# 상속성과 병행성에서 오는 상속변칙 문제 해결에 관한 연구

박영옥* · 문정환* · 이철승* · 홍성표* · 이호영** · 이준***
*조선대학교 컴퓨터공학과
**초당대학교 정보통신공학과
*** 조선대학교 컴퓨터공학부

# A Study on Solution of Anomaly due to Integrated of Inheritance and Concurrency

*Young-ok Park, *Jeong-hwan Moon, *Chiol-seong Lee,
*Seong-pyo Hong, **Ho-young Lee ** , Joon Lee***
*Dept. of Computer Engineering, Graduate School, Chosun University.
**Dept. of Information Communication, Chodang University,
***School of Computer Engineering, Chosun University.

## 요 약

병행 객체지향 프로그래밍 언어는 병행 프로그래밍을 위한 객체의 병행성과 객체지향 프로그래밍 언어의 중요한 장점인 상속성과 재사용성, 캡슐화를 동시에 지원하기 위한 목적을 가진 언어이다. 병행 객체지향 프로그래밍의 병행성과 객체지향 프로그래밍 언어의 특성이 통합된 여러 가지 모델이 제안되어 왔다. 병행프로그래밍과 객제지향프로그래밍 기법을 결합한 병행객체지향 언어는 병행 응용프로그램을 개발하는데 여러 가지 이점을 얻을 수 있다.

병행성과 상속성의 결합으로 인하여 객체의 재사용성을 현저하게 떨어뜨리게 되거나 서브 클래스에서 상속된 코드의 재정의를 요구하게 된다. 이렇게 병행성과 상속성을 결합할 때 두 특성 사이에서 발생하는 충돌 또는 간섭현상을 상속변칙이라고 하는데, 이 상속 변칙의 영향을 최소화하고 코드 재사용을 개선하기 위한 접근 방법에 대해 많은 연구 결과가 발표되었다. 이와 같이 상속성과 병행성 사이의 간섭 문제를 해결하기 위해서 동기화 코드와 메소드 코드로 구분하여 본 논문에서는 접근하고자 한다.

## ABSTRACT

The concepts from OOP have been integrated in a concurrency, leading to the emergence of concurrent OOP. Concurrency of concurrent OOP and various model technique of OOP language are integrated had been proposed. Concurrent programming and OOP technique unite that can gain various kinds advantage to develop concurrency application program. There have been a number of models proposed for integrating concurrency and OOP . However, concurrency and inheritance are two paradigms which are difficult to combine in a suitable manner. The inheritance anomaly is the conclicted phenomena, which occurs only when concurrency is integrated with inheritance. The inheritance anoay is referred to as the serious difficulty in integrating inheritance and concurrency in a simple and efficient manner within a concurrent OOP. Concurrency and inheritance with integrated that Drop reusability of object remarkably and require re-justice of code that is inherited in subclass. So concurrency and inheritance with integrated Collision that happen between two special qualitys or Interference phenomenon is inheritance anomaly. Effect of inheritance anomaly minimum Much study findings announced about access method to improve code reusability. Wish to approach in paper that is division by synchronization code and method code to solve interference problem between and concurrency.

## 1. 서 론

최근에 들어와서 소프트웨어의 급격한 발전으로 인하여 프로그램의 복잡하여 진 것이 프로그래밍 기법을 빠르게 변환시키는 요인이 되었다.

이러한 변화는 정확성, 확장성, 재사용성, 호환성, 유지보수성 등에 대한 어려운 문제가 대두되

었다 COOP는 객체(Object)의 병행성을 지원할 뿐만 아니라 객체지향 프로그래밍 언어의 장점인 객체의 상속성, 재사용성, 캡슐화, 등 객체지향 개념과 쓰레드, 동기화, 통신 등의 병행성 개념을 동시에 지원할 수 있는 언어이다. COOP에서는 병행성과 상속성 간에 결합으로 인하여 충돌하는 특성과 상속변칙이 발생하게 된다. 동기화가 이루어진 객체의 구조를 언어의 동기화 구조라고 하는데, 상속변칙은 언어의 동기화 구조에 달려있다. 상속변칙의 발생 원인과 유형을 파악하기가 쉽지 않기 때문에 이를 해결하는 데 많은 어려움이 따른게 된다.

병행 객체의 동기화와 상속성 사이의 충돌로 인하여 발생하는 상속변칙 문제를 해결하는 방법에는 메소드 방식과 허용집합 개념을 기반으로 한 접근 방식이 있는데 본 논문에서는 허용 집합 내에서 객체 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 이용하여 여러 가지 상속 변칙의 문제를 해결하고자 한다.

## II. 병행성 객체지향 언어

현재 각광 받고 있는 프로그래밍 기법인 객체지향 기법에서는 객체를 독립적으로 동작하는 모듈로 보며, 다른 객체들과의 정보 교환은 반드시 명시적인 메시지를 통해서만 이루어진다[1].

COOP는 메시지 패싱에 의해 서로 정보를 주고 받고 동시에 수행 가능한 객체를 기본으로 하는 프로그램 방법으로 객체의 병행성을 통한 효율과 모델링 능력의 극대화를 제공하고자 노력하고 있다. 프로그램에서 이러한 병행성을 고려해야 이유로 몇 가지가 있다.[1][3]

첫째, 실세계가 병행적이므로 대부분의 정보처리 시스템은 이러한 실세계를 모델화하려고 하므로, 언어 자체에 병행성이 있다면 프로그래밍 쉬워질 수 있다.

둘째, 해결하려고 하는 문제에 존재하는 병행성을 표현하는 수단이 제공되면 순차적인 프로그램 보다는 성능의 향상을 얻을 수 있으며 마지막으로 시스템의 결합에 대한 내성이 증가하게 된다.[1][3]

### 1. 상속성

상속은 OOP가 다른 프로그래밍 언어와 차별되는 주요개념이다. 어떤 한 클래스가 다른 클래스의 속성을 물려받는 것을 상속이라 한다.[4]

상속은 소프트웨어의 실행 원소인 객체들의 재사용을 목적으로 한다. 여기서 상속을 받는 클래스를 하위 클래스, 상속을 물려주는 클래스를 상위 클래스라 한다. 이때 클래스들간의 관계는 계층구조로 나타나는데, 하위 클래스는 상위 클래스의 속성을 물려받으며 추가로 새로운 속성을 정의 할 수 있다. 이때 상위 클래스의 구성 원소 이름이 하위 클래스의 구성 원소 이름과 같아 이름의 충돌이 발생하므로 상위 원소 이름을 다른 이름으로 변경 할 수 있고, 상위 클래스내의 한 메소드에 대해 하위 클래스에서 구현을 달리해서 재정의할 수 있으며, 상위 클래스의 속성 중 일부를 배제하여 하위 클래스에서 상속받을 수 있다.[4][6]

상속성을 기본 클래스 또는 부모 클래스에서 파생 클래스 또는 서브 클래스를 생성하면 객체간에 공통의 성질을 갖는 메소드나 데이터를 자동적으로 공유하게 되는 메카니즘이다. 이 때 부모 클래스에서 상속된 속성들과 메소드는 자식 클래스의 객체로 취급된다.[3][4]

### 2. 병행성

병행성이란 시스템 내에 존재하는 제어와 통신을 포함한 모든 독립적인 활동을 말한다. 서로 독립적인 프로세스들이 병행 수행되기 위해서는 별행 수행될 부분과 수행 시점을 프로그램에 명시하는 방법, 병행 수행될 부분의 크기 및 종류, 병행 수행되는 프로세스들 사이의 상호 작용 제어 등의 문제가 해결되어야 한다.[1][5]

병행 객체가 병행 상태이면 내부의 무결성(integrity)을 유지하기 위해서 메시지 집합의 일부를 허용하게 되는데, 이 때 허용 가능한 메시지 집합에 제한성을 둔다. 다른 객체들과 병렬적으로 수행되는 병행 객체 지향 프로그래밍 언어의 한 객체는 자신의 제어 스레드를 소유할 수 있다. 제어 스레드의 소유 상태 여부에 따라 능동객체와 수동객체로 구분한다.[2]

- 능동객체 - 객체가 생성될 때 자신의 제어 스레드를 소유하는 객체를 능동객체라고 한다. 개개의 객체들의 스레드 할당 상태를 보면 하나의 객체가 생성될 때마다 스레드가 할당되어서 객체에 전달되는 메시지를 순차적으로 수행하게 된다. 어느 한 순간에 수행 상태인 멤버 함수는 많아야 하나이기 때문에 각 멤버 함수들 사이의 동기화는 자동적으로 이루어진다. [3]
- 수동객체 - 객체가 생성 될 때 자신의 제어 스레드를 가지지 못하는 객체를 수동 객체라고 한다. 하나의 멤버 함수가 호출 될 때마다 하나의 스레드를 할당하기 때문에 하나의 객체에 호출된 여러 개의 멤버 함수를 수행하는 여러 개의 스레드가 동시에 존재할 수 있다.[[3][4]

수동 객체를 갖는 언어의 경우 다수의 스레드 사이의 동기화를 위한 프로그램 구조가 필요한데, 그 이유는 개개의 스레드가 동시에 같은 객체 내의 멤버 데이터를 참조하거나 수정할 수 있기 때문이다.

## III. 병행객체지향 프로그래밍 언어의 비교 및 분석

다중 프로세서 구조에서 COOP을 구현하려는 목적으로 추진되었던 Esprit 프로젝트에서 개발 된 POOL-T, 시스템의 모든 것이 병행적인 특성을 지닌 actor로 간주하며 이러한 Actor 모델을 토대로 하여 Agha가 제안한 Act3, 객체지향 언어는 Eiffel 언어를 병행적으로 확장한 언어로서 병행성과 상속성을 모두 지원 하는 Extended Eiffel, 순수 객체지향 프로그래밍 언어인 Smalltalk-80 언어에 병행성을 추가시켜 병행성과 상속성을 모두 지원하도록 설계된 Concurrented-Smalltalk 언어 등에서 나타나는 상속성과 병행성 결합문제를 살펴보고 각 언어에 대해 병행성의 제어 방식과 접근 방식에 관점에서 상속성과 병행성의 충돌에 관한 문제를 분석하며 그 결과는 표 1과 같다.[7][9]

| 언 어 | 병행성 제어방식 | 동기화 장치 | 문제점 |
|---|---|---|---|
| POOL-T | 집 중 제 어 | body 프로시저 | body 수정요구 |
| Act 3 | 인터페이스 제어 | become 연산 | 새로운 메소드 정의시 슈퍼 클래스 메소드 무효화 |
| Extended Eiffel | 집중제어 | Live 프로시저 | Live 프로시저 재기록 |
| Concurrent -Smalltalk | 임계구역 제어 | lock 메카니즘 | 캡슐화, 화괴, 비구조적인 구축 |
| Hybrid | 분산제어 | delay queue | 지연 큐 제어 문제 |
| Act++ | 분산제어 | lock 메카니즘 | 서브 클래스에서 상속 메소드 재사용 불가능 |
| TAO 모델 | Deontic Logic | 자율적인 body | 새로운 상속규칙 요구 |

표-1. 병행 객체지향 프로그래밍 언어의 분석.

## IV. 상속 변칙의 발생 원인과 견해

상속 변칙은 병행성과 상속성을 함께 사용할 때 병행성과 상속성 사이의 충돌로 발생하는 현상이다. 즉 동기화 코드를 보유한 클래스 선언을 재사용할 경우에 모든 타입의 변경을 지원할 수 있는 동기화 코드가 없기 때문에 상속변칙이 발생하게 되는데 병행 객체지향 프로그래밍 언어에서 여러 가지 상속 변칙의 발생 원인을 조사해 보면 body, 명시적인 메시지 수신, 경로 표현식, 직접 키 명세, 행위 추상화, 허용 집합과 가용 집합의 first-class, 카드 메소드 등의 문제가 있음을 알 수 있고, 이를 정리하면 표 2와 같다.[5][8]

| 상속변칙의 발생 원인 | 현 상 | 언어 |
|---|---|---|
| body 분 체 | 내부 메소드 허용시에 서브 클래스에서 전체 메소드 재정의 요구 | SR, Adaa, ALSP |
| 명시적인 메시지수신 | 새로운 메소드에 명시적인 수신을 허용할 때 발생 | ABC/'1, CSSA Concurrent C++ |
| 경로표현식 | 본문의 길이가 길어지면 경로 표현식의 표현에 대한 제한성을 둔다. | Protol |
| 직접키 명세 | 메시지 키를 포함한 연산이 뒤 따르는 동기화 방식을 사용할 때 발생 | SINA, OTM |
| 행위추상화 | 슈퍼클래스에서 상속되지 않은 새로운 허용집합을 서브클래스에 추가할 때 발생 | Actor 모델 |
| 허용집합과 가용집합의 first-class | 수행시 미리 first-class 연산 포함 필요. | Rosettc |
| 카드 메소드 | 카드 술어 참가에 의해 새로운 변칙 발생 | X0/R |

표 3 상속 변칙의 발생 원인.

## V. 행위 기술 방정식에 의한 상속 변칙의 해결

### 1. 행위 기술 방정식

객체는 객체의 상태에 의해 정의되고, 한 객체의 상태는 외부에서 객체의 메소드에 의해 접근하는 변수 값의 집합으로 표현된다. 메소드의 코드 부분은 객체의 상태를 판독하거나 기록하고 객체에 메시지를 전송하는 명령의 순서를 행위 기술 방정식이라 한다. 행위 기술 방정식 모델은 현재의 객체의 상태에 따라 메소드의 코드 부분의 실행 결과인 행위를 처리하는 개념이기 때문에, 객체들의 현재의 '상태' 표현이 이 모델에서는 중요하다. 행위 기술 방정식에 따라 현재의 상태에 종속된 실행을 먼저 허용하는 새로운 메소드 집합을 처리한다. 행위 기술 방정식 모델의 구현에 경계 버퍼를 사용하며, 여러 유형의 상속 변칙을 행위 기술 방정식으로 해결한다. 특히 다중 상속의 경우에 발생하는 상속 변칙 현상도 행위 기술 방정식으로 해결될 수 있음을 증명한다. 행위 기술 방정식의 STATE 구문은 식 (1), (2), (3) 과 같이 표현하였다.[9][11]

STATE = TERM + TERM +.....(1)
TERM=A_ACT{[PRED][PRED]…}.OBJ_STATE ;
.....(2)
STATE = Based(CLASS.STATE + CLASS.STATE + ……) / {-([TER,]), ([TERM])} .....(3)

여기서 식 (1)은 특별한 상태인 객체들의 행위를 표현하였다. 식(2)는 객체들의 행위의 결합을 나타내었다. 식(3)은 파생 클래스의 정의이다.
다중 상속의 경우에는 하나 이상의 슈퍼 클래스를 명세한다. [1][5]

- 식(2)에서 술어 TERM은 객체의 행위를 표현하는 술어이다.
- 식(2)에서, 주어진 메소드를 실행한 후에 술어 [PRED]가 참이면 다음 상태는 [OBJ_STATE]로 결정된다.
- 식(3)에서 지정어인 Based는 파생 클래스에 상속될 행위 표현 술어인 Based() 다음에 슈퍼 클래스의 행위를 가리키는 술어이다.
- -([TERM])은 슈퍼 클래스에서 상속된 상태로, 제거 예정인 행위를 표현한다.
- +([TERM])은 슈퍼 클래스에 추가 예정인 상태에 속하는 새로운 행위를 표현한다.

2. 경계 버퍼에서 get_2() 메소드 설계

2-1. buffer 클래스

객체의 다음 상태의 행위 표현인 buffer 클래스의 행위 기술 방적식이다.[1][8] 경계 버퍼를 사용한 구현은 C++ 언어의 구문을 사용하였다. buffer 클래스의 표현은 식 (4), (5), (6)과 같다.

```
//클래스 buffer 행위 기술
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)
```

위의 buffer 클래스의 식 (4)에서 객체의 현재의 상태가 EMPTY이고, 입력이 put() 메소드이면 다음 상태는 식(7)의 ONE이 된다.[7]

식(5)에서 현재의 상태가 PARTIAL이므로 put() 메소드와 get() 메소드를 모두 허용한다.

식(6)은 현재의 상태가 FULL이므로 get() 메소드만 허용한다.

* BEHAVIOR_CONTROLLER :동기화 코드에 대응한다.
* MEMBERS : 메소드 코드에 대응한다.
* SIZE : 지역 변수이고, 경계 버퍼의 크기이다.

2-2. get_2() 메소드 설계

buffer 클래스와 비교하기 위해서 buffer 클래스에서 상속되는 새로운 클래스인 get_2_buffer 클래스를 설계한다. get_2_buffer 클래스 설계는 식 (7), (8), (9), (10) 과 같다.

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

get_2_buffer 클래스에서 새로 추가되는 get_2() 메소드는 각각 다른 시간에 두개의 항목을 얻는데 사용하며, 식(8)과 같다. 식(9)에 새로 추가된 메소드인 get_2() 메소드의 나머지 행위 구분이 buffer 클래스의 식(5)에서는 이루어지지 않음을 알수 있다. 따라서 buffer 클래스에서 상속이 이루어지는 새로운 get_2_buffer 클래스 설계가 필요하다. get_2_buffer 클래스를 도입해도 단 하나의 항목을 가지고 있는 버퍼를 처리할 수 있는 추가 상태가 필요하다. 이와 같은 상태에서 get_2() 메소드의 호출은 허용되지 않는다.

결과적으로, get_2_buffer 클래스의 STATE 구문 PARTIALDMS을 제거할 수 있는 다른 술어 표현이 필요하다. get_2_buffer 클래스를 얻기 위해 행위 표현 술어 put.ONE을 식(4)에 추가해야 하기 때문에 get_2_buffer 클래스의 새로운 행위 기술 방정식인 EMPTY를 도입하였다.

그 결과, 메소드 코드는 슈퍼 클래스에서 상속되기 때문에 재정의는 불필요하게되었다. 또한 행위 기술 방정식의 사용으로 동기화 코드의 재사용이 가능하였다. 동일한 동기화 코드를 포함하는 두 객체는 동일하게 관찰되는 외부 행위이며, 이 두 행위의 차이점은 상대방 객체의 행위의 정도에 다르게 종속되는 각자의 동기화 코드를 포함한다는 점이다.[1][13]

2-3. naget() 메소드를 사용한 과거 민감성 변칙 해결

허용 상태의 문제에 있어서 과거 민감성의 상속 변칙을 해결하는 수단으로 행위 기술 방정식을 사용해야 한다. 따라서 buffer 클래스에 새로운 병행 객체 naget_buffer 클래스를 설계하였으며, naget_buffer 클래스의 행위는 식 (11), (12), (13), (14)와 같이 표현하였다.[2][8]

naget_buffer 클래스 객체를 위하여 새로운 메소드인 naget() 메소드를 도입하였다. naget() 메소드의 행위는 get() 메소드의 행위와 비슷하나, get() 메소드가 호출된 후에 즉시 허용될 수 없는 예외 조건을 포함한다.

```
//naget_buffer 클래스 행위 기술
EMPTY   =put.PARTIAL.....(11)
```

```
PARTIAL  =put.[0<number<SIZE].PARTIAL
         +put[number=SIZE].PARTIAL
         +put[0<number<SIZE].PARTIAL
         +naget[number=0].EMPTY
         +get.GET
FULL     =get.GET + nagetPARTIAL.....(13)
GET      =get[number=0].EMPTY
         +get[number<SIZE].GET
          put.PARTIAL.....(14)
```

naget_buffer 클래스의 행우 기술 방정식에서, 식(11)의 STATE 구문인 EMPTY는 bufer 클래스의 식(4)의 STATE 구문인 EMPTLY와 동일하게 표현하였다.

naget_buffer 클래스의 식(12)의 STATE 구문인 PARTIAL은 식(5)의 STATE 구문인 PARTIAL에서 get.GET, naget.PARTIAL를 도입하였다.

식(13)의 STATE 구문인 GET은 naget_buffer 클래스의 의사 코드 표현은 그림 과 같다.

naget_buffer 클래스의 행위 기술 방정식에서 메소드 코드 동기화 코드에서 상속 변칙이 일어나지 않았음을 알 수 있다. 또한 상속된 메소드 중 어느 것도 재정의를 요구하지 않았음을 알 수 있다. 또한 상속된 메소드 중 어느 것도 재정의를 요구하지 않았다. 그러나 동기화 코드에서 더 많은 행위 표현 술어를 제거하고, 일부 행위 표현 술어는 새로 도입해야 한다.

그 이유는 buffer 클래스에서 naget_bufer 클래스의 행위 변경이 거의 없기 때문이다. 그러나 동기화 코드 대부분이 분리되어 재상용되었고, buffer 클래스에서 코드의 상당 부분이 재사용되었다.

## VI. 결 론

병행 객체지향 프로그래밍의 목적은 객체들의 병행성과 상속성을 통한 처리 능력의 향상, 강력하고 유연한 소프트웨어 설계, 동적 설정 획득 등이다. 그러나 OOP에서 상속성과 병행성을 결합할 때 발생하는 상속 변칙은 분산 환경에서 공유 메소드의 어려움, 상속 변칙으로 인한 캡슐화 파괴 및 코드 재사용을 저해하는 요소가 된다. 본 연구에서는 허용 집합을 토대로 상태 분할 변칙, 허용 상태의 과거 민감성 에 대하여 상속 변칙을 해결하고자 행위 기술 방정식 모델을 설계하였다. 향후 여구 방향은 새로운 용어가 술어와 독립적으로 표현된 것처럼 행위늬 나머지 부분을 포함하는 술어들 사이의 종속성을 분리하고자 한다.

## 참고문헌


[1] 신동욱, 지동해 : 객체 기반 병행 언어 뮤치을 위한 분산 객체 관리기의 설계 및 구현

[2] 이준경 : 병행 객체의 상속 변칙을 재거한 가드 조건부 동기화의 확장

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

[4] 이준경 et al., "객체지향 CHILL을 위한 타입 확장에 관한 연구," 93'춘계 정보과학 학술발표회, Vol.20, No.1, pp.289-292, 1993.

[5] 김상훈 et al., "프로세스와 객체의 단일화를 이용한 병행 객체 언어의 설계 및 구현," 한국정보과학회 논문지, Vol.22, No.1, 1995

[6] G. Agha, ACTORS: A Model of Concurrent Computation in Distributed Systems, The MIT Press, Cambridge, Massachusetts, London, England, 1986.

[7] G. R. Andrew, Concurrent Programming : Principles and Practice, The Benjamin /Cummings Publishing Company, Inc., Redwood City, California, 1991.

[8] A. Black, J. Palsberg, "Foundations of Object-Oriented Languages," ACM SIGPLAN Notices, Vol.29, No.3, pp.3-11, March 1994.

[9] P. A. Buhr et al., "C++ : Concurrency in the Object-oriented Language C++," Software-Practice and Experience, Vol.22, No.2, pp.137-172, February 1992.

[10] L. Cardelli et al., "Modula-3 Language Definition," ACM SIGPLAN Notices, Vol.27, No.8, pp.15-42, August 1992.

[11] N. H. Choen, "Type-Extension Type Tests Can be Performed In Constant Time (Technical Correspondence)," ACM Transactions on Programming Languages and Systems, Vol.13, No.4, pp.626-630, October 1991.

[12] J. Ichbiah et al., Rationale for the Design of the Ada Programming Language, Cambridge Univ. Press, 1991.

[13] J.K.Lee et al., "An Efficient Implementation of Type-Test and Type-Guard for an Object-Oriented Switching System," Proceedings of INFOCOM'93, Publ: Tata McGraw-Hill, pp.148-155, November 1993.