병행 객체지향 언어에서 상속 인터페이스 메커니즘을 이용한 상속 변칙의 해결

김국세, 이 광, 방극인, 박영옥, 이 준

조선대학교 컴퓨터공학과

Solving Inheritance Anomaly using Inheritance Interface Mechanism in Concurrent Object-Oriented Programming Languages.

Kuk-se Kim, Gwang Lee, Keug-in Bang, Young-ok Park, Joon Lee

Dept. of Computer Engineering, Chosun Univ.

E-mail : hamster@sslab1.chosun.ac.kr

요 약

상속과 병행성은 병행 객체지향 언어에서 가장 주된 개념이며 특히 코드의 재사용에 있어서 매우 중요하다. 이들은 객체의 병행 수행을 통해 최대의 계산력과 모델링 능력을 제공한다. 하지만, 병행 객체와 상속은 서로 상충되는 특성을 가지고 있으며, 이들의 동시 사용은 병행 객체들의 무결성을 유지하기 위해 상속된 메소드들의 코드 재정의를 요구하는 상속 변칙 문제를 발생시킨다. 본 논문에서는 상속 이상의 해결을 위해 캡슐화된 객체의 내부 상태들이 객체의 외부 인터페이스의 한 부분으로 이용될 수 있는 상태 추상화 개념을 도입하였다. 또한, 메소드들을 효과적으로 상속할 수 있는 상속 인터페이스 메커니즘을 설계하였고 이를 통해 전형적인 상속 이상을 해결하였다.

ABSTRACT

Inheritance and concurrency are the primary feature of object oriented languages, and are especially important for code re-use. They provide maximum computational power and modeling power through concurrency of objects. But, concurrent objects and inheritance have conflicting characteristics, thereby simultaneously use of them causes the problem, so called inheritance anomaly, which requires code redefinition of inherited methods to maintain integrity of objects. In this paper, to solve the inheritance anomaly problems we introduce concept of state abstraction, in which internal states of encapsulated objects are made available from a part of object's external interface. And we design inheritance interface mechanisms which methods are inherited efficiently. In our scheme, we can solve the *typical inheritance anomaly* problems.

Ⅰ. 서론

프로세서 기술 발전의 결과 여러 프로세서들의 유기적인 구성을 통해 병행 실행 환경이 이루어졌다. 이에 따라 병행성을 지원하기 위한 프로그래밍 기법의 요구 역시 증대되었다. 또한 실세계를 보다 정확하고 효과적으로 표현하기 위한 객체지향 모델의 개념이 등장하였고 유기적인 프로세서의 병행 실행 환경을 지원하기 위해 객체지향의 개념과 병행성의 개념을 결합한 병행 객체지향 프로그래밍이 요구되었다. 객체 지향 프로그래밍은 상속과 캡슐화를 통해 재사용성을 증대시켜 프로그래밍의 전체적인 성능을 향상시키는데 목적이 있다. 하지만 병행 프로그래밍에 객체지향의 개념을 도입할 때 병행성과 상속성의 충돌을 유발하는 상속 변칙이라는 문제를 발생한다. 상속 변칙은 상속 계층(inheritance hierarchy)내의 한 클래스에 대한 메소드 변경이 상속 계층에 있는 모든 클래스의 메소드 재정의를 요구하게 되어 캡슐화의 손상을 초래하며 그 결과 재사용을 불가능하게 한다.

본 논문에서는 상속 이상의 해결을 위해 캡슐화된 객체의 내부 상태들이 객체의 외부 인터페이스의 한 부분으로 이용될 수 있는 상태 추상화 개념을 도입하였다. 또한, 메소드들을 효과적으로

상속할 수 있는 상속 인터페이스 메커니즘을 설계하였고 이를 통해 전형적인 상속 이상을 해결하였다.

II. 병행 객체지향 언어와 상속 변칙

1. 병행 객체지향 프로그래밍의 특성

병행 객체지향 프로그래밍 기법은 병행 프로그래밍과 객체 지향의 개념을 결합한 기법으로 병행적으로 존재하는 다수의 객체들 사이의 병행 실행을 허용하거나, 한 객체 내부에서 다수의 스레드가 병행으로 실행되도록 허용함으로써 객체들의 병행적인 처리를 통해 처리기(processor)의 처리 능력을 증가시킨다[1]. 병행 객체 지향 언어에 객체 지향의 개념을 도입함으로써 객체지향 프로그래밍 언어에서 사용되는 캡슐화(encapsulation)가 한 객체 내의 참조를 최대한 특수화하여 객체들 사이의 통신량을 감소시킬 수 있기 때문에 객체들간의 독립적인 병행 수행 가능성을 최대화 할 수 있다.

객체지향과 병행성의 결합은 기존의 순차적인 객체지향 프로그래밍 언어를 확장하여 병행 처리하는 방법[1][2]과 병행 프로그래밍 언어의 설계에 있어서 객체지향의 개념을 도입하는 방법[3] 그리고 라이브러리를 설계하는 방법[4]으로 이룰 수 있다.

2. 상속과 병행 객체

객체지향 프로그래밍 언어에서 객체의 독립적인 활동과 다른 객체들과 메시지 전달을 통해 상호 작용을 위해 캡슐화와 상속을 사용한다.

병행 객체지향 프로그래밍 언어의 한 객체는 자신의 제어 스레드를 소유할 수 있다. 병행 객체는 소유한 제어 스레드의 상태에 따라 능동 객체와 수동 객체로 구분한다. 능동 객체는 자신의 제어 스레드를 가지는 경우로 독립적인 객체를 의미하며, 수동 객체는 자신의 제어 스레드를 가지지 못하는 경우로 요구 메시지를 수용했을 때만 활성화된다. 병행성은 능동 객체의 생성 및 능동 객체와 수동 객체의 상호 작용에 의해 이루어진다[5].

그러나, 효율적인 병행 프로그래밍을 위해 객체지향 개념을 도입하는 것은 다음과 같은 부수적인 문제들을 발생시킨다. 즉, ABCL/1 이나 POOL-T와 같은 언어에서는 병행 객체들의 상속을 지원하지 못하므로 코드의 재사용을 할 수 없다. 또한, ConcurrentSmalltalk나 Orient84/K 와 같은 언어는 병행 객체의 상속을 지원하지만, 병행성을 지원하지 못한다[6][7]. 따라서, 이와 같은 문제들은 재사용성을 배제하게 되며 프로그램의 전체적인 성능을 저하시키게 된다.

3. 병행 객체지향 언어의 상속변칙

(1) 상속 변칙의 특성

병행 객체지향 프로그래밍 언어에서는 재사용을 위해 한 객체의 동기화 제약조건(synchronization constraints)을 명시해야 한다. 이와 같이 객체의 동기화에 관련한 객체의 행위가 제어되는 부분은 동기화 메소드 코드 내에 동기화 코드(synchronization code)로 표현된다. 따라서 동기화 코드는 동기화 제약조건과 일관성을 유지해야 한다. 상속 변칙은 병행 프로그래밍에 객체 지향의 개념을 도입할 때 병행성과 상속성의 충돌로 발생한다[8]. 상속 변칙은 상속 계층(inheritance hierarchy)내의 한 클래스의 메소드 변경이 그 부모 클래스는 물론 자식 클래스의 메소드 재정의를 요구하게 된다.

(2) 상속 변칙의 분석

Matsuoka와 Yonezawa는 이 상속 변칙을 상태 분할(state partitioning), 과거 민감성(history-only sensitive), 상태 변경(state modification)으로 분류하였다[9].

상태 분할 상속 변칙은 하나의 상태가 여러 상태로 분할될 때 발생한다. 병행 객체지향 언어에서 하나의 객체는 집합으로 표현될 수 있는 상태들을 가진다. 클래스 내에 메소드를 추가하거나 하나의 상태가 분할될 때 이 변경을 모든 클래스에 적절하게 설명할 수 있어야 하므로 상속 계층에 존재하는 모든 클래스 내의 메소드들의 재정의를 요구하게 된다.

과거 민감성 상속 변칙은 역사적 정보(historical information)에 기반을 둔 동기화를 설명하지 못할 때 발생한다. 현재의 클래스 상태가 과거 상태에 의존할 경우 현재의 클래스 변수들은 과거 상태와 식별되지 못하므로 한 객체는 메소드 실행 결과를 반영될 수 있는 새로운 변수의 도입을 필요로 하게 된다. 새로운 변수의 도입은 상속 계층 내에 존재하는 클래스들의 재정의를 요구하게 된다.

상태 변경 상속 변칙은 상태들을 구분하기 위해 필요한 메소드 추가함으로써 발생한다. 특정 메소드가 부클래스 내에 혼합될 때 부모 클래스 내의 상태 집합을 변경하게 된다. 따라서, 부모 클래스의 상태를 분할하기보다는 변경하기를 요구함으로써 상속 계층 클래스 내의 모든 클래스에 대한 재정의를 요구하게 된다.

III. 추상형 상태를 이용한 상속 인터페이스 설계

(1) 추상형 상태의 개념

병행 객체 지향 언어에서 각 객체들의 상태 정보는 인스턴스 변수나 상태 변수의 집합으로 표현된다. 이 변수들은 들은 캡슐화 되어 보호되기 때문에 외부에 대해서는 은닉(hiding)된다. 그러나 객체의 내부 상태 정보가 절대적으로 필요한 상황들이 있다. 이러한 상황들을 해결하기 위한 방법으로 어떤 인스턴스 변수를 외부에서 접근 가능하도록 만드는 방법이 있다. 하지만, 이는 캡슐화의 손상을 가져온다. 다른 방법으로, 내부 상태 정보를 반영하는 polling 메소드를 첨가하는 방법이 있다. 이는 객체의 상태가 polling 메소드들의 호출과 실제 객체 접근 사이에서 변경될 수 있기 때문에 병렬 언어에는 적합하지 않다.

이러한 문제점들의 해결을 위해 상태 추상화 (state abstraction) 개념을 도입하였다. 추상형 상태 정보는 객체의 외부 인터페이스에 포함되어 외부로부터 관찰될 수 있다. 즉, 객체의 인스턴스 변수나 상태 변수의 상태를 나타내는 내부 상태가 추상형 상태로 사상됨으로써 외부에 가시적이 된다. 각각의 객체는 연관된 추상형 상태의 집합으로 선언되며 이 집합은 프로그래머에 의해 변경될 수 있다.

(2) 상속 인터페이스의 설계

병행 객체는 반드시 동기화 되어야 한다. 병행 객체가 어떤 상태에 있을 때 내부적 일관성을 유지하기 위해 동기화 제약조건을 통해 객체의 전체적인 메시지 집합에서 일부를 수용할 수 있다. 그림 1은 효율적인 상속 인터페이스 설계를 위해 조건 동기화를 가지는 경계 버퍼를 나타낸 것이다.

```
class bbuf : public actor {
  put();
    제약조건 : 0< size < MAX_SIZE
    { 입력값 증가 }
  get();
    제약조건 : size > 0
    { 출력값 증가 }
}
```

그림 1. 조건부 동기화를 가지는 경계버퍼

그림 1에서 put() 메소드는 경계 버퍼 객체 내부에 하나의 데이터를 저장하는 연산을 수행하며 버퍼가 'full'인 상태에서는 허용될 수 없다는 제약 조건을가진다. get() 메소드는 경계 버퍼 객체에서 하나의 데이터를 제거하는 연산을 수행하며 버퍼가 'empty'인 상태에서는 허용될 수 없다는 제약 조건을 가진다.

일반적으로 상속은 인스턴스 변수들의 집합과 그들의 형태를 상속하는 기능과 메소드의 구현을 상속하는 기능으로 구성된다. 또한, 상속은 새로운 클래스의 서로 다른 부분만을 기입하는 코드 재사용과 부가적인 기능을 기본 클래스로 혼합하는 코드 혼합를 위해 사용되므로 유사한 객체들의 일반적인 처리를 효율적으로 제공하고 중복되는 인터페이스의 선언을 피하기 위해 상속 인터페이스의 설계가 요구된다. 그림 2는 경계 버퍼 객체에 대한 상속 인터페이스 모듈을 나타낸 것이다.

```
class interface : bbuf{
  private:
    empty() → put()  return
    mid() → put(), get()  return
    full() → get()  return
  public:
   new method { size=0일 때 empty() 호출}
   put()
   {size<max_size에 따라 size++→아이템 증가}
   get()
   {size>0에따라 size-- → 출력값 증가}
}
```

그림 2. 경계 버퍼의 상속 인터페이스 모듈

상속 인터페이스 모듈이 가질 수 있는 추상형 상태의 집합은 {empty, full, mid}이다. 또한, 상속 인터페이스 내의 각각의 모듈은 new(), put(), get() 이라는 메소드들을 가진다. put() 메소드는 제약 조건 'size<MAX_SIZE' 상태에서 {mid}가 될 때 하나의 데이터를 저장하게 되고 경계 버퍼가 {full}이 되면 저장할 수 없다. get() 메소드는 객체가 {empty} 상태가 되면 제거할 수 없게 된다.

추상형 상태가 새로운 인터페이스 내에서 변경될 필요가 있을 때 다음과 같은 경우에만 허용된다. 첫째, 상속 인터페이스 내의 한 상태는 새로운 인터페이스 내에서 둘이나 둘 이상으로 조각난다. 둘째, 상속 인터페이스 내의 어떤 상태들은 제거될 수 있다. 그림 3은 외부에 대해 가시적인 추상형 상태의 변경의 예를 나타낸 것이다. 상태가 {low, high, full} 일 때만 get()이 수행될 수 있으며 이와 같은 객체들이 bbuf처럼 사용될 때 {high, low}는 {mid}로 간주된다.

```
class bbuf-hl : interface {
  private:
    empty() { return put()}
    low() { return get()}
    high() { return get()}
    full() { return get()}
  public:
    { low와 high } → { mid }
}
```

그림 3. 추상형 상태의 변경

추상형 상태들이 새로운 인터페이스 내에서 변경될 때 이에 대응되는 구현들이 얼마나 변경되어야 하는지 알 필요가 있다. 상태의 변경 후 상태 값들이 메소드의 body들 내에서 재정의되기 때문에 대부분의 코드들이 재정의되어야 하는 것처럼 보인다. 하지만, 새로운 상태를 동적으로 결정할 필요가 있을 때 메소드 실행이 완료된 후에 그 연산을 수행할 수 있기 때문에 코드의 재정의를 최소화 할 수 있다. 따라서 추상형 상태는 객체의 내부 상태들에 대해 필요에 따라 함수적으로 계산될 수 있다. 이 연산을 지원하는 bbuf-hl의 예에 대한 구현은 그림 4와 같다.

```
class bbuf-hl-1 : bbuf-hl {
  demon method hilow 사용
  get(){mid},  put(){mid}
  hilow{mid}→경계버퍼상태{high,low}
  제약조건 {out>=size/2}→경계버퍼상태{high}
   or 경계 버퍼 상태 {low}
}
```

그림 4. 추상형 상태 변경의 구현

bbuf-hi-1 인터페이스는 hilow()의 조건식에 따라 {mid}상태를 {high, low}로 나눈다. 그러므로, put()과 get()의 행위는 그에 따라 변경되어야 한다. 그러나 이 변경은 단지 put()과 get()이 수행을 완료하고 그 상태가 {mid}일 때만 요구된다. 이와 같은 경우 메소드 hilow()는 데몬(demon)의 역할로 활성화되고 상태는 재설정 된다.

IV. 상속 이상의 해결

1. 상태 분할 상속 변칙의 해결

상태 분할 상속 변칙은 새로운 상태가 동적으로 결정될 필요가 있을 때 메소드의 실행이 완료된 후에 연산의 수행을 행함으로써 해결될 수 있다. 그림 5는 추상형 상태의 분할을 이용한 경계 버퍼에서의 상태 분할 상속 변칙의 해결을 나타낸 것이다.

한번에 두 데이터를 삭제하는 get2() 메소드를 추가하는 경우 {mid}는 {one, mid}로 분할되어야 한다. 따라서, 추상형 상태의 집합은 {empty, one, mid, full}이 되어야 한다. get2() 메소드는 추상형 상태가 {mid, full}일 때 허용될 수 있다. {empty} 상태일 때는 get() 메소드 차체가 허용되지 않으므로 고려하지 않아도 된다. 이는 setstate()를 사용함으로서 해결할 수 있다. 즉, 그림 5에서 추상형 상태의 분할과 stestate() 메소드를 데몬 메소드로 사용함으로써 상태 분할 변칙을 해결할 수 가 있다.

```
class bbuf2 : interface{
  경계버퍼 상태 {mid}
  get2();
  {out=+2,(in = out)→{empyt or one or mid;}}
}
class bbuf2i : bbuf2 {
  get2();  v1, v2를 얻음, return
  get() 상태 {mid}, put() 상태 {mid}
}
setstate() {
  제약조건 (in=size)
  경계버퍼 {full or empty or one or mid}
};
```

그림 5. 상태 분할 변칙의 해결

2. 과거 민감성 상속 변칙의 해결

경계 버퍼에서 put() 메소드의 허용 직후에 get() 메소드를 허용할 수 없는 gget() 의 경우 가질 수 있는 추상형 상태의 집합은 {empty, mid, full, midfull, fullput} 이다. 여기서 {midfull, putfull}은 {mid} 상태와 {full} 상태를 나타낸 것이지만, 직전에 put()을 수용하여 mid와 full이 되는 상태를 나타낸다. 즉, {midput, fullput}은 직전에 put() 메소드를 허용한 과거 정보를 가진 상태이다. 따라서, gget() 메소드는 추상형 상태 집합 {mid, full}에서만 실행될 수 있다. {empty} 상태일 때는 get() 메소드 차체가 허용되지 않으므로 상관없다. 즉, put() 메소드의 실행에 있어 추상형 상태 집합이 {midput, fullput} 인 경우 put() 메소드의 실행은 가능하나 gget() 메소드의 실행은 불가능하게 함으로써 과거 민감성 상속 변칙을 해결할 수 있다. 그림 6은 경계 버퍼에서의 과거 민감성 상속 변칙의 해결을 나타낸 것이다.

```
class bbufh : interface{
  상태 {empty, mid, full}
  경계 버퍼 상태 {midput, fullput}
  gget() (in >= out + 1) {out++}
  midput, fullput=false
}
class bbufhi : bbufh {
  get() (in>=out+1) {out++;}
  when {midput,fullput} FALSE
  put() (in<max_size) {in++}
  when {midput, fullput} TRUE
  gget() (in>out+1) {out++;}
  when {midput, fullput} FALSE
  return get();
}
```

그림 6. 과거 민감성 상속 변칙의 해결

3. 상태 변경 상속 변칙의 해결

상태 변경 상속 변칙은 하위 클래스가 직교적으로 제한된 기능성을 도입할 때 발생한다. 하위 클래스 lock 인터페이스가 상위 클래스 bbuf로부터 상속할 때 lock 인터페이스에는 새로운 lock() 메소드와 free() 메소드가 추가된다. 이 중 free() 메소드의 수행 후에만 상위 메소드 put()과 get()이 수행될 수 있다. 즉, 차후의 put()과 get()의 실행 여부는 현재의 lock()와 free()의 수행에 의존한다. put()과 get()에 대한 연산들은 lock이 free일 때만 호출된다. 따라서 허용할 수 있는 상태들을 제한함으로써 put()과 get()에 대한 이전의 구현은 새로운 인터페이스를 따르게 되며 그로 인해 재사용될 수 있다. 그림 7은 경계 버퍼에서의 상태 변경 상속 변칙의 해결을 나타낸 것이다. 여기서, free()의 수행 후 put()의 수행은 경계 버퍼의 추상형 상태를 {mid, full}상태로 만들며, get()의 수행은 경계 버퍼의 추상형 상태를 {mid, empty}상태로 만든다.

```
class lock : interface{
  상태 {lock, free}
  new() → locked=0
  lock() (free) → lock=1
  free() (locked) → free=1
}
class locki : lock {
  locki();
  new() return free
  put() (in<max_size) {in++}
  get() (in>=out+1) {out++}
}
```

그림 7. 상태 변경 상속 변칙의 해결

V. 결론

본 논문에서는 상속 이상의 해결을 위해 캡슐화된 객체의 내부 상태들이 객체의 외부 인터페이스의 한 부분으로 이용될 수 있는 상태 추상화 개념을 도입하였다. 또한, 메소드들을 효과적으로 상속할 수 있는 상속 인터페이스 메커니즘을 설계하였고 이를 통해 전형적인 상속 이상을 해결하였다. 추상형 상태 분할과 데몬 메소드를 사용해 새로운 상태가 동적으로 분할될 필요가 있을 때 메소드의 실행이 완료된 후에 연산의 수행을 행함으로써 상태 분할 상속 변칙을 해결할 수 있었다. 추상형 상태의 집합이 발생해서는 안될 과거 정보를 포함하는 상태를 제한함으로써 과거 민감성 상속해결할 수 있었다. 또한, 허용 가능한 상태를 제한함으로써 이전 연산의 구현을 새로운 인터페이스의 상속을 따르게 재사용이 가능하게 함으로써 상태 변경 상속 이상을 해결하였다.

참고문헌

[1] M. Karaorman, J. Bruno, Introducing concurrency to a sequential language, Communication of the ACM Vol36 No2, pp103-116, 1993

[2] K. P. Lohr, Concurrency Annotations, OOPSLA'92 Vol.27, pp327-340, 1992

[3] P. delas H. Quiros, J. M. O. Millan, Inheritance anomaly in CORBA multithreaded environments. Theory and Practice of Object System, Vol.3. No.2. pp45-543, 1997

[4] T. Chikayama, KLIC : A portable parallel implementation of a concurrent logic programming language, Parallen Symbolic Languages and system international workshop PSLS'95, pp286-294, 1996

[5] L. R. Welch, COCOON : Areator of concurrent object-oriented Systems, Ada Letters, Vol 17 No 6, pp32-38, 1997

[6] B. Meyer, Eiffel : Programming for Reusability and Extensibility, SIGPLAN Notics, Vol 22 No 2, pp85-94, 1987

[7] D. G. Kaufra, K. H. Lee, Inheritance in Actor based concurrent object-oriented language, ECOOP'89, pp131-145, 1989

[8] C. Barry, L. Leung, P. Peter, K. Chui. Behavior equation as soution of inheritance anomaly in concurrent object-oriented languages, IEEE'96 Proceedings of PDP'96, pp360-366, 1996

[9] G. Agha, P.Wenger, A, Yonezawa 'Research direction in cncurrent object oriented programming, MIT express, pp107-150, 1993