상속변칙 해결을 위한 상태 추상화 기반 상속 인터페이스 설계

(Design of Inheritance Interface based on State Abstraction to Solve the Inheritance Anomaly)

이 광 † 이 준 ††
(Gwang Lee) (Joon Lee)

요 약 병행 객체지향언어에서 상속성과 병행성은 가장 중요한 특성이다. 하지만, 상속성과 병행성은 상충적인 특성을 가지고 있기 때문에 이들을 병행으로 사용할 경우, 객체 내부의 코드 재정의를 요구하는 상속변칙이 발생된다. 본 논문에서는 캡슐화의 손상 없이 상속변칙을 해결하기 위해 상태 추상화 개념을 도입하였다. 이를 통해 캡슐화된 객체의 내부 상태를 추상형 상태로 사상하였고, 효율적인 상속을 위해 상속 인터페이스를 설계하였다. 추상형 상태 집합을 가진 상속 인터페이스를 통해 메소드의 재정의가 발생하더라도 상속 계층 내에 존재하는 클래스들에 미치는 영향을 제거함으로써 상속변칙 문제를 해결하였다.

키워드 : 병행객체지향, 상속변칙, 동기화 제약

Abstract In concurrent object-oriented languages, inheritance and concurrency are the primary features. 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 this inheritance anomaly without broken of encapsulation, we introduce the state abstraction concept and map internal states of the encapsulated object into abstract states. And also, we design the inheritance interface for efficient inheritance. Through the inheritance interface containing abstract state set, though redefinition of the method is incurred, we can remove the influence of other classes in inheritance hierarchy. And also, we can solve the inheritance anomaly problems.

Key workds : Parallel Object-Oriented, Inheritance anomaly, Synchronization constraint

1. 서 론

병행 객체지향언어는 객체간의 참조와 통신량을 최소화하고 객체들의 병행 수행을 최대화하여 순차적 프로그램보다 높은 성능을 얻고자 하는 데 목적이 있다[1][2][3]. 하지만 병행 프로그래밍 기법과 객체지향 개념을 함께 사용할 경우 병행성과 상속성의 상충되는 특성으로 인해 상속의 장점을 살리지 못하는 상속변칙(inheritance anomaly) 문제가 발생된다[4][5]. 상속변칙은 병행 클래스 계층 구조내의 상위 클래스에서 정의된 메소드가 병행성을 위한 조건부 동기화의 확장으로 인해 재정의 없이 하위 클래스에 상속되지 못하는 현상으로 캡슐화의 손상을 초래하고 나아가 재사용을 불가능하게 한다. 본 논문은 객체의 상태에 기반을 두고 캡슐화를 손상시키지 않는 범위 내에서 객체의 내부 상태를 참조하여 효율적인 상속과 메소드 동기화를 이루어 상속변칙을 해결하고자 하였다. 이를 위해, 상태 추상화 개념 [6][7]을 도입하여 캡슐화된 객체의 내부 상태가 추상형 상태로 표현되도록 하였다. 또한, 효과적인 상속을 위해 상속 인터페이스를 설계하였고 객체의 내부 상태를 추상형 상태 집합으로 표현하여 상속 인터페이스에 포함시킴으로써 외부로부터 접근 가능하게 하였다. 따라서, 메소드의 재정의가 발생되더라도 상속 인터페이스와 추상형 상태 집합의 상속을 통해 상속 계층 내의 클래스들에 영향 미치지 않게 함으로써 상속변칙을 해

† 정 회 원 : 청주과학대학 컴퓨터과학과 교수
gwang@chongjunc.ac.kr
†† 종신회원 : 조선대학교 전자·정보공과대학 컴퓨터공학부 교수
jlee@mail.chosun.ac.kr
논문접수 : 2001년 5월 14일
심사완료 : 2002년 3월 8일

결하였다.

2. 병행 객체지향언어와 상속변칙

2.1 객체지향언어와 병행 객체지향언어

객체지향언어는 문제를 해결하기 위해 실세계를 표현하는 객체를 정의하고 객체들의 역할 및 상호 작용을 명시하여 객체 사이의 연산을 프로그래밍 한 것이다. 객체지향언어는 캡슐화와 상속성을 사용한다. 캡슐화란 서로 연관된 데이타와 메소드를 결합하여 객체들의 내부 정보를 은닉하는 것이다. 상속성이란 객체들 간의 계층 경로를 따라 부모 클래스로부터 자식 클래스로 공통된 성질을 가지는 속성과 메소드를 전달하여 기존의 모듈을 재사용할 수 있도록 하는 것이다. 객체지향언어의 가장 큰 목적은 소프트웨어의 모듈화를 통해 재사용성을 증대시키는데 있다.

병행 객체지향언어는 병행 프로그래밍 언어의 병행성과 객체지향 개념을 함께 사용하고자 하는 것으로 객체들을 병행으로 수행시키거나 객체 내부에 존재하는 다수의 쓰레드(thread)를 병행으로 실행시키는 것을 말한다. 병행 객체지향언어의 목적은 객체간의 참조와 통신량을 최소화하고 객체들의 병행 수행을 최대화하여 순차적 프로그램보다 높은 성능을 얻고자 하는 데 있다.

하지만, 병행 프로그래밍 기법과 객체지향 개념을 함께 사용할 경우 병행성과 상속성 사이의 상충성으로 인해 상속변칙 문제가 발생된다. 즉, 동기화 코드를 보유한 클래스를 하위 클래스에 상속할 경우, 동기화 코드가 상속 계층 내에서 발생되는 모든 변경을 지원할 수 없기 때문에 동기화 코드의 재정의 없이 효과적으로 상속될 수 없게 된다. Matsuoka와 Yonezawa는 상속변칙을 상태 분할(state partitioning), 과거 민감성(history-only sensitive), 상태 변경(state modification)으로 분류하였다[8][9].

2.2 상속변칙의 분류

(1) 상태 분할 상속변칙

상태 분할 상속변칙은 한 상태가 여러 상태로 분할될 때 발생된다. 자식 클래스 내에 새로운 메소드가 추가될 때, 새로운 메소드에 대한 동기화 제약은 부모 클래스에 대한 상태 분할을 요구하여 자식 클래스는 물론 상속 계층 내의 모든 클래스에 대해 재정의를 요구한다. 상태 분할 상속변칙 발생의 경계 버퍼 객체를 예는 그림 1과 같다.

경계 버퍼 객체는 empty, partial, full이라는 상태와 데이타를 추가하는 메소드 그리고 데이타를 제거하는 메소드를 가질 수 있다. 만약, 두개의 데이타를 연속으로 제거하는 메소드가 추가되는 상황을 고려한다면, mid 상태는 하나의 데이타만을 가진 one 상태와 하나 이상의 데이타를 가진 partial 상태로 분리되어야 한다. 이 새로운 메소드는 객체가 one 상태일 때 두 개의 데이타를 연속으로 제거하는 메소드가 실행될 수 없다는 동기화 제약조건의 추가를 요구하므로 상속 계층에 존재하는 객체들에 대한 메소드의 재정의를 요구한다.

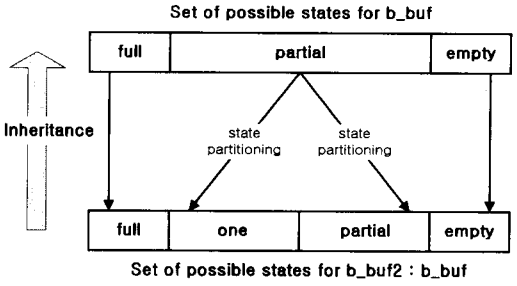

그림 1 상태 기반 상태 분할 상속변칙

(2) 과거 민감성 상속변칙

과거 민감성 상속변칙은 과거 정보(historical information)에 기반을 둔 동기화를 해결하지 못할 때 발생한다. 현재의 상태가 과거의 특정 상태에 의존할 경우 현재의 클래스 변수들로는 과거 정보를 표현할 수 없게 된다. 따라서, 한 메소드의 과거 실행 결과를 참조할 수 있는 새로운 과거 정보 변수의 도입이 필요하게 되며 이는 상속 계층 내의 클래스들의 재정의를 요구한다. 경계 버퍼에서 과거 민감성 상속변칙의 발생은 그림 2와 같다.

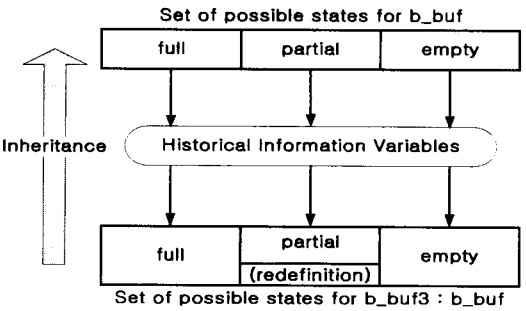

그림 2 상태 기반 과거 민감성 상속변칙

기존의 제거 메소드와 동일한 기능을 수행하지만 저장 메소드의 실행 직후에 수행될 수 없다는 제약조건을 가진다고 새로운 메소드의 도입을 가정하면, 현재 수행한

메소드의 결과를 미래에 수행할 메소드가 참조할 수 있는 과거 정보 변수가 필요하게 되며 이는 부모 클래스는 물론 상속 계층내의 모든 객체의 재정의를 요구한다.

(3) 상태 변경 상속변칙

상태 변경 상속변칙은 한 클래스에 특정 클래스가 혼합(mixed-in)될 경우, 부모 클래스 내의 상태 집합을 변경하게 되므로 상속 계층 클래스 내의 클래스에 대한 재정의를 요구함으로써 발생된다. 경계 버퍼에서 발생되는 상태 변경 상속변칙은 그림 3과 같다.

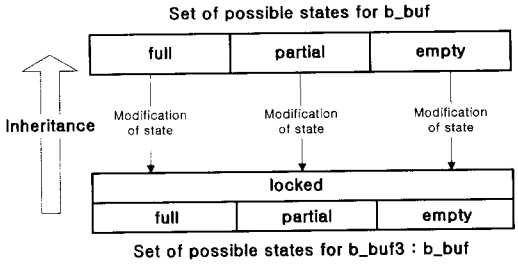

그림 3 상태 기반 상태 변경 상속변칙

객체의 잠금 기능을 수행하는 메소드와 잠금을 해제하는 메소드가 추가되는 상황을 고려할 경우, 잠금 메소드를 실행하게되면 해제 메소드가 실행되기 전까지 다른 메소드를 허용하지 않는다고 가정할 때, 과거 민감성 메소드인 잠금 메소드와 해제 메소드는 상태를 분할하기보다는 이미 정의된 동기화 제약조건을 변경함으로써 상속 계층 내의 객체의 메소드 재정의를 요구하게 된다.

3. 기존의 연구와 문제점

병행 객체지향언어에서 각 객체들의 내부 상태 정보는 인스턴스 변수나 상태 변수의 집합으로 표현되며 캡슐화되어 은닉된다. 그러나 경계버퍼에서의 put(), get() 메소드의 수행 조건처럼 객체의 내부 상태 정보가 필요한 상황들이 있다. 이를 해결하기 위해 동기화 코드의 상속을 효과적으로 허용하면서 상속변칙을 최소화하는 많은 제안들이[10][11] 이루어졌다. 기존의 연구는 객체지향언어에 병행성 위한 동기화 장치를 부여하는 것과 독자적인 언어와 컴파일러는 개발하는 것으로 분류된다.

동기화 장치를 부여하는 제안으로, 모든 메소드에 가드(guard)라는 동기화 장치를 첨가하여 각 객체가 임계영역 내에 존재하게 하는 방법이 있다[12]. 가드는 특정 메소드의 실행을 위한 부울 조건식으로 그 값이 참일 때만 해당 메소드가 활성화될 수 있다. 가드가 첨가된 경계 버퍼 객체의 예는 그림 4와 같다.

```
class b_buf {
      int in, out, size, buf[SIZE];
 public :
      void b_buf() { in = out = size = 0; }
      void put() when (in < out + size) {in++;}
      void get() when (in >= out + 1) {out++;}
}
class x_buf2 : b_buf {
 public :
      viod x_buf2()
      void get2() when (in>=out+2) {out+=2;}
}
```

그림 4 가드 메소드를 가진 파생 클래스 x_buf2

get() 메소드에 사용된 가드 'when (in>=out+1)'은 버퍼가 빈 상태에서 get() 메소드는 활성화 될 수 없다는 것을 나타낸다. b_buf의 파생 클래스인 x_buf2에 연속해서 두 개의 데이타를 연속해서 제거하는 get2() 메소드를 추가할 경우, b_buf의 put(), get() 메소드에 영향을 미치지 않으므로 상태 분할 상속변칙을 해결한다, 그러나 과거 민감성 상속변칙이나 상태 변경 상속변칙을 해결하지 못한다. 그림 5는 과거 민감성 메소드인 gget()을 포함하는 b_buf의 파생 클래스 gb_buf를 나타낸 것이다.

```
class gb_buf : b_buf {
    bool after_put;
public :
   void gb_buf() { after_put = False; }
   void gget() when (!after_put && (in>=out+1))
         { out++; after_put = False; }
   void put() when (in < out + size) {
        in++; after_put = True;
    }
    viod get() when (in >= out + 1)
         { out++; after_put = False; }
}
```

그림 5 과거 민감성 메소드를 가진 파생 클래스 gb_buf

gget() 메소드는 get() 메소드와 동일한 기능을 수행하지만, put() 메소드의 실행 직후에 수행될 수 없다는 제약조건을 가진다고 메소드이다. 따라서, 과거 상태를 반영하는 새로운 인스턴스 변수 after-put을 도입해야 하므로 상속된 put(), get() 메소드는 재정의되어야 하는 과거 민감성 상속변칙이 발생된다. 그림 6은 객체의 잠금 능력을 향상시키기 위해 Lock 클래스가 혼합된 b_buf의 파생클래스인 lb_buf를 나타낸 것이다.

```
class Lock {
  bool locked;
public :
  void Lock() { locked = False; }
  void lock() when (!locked) { lock=True; }
  void unlock() when (locked) { lock=False; }
}
```

```
class lb_bud : b_buf, Lock {
public :
    void put() when (!locked && (in<out+size))
        { in++; }
    void get() when (!locked && (in>=out+1))
        { out++; }
}
```

그림 6 클래스가 혼합된 파생클래스 lb_buf

lock() 메소드는 객체의 잠금 기능을 수행하며 unlock() 메소드는 잠금을 해제하는 기능을 수행한다. lock() 메소드는 unlock() 메소드가 실행되기 전까지 다른 메소드의 실행을 허용하지 않는다. 따라서, 현재 객체의 상태를 구분할 수 있는 인스턴스 변수 locked가 필요하게 되고 상속된 put(), get() 메소드는 재정의되어야 하는 상태 분할 상속변칙이 발생된다.

또한, 병행성과 상속성 사이의 문제점을 최소화하거나 지역화하여 상속변칙을 해결하려는 병행 객체지향언어들이 제안되어 왔다. ConcurrentSmalltalk는 Smalltalk80에 병행성을 추가시켜 병행성과 상속성을 모두 지원하도록 설계된 언어다[13]. 실행 단위인 원자 객체는 능동객체로 객체 사이에 비동기적 메시지 전송을 통해 동기화되며 병행으로 수행된다. 하지만 이 언어는 lock 메커니즘을 동기화 장치로 사용하기 때문에 과거 상태에 기반을 둔 상속변칙을 해결하지 못했으며, 메시지의 전송 시 송신객체는 수신객체의 내부상태 정보를 요구하기 때문에 캡슐화의 손상을 가져오는 단점을 가진다. Extended Eiffel은 객체지향언어인 Eiffel을 확장한 언어로 집중 제어 방식을 사용해 병행성을 제어한다[14]. 이 언어는 동기화 장치를 수퍼클래스에 정의하고 위임을 통해 동기화 코드를 각 메소드들에 동적으로 제공함으로써 상속성의 영향을 최소화하고자 하였다. 하지만, 클래스를 정의하고 위임을 하게되면 수퍼클래스는 이 위임을 반영해야 하며 최악의 경우 동기화 장치 전체가 재프로그래밍 되어야 해 캡슐화가 손상될 수 있다는 단점을 가진다.

4. 상태 추상화를 이용한 상속 인터페이스의 설계

4.1 상태 추상화

문제점들의 해결을 위해 상태 추상화(state abstraction) 개념을 도입하였다. 상태 추상화는 추상형 데이타 구조(abstract data structure)의 내부 상태 정보를 외부에서 접근 가능하도록 하는 개념이다[6][7]. 즉, 추상형 데이타 구조에 있어서 외부에서 제공되는 일련의 작동과 내부 구조의 대응을 분리하는 것과 유사하게, 상태 추상화는 외부에서 관측할 수 있는 추상형 상태와 그 객체의 내부 상태를 분리하는 것이다. 추상형 상태 정보는 객체의 외부 인터페이스에 포함되어 외부로부터 관찰될 수 있다. 각 객체마다 외부에서 관측될 수 있는 추상형 상태를 일련의 집합으로 선언하고, 객체의 내부 상태 정보와 추상형 상태와의 사상은 프로그래머에 의해 제어되게 한다. 선언되는 추상형 상태는 객체의 내부 상태를 외부에서 접근이 가능하도록 추상화한 것으로 객체의 내부 실현과는 독립적으로 존재하게 된다.

4.2 상속 인터페이스의 설계

캡슐화를 통해 외부로부터 은닉되는 객체 상태들의 내부 표현을 캡슐화가 손상되지 않는 범위 내에서 참조하여 메소드 재정의를 최소화하기 위해 상태 추상화 개념을 사용해 객체의 내부 상태를 객체 외부로의 추상형 상태로 사상될 수 있도록 하였다. 또한, 코드 재사용과 코드 혼합을 위해 상속성을 사용하고 객체들로부터의 상속을 효율적으로 처리하며 중복되는 인터페이스의 선언을 피하기 위해 추상형 상태 집합을 가진 상속 인터페이스를 구현하였다.

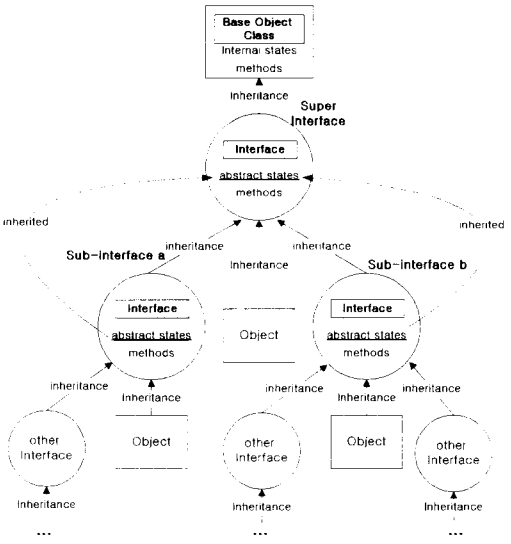

그림 7 상속 인터페이스의 상속 계층 구조

하나의 상속 인터페이스는 필요에 따라 하나 이상의 파생 인터페이스를 가질 수 있다. 상속 인터페이스가 상속될 때, 인터페이스 내의 추상형 상태 집합은 메소드와 함께 상속된다. 또한 상속을 통해 생성되는 파생 인터페이스에는 필요에 따라 새로운 메소드가 추가될 수 있다. 기저 클래스로부터 상속되는 파생 클래스는 기저 상속 인터페이스로부터 직접 상속으로 통해 생성될 수 있으며, 경우에 따라 기저 인터페이스로부터 상속된 파생 인터페이스를 통해서도 생성될 수 있다. 또한, 인터페이스를 통해 생성할 객체의 특정 실행을 위해 인터페이스에 특정 메소드를 추가하거나 기존의 메소드를 변경할 수 있고, 추가나 변경된 메소드의 동기화를 위해 상속받은 추상형 상태 집합을 변경함으로써 메소드가 실행될 수 있다. 이와 같이 하나의 기저 인터페이스나 다수 개의 파생 인터페이스를 통해 생성된 클래스들은 내부의 변경이 발생하더라도 부모 인터페이스만을 변경함으로써 다른 인터페이스 및 클래스에 미치는 영향을 제거할 수 있다. 상속 인터페이스의 상속 계층 관계는 그림 7과 같다.

5. 상속 인터페이스를 이용한 상속변칙 해결

5.1 경계 버퍼의 상속 인터페이스 구현

경계 버퍼에서 추상형 상태를 집합으로 표현하여 상속 인터페이스에 추가시킴으로써 캡슐화를 손상시키지 않은 범위 내에서 경계 버퍼 객체의 내부 정보에 접근하여 처리할 수 있게 하였다. 경계 버퍼와 추상형 상태 그리고 상속 인터페이스의 관계는 그림 8과 같다.

경계 버퍼 객체의 내부 상태는 내부 상태 표현과 관계없이 외부의 추상형 상태로 사상된다. 즉, 버퍼의 가득 찬 상태를 나타내는 내부 상태 표현과 비어 있는 상태를 나타내는 내부 상태 표현 그리고 하나 이상의 데이타를 가지는 상태를 나타내는 내부 상태 표현이 각각 full, empty 그리고 partial이라는 추상형 상태로 사상되기 때문에 추상형 상태와 내부 표현은 독립적이다. 따라서, 추상형 상태를 외부에 공개하는 것은 재사용성이나 캡슐화를 손상시키지 않는다. 이러한 추상형 상태를 집합으로 표현하여 객체에 존재하는 메소드 실행의 동기화 제약으로 사용하였다. 경계 버퍼 객체의 모듈는 그림 9와 같고, 경계 버퍼 객체에 대한 상속 인터페이스 구현 모듈은 그림 10과 같다.

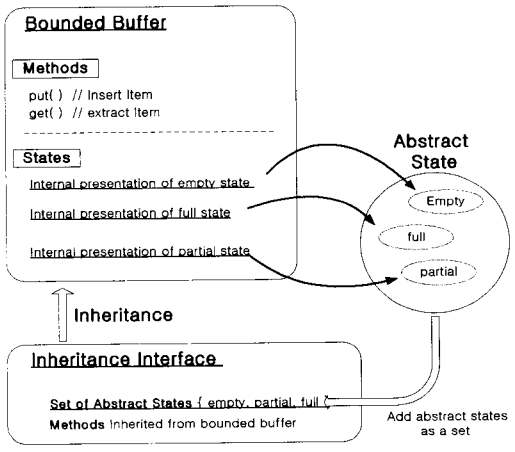

그림 8 경계 버퍼의 추상형 상태와 상속 인터페이스

```
class bbuf {
 int in, out, size, item[MAX_SIZE];
public :
 void bbuf() { in = out = size = 0; }
 void put( int x ){
   if ( size<MAX_SIZE )
   {in=(in+1)/MAX_SIZE; item[in]=x; size++ }
  int get() {
   if ( size > 0 )
   {out=(out+1)/MAX_SIZE; return item[out]; size--;}
}
```

그림 9 동기화 제약을 가지는 경계 버퍼의 구현

```
class interface : bbuf{
    int in, out, item, size, empty, mid, full;
private:
    empty() { return put(); }
    mid()   { return ( put(), get() ); }
    full()  { return get(); }
public:
    void new(int n) { if(size=0) {empty()}; }
    void put(int x) {
      if ( size<MAX_SIZE )
      {
        in=(in+1)/MAX_SIZE; item[in]=x; size++;
      }
      if(in = MAX_SIZE)
      { full(); }
      else
      { mid(); }
    }
    void get()
    {
      if ( size > 0 )
      { out=(out+1)/MAX_SIZE; item[out]; size--}
      if( size == 0)
      { empty(); }
      else
      { mid(); }
    }
}
```

그림 10 경계 버퍼에서 상속 인터페이스의 구현

각 메소드는 특정 상태에서 그 행동을 지연시키기 위한 동기화 제약을 가진다. put() 메소드는 경계 버퍼 객

체 내부에 하나의 데이타를 저장하는 연산을 수행하며, 버퍼가 full인 상태에서는 허용될 수 없다는 제약 'while (size<MAX_SIZE)'을 가진다. get() 메소드는 경계 버퍼 객체로부터 하나의 데이타를 제거하는 연산을 수행하며, 버퍼가 empty인 상태에서는 허용될 수 없다는 제약 'while (size>0)'을 가진다. 상속 인터페이스 모듈은 추상형 상태의 집합 {empty, full, partial}과 put(), get() 이라는 메소드를 가진다. put() 메소드는 bbuf()의 추상형 상태 집합이 {empty, partial} 일 때 실행 가능하다. 만약 이 조건이 만족되지 않을 경우 조건이 만족될 때까지 지연됨으로써 동기화를 수행한다. get() 메소드의 경우, 추상형 상태 집합이 {partial, full} 일 때 실행 가능하며 조건이 만족하지 않을 경우 지연됨으로써 동기화를 수행한다.

5.2 상속변칙의 해결

(1) 상태 분할 상속변칙의 해결

get2() 메소드를 수행하기 위해, 추상형 상태 {partial}은 {one, partial}로 분할되어야하므로, 추상형 상태의 집합은 {empty, one, partial, full}이 되어야 한다. 상태 분할 상속변칙은 부 인터페이스 내의 추상형 상태 집합의 분할을 이용해 해결할 수 있다. 즉, 새로운 인터페이스 내에서 객체의 상태가 동적으로 결정될 필요가 있을 때, 추상형 상태를 분할하는 기능을 수행하는 메소드를 추가하고, 그 실행을 통해 추상형 상태를 분할하여 새로 설정된 추상형 상태 집합을 메소드의 동기화 조건으로 사용함으로써 상태 분할 상속변칙을 해결할 수 있다. 추상형 상태의 분할을 이용한 경계 버퍼에서의 상태 분할 상속변칙의 해결은 그림 11과 같다.

bbuf2 클래스는 기저 상속 인터페이스인 'interface'로부터 상속된 파생 인터페이스이다. 이 인터페이스에는 get2() 메소드가 추가되어 있으며 추상형 상태 집합 {full, empty, partial, one}이 정의된다. set_state() 메소드는 추상형 상태 집합 {partial}을 {one, partial}로 분할하는 기능을 수행하는 메소드이다. bbuf2_1 클래스는 bbuf2로부터 상속된 클래스로 실제적인 get2() 연산을 수행한다. 클래스의 실행 중 버퍼 내에 데이타가 하나 남은 상태에서 get2() 메소드가 수행되면 bbuf2_1 객체는 set_state() 메소드를 호출함으로써 put() 메소드가 발생될 때까지 지연된다. put() 메소드의 실행으로 get2() 메소드의 동기화 제약이 만족되면 get2() 메소드는 두 개의 데이타를 반환한다. 따라서, 상태 분할 상속변칙은 추상형 상태를 분할하는 set_state() 메소드를 사용해 추상형 상태 집합이 {partial, full}일 때만 수행되게 하고, 그렇지 않을 경우 제약을 만족할 때까지 지연시킴으로써 해결할 수 있다.

```
class bbuf2 : interface{
    int in, out, size, item[];
public:
    void bbuf2(){in == out;}
    void get2() {
        out+=2;
        if(in = size) {full;}
        else if(size == 0) { empty;}
          else if(size == 1) { one; }
            else { partial; }
        return (return[in]);
    }
}
```

```
class bbuf2_1 : bbuf2 {
    int in, out, size, item[],v1, v2;
pubic:
    get() { partial; }
    void get2() {
      out = (out + 1) / size; v1 = item[out];
      out = (out + 1) / size; v2 = item[out];
      out = out - 2;
      bbuf( set_state() )
    put() { partial; }
    return(v1, v2);  }

void set_state(){
  if(in = size){ bbuf(full); }
     else if(in = 0){bbuf(empty);}
       else if(in = 1){bbuf(one); }
   else {bbuf(partial);}
 }
```

그림 11 상태 분할 상속변칙의 해결

(2) 과거 민감성 상속변칙의 해결

과거 민감성 메소드인 gget() 메소드의 실행을 위해서는 바로 직전에 수용한 메소드가 put() 메소드가 아니어야하므로, 과거에 수용한 메소드에 의해 재설정되는 상태를 표현하기 위해 경계 버퍼 객체의 추상형 상태 집합은{empty, partial, full}에서 {empty, partial, full, partialput}으로 변경되어야 한다. 과거 정보를 가진 추상형 상태 {partialput}은 인터페이스에 추가되는 새로운 상태로 직전에 put() 메소드를 허용하여 partial이 되는 상태를 나타낸다. 경계 버퍼에 gget() 메소드를 추가할 경우, 직전에 put() 메소드가 실행되었다는 과거 정보를 가진 추상형 상태 집합을 동기화 제약으로 사용해 gget() 메소드가 실행되게 하지 않음으로써 과거 민감성 상속변칙을 해결할 수 있다. 즉, gget() 메소드는 추상형 상태 집합 {partial}에서만 실행될 수 있어야 하므로 추상형 상태 집합이 {partialput}일 때 get() 메소드의 실행을 허용하지 않음으로써 과거 민감성 상속변칙을 해결할 수 있다. 경계 버퍼에서의 과거 민감성 상속변칙의 해결은 그림 12와 같다.

```
class bbufh : interface{
bool partialput;
  private:
          empty(){ return put()};
          mid(){return put(), get()};
          full(){return get()};
  public:
  void bbuf() { partialput=FALSE }
  int gget() {
    if ( in >= out+1 )
    { partialput = FALSE;
          return super.get(); }
  }
}
-----------------------------------
class bbufh_1 : bbufh {
public:
  void get() while(in>=out+1)
    { out++; partialput=FALSE; }
  void put() while(in<MAX_SIZE)
    { in++; partialput=TRUE; }
  gget() {
    while(!midput, !fullput(in>out+1))
     { out++; partialput=FALSE;
       return get();}
     }
}
```

그림 12 과거 민감성 상속변칙의 해결

```
class lock : interface{
protected :
  bool locked;
  public:
   void lock() { if (!locked) {locked=TRUE};}
   void unlock() { if (locked) {locked=FALSE};}
}
-----------------------------------
class locki : lock {
  public:
   void put( ) {
     while(size < MAX_SIZE)
       { if(!locked)  in++; }
   }
   int get() {
     while(size > 0)
       { if(!locked)  out++; }
   }
}
-----------------------------------
class lock2() {
public :
  void unlock( ) {
      put( ); get( ); lock();
  }
  void lock( ) {
     unlock( );
  }
}
```

그림 13 상태 변경 상속변칙의 해결

bbufh 클래스에는 추상형 상태 집합 {empty, partial, full, partialput}이 선언되며 gget() 메소드가 추가된다. gget() 메소드의 실행을 위해 추상형 상태 집합 {partialput}을 FALSE로 설정한다. bbufh_1 클래스는 부 인터페이스 bbuf로부터 상속된 클래스이다. put() 메소드의 실행은 차후에 수용될 메소드가 gget() 메소드가 아니어야 하기 때문에 추상형 상태 집합 {partialput}을 TRUE로 설정한다. gget() 메소드의 실행 자체는 get() 메소드와 동일하기 때문에 실행 후 추상형 상태 집합 {partialput}을 FALSE로 설정한다. 따라서, gget() 메소드는 추상형 상태 집합 {partialput}에서는 실행될 수 없으며, get() 메소드가 수용될 때까지 지연됨으로써 동기화를 수행한다.

(3) 상태 변경 상속변칙의 해결

상태 변경 상속변칙은 lock 인터페이스를 도입함으로써 해결할 수 있다. 즉, 메소드의 실행은 첨가되는 인터페이스의 상태에 의존하므로, 실행 가능한 상태를 제한하고 기존의 메소드들은 새로운 인터페이스를 따르게 함으로써 상태 변경 상속변칙을 해결할 수 있다. 상태 변경 상속변칙의 해결은 그림 13과 같다.

하위 클래스 lock 인터페이스가 상위 클래스 bbuf로부터 상속될 때, lock 인터페이스에는 새로운 lock() 메소드와 unlock() 메소드가 추가된다. unlock() 메소드의 수행 후에만 상위 메소드 put()과 get()이 수행될 수 있으므로, put()과 get()의 실행 여부는 현재의 lock()과 unlock()의 수행에 의존한다. 즉, locked가 TRUE 이면 실행 불가능한 상태가 되고, FALSE 이면 실행 가능 상태가 된다. 클래스 locki는 put() 메소드와 get() 메소드의 실행 과정을 나타낸 것이다. 클래스 lock2()에서 unlock()을 수행한 후에 put() 메소드와 get()메소드가 성공적으로 수행되게 하고, lock()을 수행한 후에 unlock()이 수행되게 함으로써 상태 변경 상속변칙으로 해결할 수 있다. 따라서, put()과 get()에 대한 이전의 구현은 새로운 인터페이스를 따르게 되며 그로 인해 재사용될 수 있다.

5.3 실험 결과

실험은 UNIX 운영체제 환경의 SPARC/Server에서 C++ 언어를 통해 이루어졌으며 C와 C++에서 제공하는 병행 라이브러리를 사용하였다. 각 메소드는 객체의 상태 집합 일부를 동기화 제약으로 사용하기 때문에 메소드 실행의 동기화 제약으로 상속 인터페이스가 가지는 추상형 상태 집합을 사용하였다. 상속 인터페이스를 이용해 상속받은 클래스에 상속변칙이 발생될 수 있는 상황을 설정하고, 각각의 상황에서 메소드들이 동기화되는 과정을 보임으로써 상속변칙의 해결을 검증하고자 하였다. 즉, 경계 버퍼에 존재하는 데이타들의 수를 미리 설정하고, 그 버퍼에 대해 발생하는 메소드들과 그 실행

순서를 설정하여 상속변칙을 인위적으로 발생시켰으며, 각 메소드가 실행되는 과정에서의 시간을 검출하였다. 메소드의 상대적인 시간 검출을 위해 시스템 클럭을 사용하였으며, 각 메소드의 상대적인 시간의 차이와 지연 시간을 통해 메소드가 동기화됨을 확인 할 수 있다. 다음 그림들은 세 가지 상속변칙 해결에 대한 결과를 그래프로 표현한 것이다.

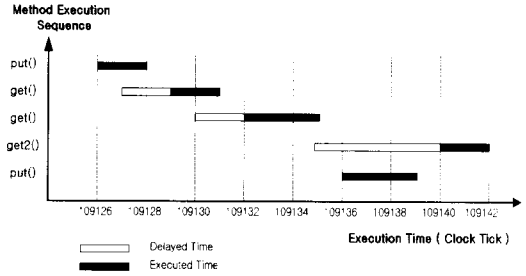

그림 14 상태 분할 상속변칙 해결 수행 결과

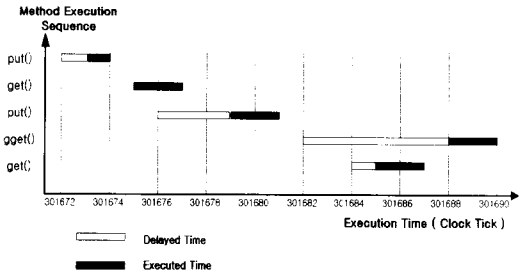

그림 15 과거 민감성 상속변칙 해결 수행 결과

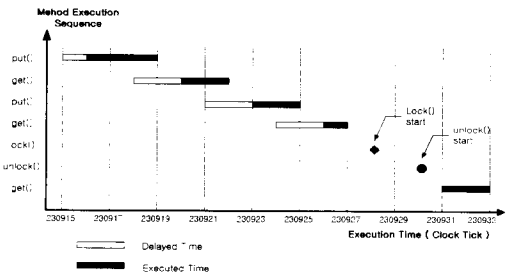

그림 16 상태 변경 상속변칙 해결 수행 결과

6. 결 론

본 논문에서는 객체의 상태에 기반을 두고 객체의 캡슐화를 손상시키지 않는 범위 내에서 객체의 내부 상태를 참조하여 효율적인 상속성과 메소드 동기화를 이루어 상속변칙을 해결하고자 하였다. 이를 위해, 상태 추상화 개념을 도입하여 캡슐화된 객체의 내부 상태들이 추상형 상태로 대응되도록 하였으며, 효과적인 상속을 위해 상속 인터페이스 메커니즘을 설계하였다. 상태 분할 상속변칙은 부 인터페이스 내의 추상형 상태 집합의 분할을 이용해 해결하였다. 또한, 직전에 특정 메소드가 실행되었다는 과거 정보를 가진 추상형 상태 집합을 동기화 제약으로 사용해 상속변칙을 발생시키는 메소드가 실행되지 않게 함으로써 과거 민감성 상속변칙을 해결하였다. 상태 변경 상속변칙에서, 메소드의 실행은 첨가되는 인터페이스의 상태에 의존하므로 실행 가능한 상태를 제한하여 기존의 메소드들은 새로운 인터페이스를 따르게 하고 이를 통해 재사용이 가능하게 함으로써 상태 변경 상속변칙을 해결하였다. 향후, 인터페이스 정의를 최소화할 수 있는 방안에 대한 이론적인 기반을 마련하도록 연구할 것이며, 상속 인터페이스 내의 추상형 상태 집합을 나타내는 멤버변수 표현에 대한 코드의 부하와 이로 인한 동기화 제약조건의 복잡성을 최소화할 수 있는 동기화 메커니즘의 구현에 대해 지속적으로 연구를 수행할 계획이다.

참 고 문 헌

[1] Lechner U. Lengauer C. Nick E. I. Wirsing M., "(Object+concurrent) and Reusability : A Proposal to Circumvent the Inheritance Anomaly," ECOOP'96 Object-Oriented Programming 10th European Conference Proceedings. pp.232-247, 1996.

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

[3] 이준, 김성근, 유재우, 송후봉., "C++의 병행성을 위한 클래스 라이브러리 구현", 한국통신학회지, 제20권, 제12호, 1995.

[4] Baquero C. Maora F., "Concurrency Annotations in C++," ACM SIGPLAN Notices, Vol.29, No.7, pp.61-67, 1994.

[5] Mitchel S. E. Wellings A. J., "Synchronization, Concurrent Object-Oriented Programming and Inheritance Anomaly," Comput. Lang., Vol.22, No.1, pp.15-26, 1996.

[6] Kuno Y. Ohki A., *p6-A State Abstraction-Based Parallel Object-Oriented Language*, Research Report 96-02, Graduate School of Systems Management, Univ. of Tsukuba, Tokyo, 1996.

[7] Kuno Y., "Solving Inheritance Anomaly Problems by State Abstraction-Based Synchronization," Proc. 1997

France-Japan Workshop on Object- Based Parallel and Distributed Computing, October 1997.
[8] Barry C. Leung L. Peter P. Chiu K., "Behaviour Equation as Solution of Inheritance Anomaly in Concurrent Object-Oriented Programming Languages," IEEE'96, Proceedings of PDP'96, pp.360-366, 1996.
[9] Agha G. Wenger P. Yonezawa A., *Research Direction in Concurrent Object-Oriented Programming*, MIT Press, pp. 107-150, 1993.
[10] Lechner U. Lengauer C. Nick F. Wirsing M., "How to Overcome the Inheritance Anomaly," ECOOP'96, LNCS 1098, 1996.
[11] Matsuoka S. Taura K., "Highly Effective and Encapsulated Re-use of Synchronization Code in Concurrent Object-Oriented Languages," OOPSLA' 93, pp.109-126, 1993.
[12] Decouchant D. et. al., "A Synchronization Mechanism for Typed Object Oriented in a Distributed System," ACM SIGPLAN workshop on Object-Based Concurrent Programming Vol.24, pp.105-107, ACM Press, 1989
[13] Yokoto Y. Tokoro M., *Concurrent Programming in ConcurrentSmalltalk*, Object-Oriented Concurrent Programming MIT Press., pp.159-198, 1987.
[14] Caromel D., "Concurrency and Reusability : From Sequential To Parallel," JOOP90, pp.34-42, 1990

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

이 광
1993년 2월 조선대학교 컴퓨터공학과(공학사). 1995년 2월 조선대학교 대학원 컴퓨터공학과(공학석사). 2000년 2월 조선대학교 대학원 컴퓨터공학과(공학박사). 1997년 3월 ~ 현재 청주과학대학 컴퓨터과학과 조교수. 관심분야는 병행객체지향언어, 병렬처리, 시스템소프트웨어