍼 클래스에서 동기화 코드와 메소드 코드의 변경이 요구되는 경우에 발생한다. 그 결과, 서브 클래스에 추가될 슈퍼 클래스의 메소드를 부적절하게 재정의하게 되며, 이로 인하여 병행 객체의 코드가 상속될 때 클래스의 캡슐화가 파괴되고, 재사용이 어렵게 된다.

상속 변칙의 유형에 대해서는 허용 집합 명세에 의한 변칙을 기반으로 하여 구분한다. 허용 집합을 기반으로 한 동기화 방식은 수시로 허용할 수 있는 메소드를 결정하기 위하여 명시적인 집합을 사용하게 된다. 허용 집합은 명시적인 집합을 사용할 때 슈퍼 클래스의 변경을 요구하는 서브 클래스에 메소드의 추가를 허용할 수 있는 집합이다. 메시지의 허용 집합에 의한 객체의 상태 변화로 서브 클래스의 정의에 종속되어 각각 다른 형태로 발생하는 세 가지의 상속 변칙은 허용 상태의 분할(partitioning of acceptable states), 허용 상태의 과거 민감성(history-only sensitiveness of acceptable states), 허용 상태의 변경(modification of acceptable states)이다[1].

허용 집합의 상태 분할 변칙은 그림 1에서와 같이 become 상태인 객체 상에 일부 조건문에 대해 메소드의 종료로 확인될 수 있다. 허용 집합 대신 가드 메소드를 채택하면 메시지가 주어진 상태에 허용되는지의 여부를 판단할 수 있다.

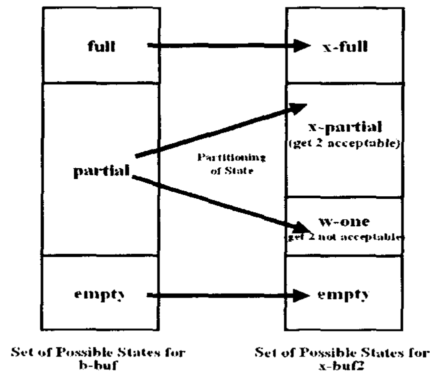

그림 1 . 상태분할변칙

허용 집합의 과거 민감성 변칙은 객체의 지나간 사건과 연관된 실행 메소드에 적용되는 것으로, 역사 정보(history information) 기반의 동기화를 명세할 수가 없

기 때문에 발생한다. 객체의 과거의 상태 정보를 구별하기 위해서 상태의 재정의가 요구되고, 객체의 상태를 재정의하기 위해서 부모 클래스의 메소드의 변경이 이루어져야 하며, 그 결과, 객체의 상태는 내부의 구별이 가능한 상태에 대하여 과거 민감성이 된다.

새로운 메소드를 서브 클래스에 추가하면 슈퍼 클래스 내의 메소드 집합의 허용 가능 상태를 변경하게 된다. 이런 변경으로 인하여 현재의 역사 정보 참작이 실패하는 일이 발생하고, 이 때 허용 집합의 상태 변경 변칙이 발생한다.

Ⅳ. 상속 변칙의 해결

1. 객체의 행위와 행위 기술 방정식

실행을 허용하는 순서인 행위(behavior)의 값은 메소드의 이름 집합으로 표현되는데, 객체에 정의된 메소드 행위의 정적인 분리가 가능하고, 한 객체에 있어서 동적인 분리도 가능하다. 행위 이름 집합을 사용하여 얻을 수 있는 이점은 첫째, 프로그램의 판독성을 개선하게 된다. 둘째, 능동 객체의 동적인 실행 시간 연산 행위를 프로그래머가 이해하기 쉽게 된다. 그러나 병행 객체들의 순차 활동으로부터 분리된 병행성 제어에 의해 행위 이름을 처리하는 집중 제어 방식에서는 동기화 코드의 상속을 배제하므로 결점이 된다. 셋째, 동기화 메카니즘이 become 연산에 일치하는 프리미티브를 전혀 요구하지 않기 때문에 쉽게 구조화된다[8][9].

현재의 객체의 상태에 따라 메소드의 코드 부분의 실행 결과인 행위를 처리하는 개념을 도입한 것이 행위 기술 방정식 모델이다. 행위 기술 방정식은 객체들의 현재의 '상태' 표현이 매우 중요하며, 객체들의 행위는 객체의 상태에 따라 동작하기 때문에 현재의 상태에 종속된 실행을 먼저 허용하는 새로운 메소드 집합을 처리한다. 여러 가지의 상속 변칙을 해결하기 위해 구현하는데 경계 버퍼를 사용하였으며, 구문은 다음과 같다.

```
STATE = TERM + TERM + ………… (1)
TERM=A ACT{[PRED][PRED]…}.OBJECT STATE : …………… (2)
STATE = Based(CLASS.STATE + CLASS.STATE +…)/{-([TERM]), +([TERM])}; ……………… (3)
```

식(1)의 구문은 특수한 상태에 있는 객체들의 행위 표현, 식(2)는 객체들의 행위의 결합, 식(3)은 파생 클래스의 정의이다. 술어 TERM은 객체의 행위를 표현하는 술어이다. 주어진 메소드 실행 후에 술어 [PRED]가 참이면 다음 상태는 [OBJECT_STATE]로 결정된다. 파생 클래스에 상속될 술어인 Based() 다음에 슈퍼 클래스의 행위를 가리키는 술어는 Based이며, -([TERM])과 +([TERM])은 슈퍼 클래스에서 상속된 상태로 제거 예정인 행위와, 슈퍼 클래스에 추가 예정인 상태인 새로운 행위를 표현한다.

2. buffer 클래스

buffer 클래스의 행위 기술 방정식은 객체의 다음 상태의 행위 표현 구문이다.

```
// buffer 클래스
EMPTY   = put.PARTIAL   ………… (4)
PARTIAL = put[0<number<SIZE].PARTIAL
        + put[number = SIZE].FULL
        + get[0<number<SIZE].PARTIAL
        + get[number = 0].EMPTY …… (5)
FULL    = get.PARTIAL   ………… (6)
```

객체의 현재의 상태가 구문(4)의 EMPTY이고, 입력이 put() 메소드이면 다음 상태는 get_2_buffer 클래스의 구문(7)의 ONE이 된다. 구문(5)에서 현재의 상태가 PARTIAL이므로 put() 메소드와 get() 메소드를 모두 허용한다. 구문(6)은 현재의 상태가 FULL이므로 get() 메소드만 허용한다.

BEH_CTL와 MEM을 포함하는 buffer 클래스의 의사 코드는 BEH_CTL(동기화 코드에 대응), MEM(메소드 코드에 대응), NSIZE(경계 버퍼의 크기)로 표현되며, 그림2와 같다.

```
// Pseudo code of buffer class
CLASS buffer {
    int NSIZE, number = 0 ;
BEH_CTL :
  EMPTY   = put.PARTIAL ;
  PARTIAL = put[0<number<NSIZE].PARTIAL
          + put[number=NSIZE].FULL
          + get[0<number<NSIZE].PARTIAL
          + get[number=0].EMPTY ;
  FULL    = get.PARTIAL ;
MEM :
  viod put( int item )
     { number++; … };   // store an item
  int get()
     { number--; … };   // remove an item
}
```

그림2. buffer 클래스의 의사 코드

buffer 클래스에서 상속되는 get_2_buffer 클래스 구문은 다음과 같다.

```
// get_2_buffer 클래스
EMPTY    = put.ONE            ……………… (7)
ONE    = put.PARTIAL + get.EMPTY  … (8)
  PARTIAL  = put[0<number<NSIZE].PARTIAL
           + put[number=NSIZE].FULL
           + get[0<number<NSIZE].PARTIAL
           + get[number=1].ONE
           + get_2[number=0].EMPTY
           + get_2[number=1].ONE
           +
get_2[0<number<NSIZE].PARTIAL  ……… (9)
FULL = get.PARTIAL + get_2.PARTIAL …(10)
```

구문(8)의 새로운 메소드인 get_2() 메소드는 각기 다른 시간에 두 개의 항목 획득에 사용한다. get_2_buffer 클래스를 설계했어도 하나의 항목만을 가지고 있는 버퍼를 처리할 수 있는 상태의 추가가 필요하다. 왜냐하면 이와 같은 상태에서는 get_2() 메소드의 호출이 허용되지 않기 때문이다.

3. 새로운 클래스와 nget()메소드

nget_buffer 클래스는 허용 집합의 과거 민감성 변칙을 해결하기 위해서 buffer 클래스를 확장한 새로운 클래스이다. nget_buffer 클래스에 get() 메소드와 비슷한 nget() 메소드를 새로 도입하였다. nget() 메소드는 get() 메소드가 호출된 후에 즉시 허용될 수 없다는 제약조건이 따른다.

```
// nget_buffer 클래스
 EMPTY    = put.PARTIAL       ……… (11)
 PARTIAL  = put[0<number<NSIZE].PARTIAL
          + put[number=NSIZE].FULL
          +
nget[0<number<NSIZE].PARTIAL
          + nget[number=0].EMPTY
          + get.GET …………………… (12)
FULL = get.GET + nget.PARTIAL  …… (13)
  GET     = get[number=0].EMPTY
          + get[number<NSIZE].GET
          + put.PARTIAL ……………… (14)
```

구문(11)의 EMPTY는 구문(4)와 동일하고, 구문(12)의 PARTIAL은 구문(5)의 PARTIAL에서 get.EMPTY와 get.PARTIAL를 제거한 대신 get.GET, nget.EMPTY, nget.PARTIAL를 새로 도입하였다. 구문(13)의 FULL도 구문(6)의 get.PARTIAL를 제거하고 get.GET를 도입하였다. 구문(14)에서 GET은 nget_buffer 클래스를 얻기 위해 새로 도입하였다. 그 결과 nget_buffer 클래스의 메소드 코드와 동기화 코드에서 상속 변칙이 해결되었다.

4. 다중 상속

n_buffer 클래스의 구문은 다중 상속을 위한 구문으로 다음과 같다.

```
// n_buffer 클래스
EMPTY   = put.ONE          ……………… (15)

ONE   =   put.TWO   +   get.EMPTY   +
```

nget.EMPTY ··············· (16)

TWO = put.PARTIAL + get.GET + nget.ONE + get_2.EMPTY ·················· (17)

PARTIAL = get.GET
+ put[0<number<NSIZE].PARTIAL
+ put[number=NSIZE].FULL
+ nget[2<number<NSIZE].PARTIAL
+ nget[number=2].TWO
+ get_2[number=0].EMPTY
+ get_2[number=1].ONE
+ get_2[2<number<NSIZE].PARTIAL
+ get_2[number=2].TWO ········· (24)

FULL = get.GET + get_2.PARTIAL
+ nget.PARTIAL ······ (25)

GET = put[2<number<NSIZE].PATRIAL
+ put[number=2].TWO
+ get[number=0].EMPTY
+ get[number<NSIZE].GET ······ (26)

n_buffer 클래스의 모든 메소드는 get_2_buffer 클래스와 nget_buffer 클래스에서 다중으로 상속된다. put() 메소드와 get() 메소드는 n_buffer 클래스의 슈퍼 클래스 내에서 두 가지로 표현된다. nget_buffer 클래스 내에서 메소드 코드의 다의성을 피하기 위해 메소드 선택의 기준을 결정한다.

구문(15)의 EMPTY는 슈퍼 클래스 구문(7)의 EMPTY에서 상속되었으므로 동일하나, 또 다른 슈퍼 클래스인 nget_buffer 클래스에서 상속된 것은 아니다. 구문(16)의 ONE과 구문(17)의 새로운 구문인 TWO는 다의성을 해결하기 위한 메소드 선택의 기준으로 도입하였다. 구문(18)은 get_2.TWO, get_2.PARTIAL, nget.TWO, nget.PARTIAL이 추가되었다. 구문(19)의 FULL은 get_2_buffer 클래스의 구문(10)의 FULL에서 get.PARTIAL을 제거하였고, nget_buffer 클래스의 구문(13)의 FULL에서는 모든 행위가 상속됨을 알 수 있었다. 구문(20)의 GET은 구문(14)의 GET에 put.TWO를 추가하였다. n_buffer 클래스는 병행성을 방해하는 상속 변칙을 해결하기 때문에 메소드의 재정의가 불필요할 뿐만 아니라 대부분 재사용되었다.

V. 결론

병행 객체지향 프로그래밍 언어에서 상속성과 병행성의 결합으로 인하여 충돌 현상인 상속 변칙이 생기고, 이는 분산 환경에서 공유 메소드, 캡슐화 파괴 및 코드 재사용을 방해하는 요인이 된다.

본 연구에서 도입한 행위 기술 방정식 모델은 상속변칙을 해결하기 위해 새로운 클래스를 도입하여 부모 클래스의 메소드 코드가 완전하게 상속됨으로써 재정의가 불필요하게 되었으며, 동기화 코드의 재사용도 또한 가능하였다. 병행 객체지향 프로그래밍 언어에서 발생하는 상속 변칙의 문제점을 행위 기술 방정식으로 해결하였음에도 불구하고, 앞으로 개선해야 할 점은 새로운 용어의 술어와의 독립성이며, 독립성 유지는 동기화 코드의 재사용과 병행 객체의 설계에 있어서 반드시 필요하기 때문이다.

참고문헌

[1] G. Agha, P. Wegner, A. Yonezawa : Research Directions in ConcurrentObject-Oriented Programming, Massachusetts, MIT Press., pp. 107-150, 1993.

[2] 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.

[3] L. Thomas : "Inheritance anomaly in true concurrent object-oriented languages : a proposal," Proceedings of 1994 IEEE Region 10's Ninth Annual International Conference., pp. 2 Vol. xxvii+1111, 541-

Vol. 2, 1994.
[4] C. Tomlinson, V. Singh : "Inheritance and Synchronization with Enables-Sets," OOPSLA' 89 Proceedings. , Vol. 24, pp. 103-112, 1989.
[5] L. Crinogorac, S. Rao, K. Ramamohanarao : "Inheritance anomaly. A formal treatment," FMOODS'97, Vol. 2, pp. vii+470, 319-34, 1997.
[7] 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.
[8] U. Lechner, C. Lengauer, F. Nick, M. Wirsing : "How to overcome the inheritance anomaly," ECOOP'96, LNCS 1098, 1996.
[9] 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.

저 자 소 개

오 승 재
1992. 9 - 1994. 2 조선대학교 전산기공학과(석사)
1999. 2 - 현재 순천대학교 컴퓨터과학과(박사)
현재 순천청암대학 컴퓨터정보과학과 조교수