병행 객체지향 언어에서 상태 추상화를 이용한 상속 변칙의 해결

이 광[*], 이 준[**]

Solving Inheritance Anomaly using State Abstraction in Concurrent Object Oriented Programming Languages

Gwang Lee[*], Joon Lee[**]

요 약

상속성과 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 상속성은 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 변칙의 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 설계하였으며 이를 통해 전형적인 상속 변칙 문제를 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

* 청주과학대학 컴퓨터과학과
** 조선대학교 컴퓨터공학과
접수일자 : 1999년 4월 14일

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. 서 론

모든 분야에서 컴퓨터가 이용되면서 프로세서의 급격한 기술 발전과 더불어 고성능 소형 프로세스들의 유기적인 결합을 통해 강력한 기능의 병행처리 환경이 이루어졌다. 이에 따라 프로세서의 유기적인 환경에서 병행성(parallelism)을 지원하기 위한 프로그래밍 기법의 요구 역시 증대되었다. 또한, 소프트웨어의 수요가 급격히 증가되고 그 구조도 복잡해짐에 따라 실세계를 보다 정확하고 효과적으로 표현하기 위한 방법 중 하나로 인간의 사고 체계와 유사한 객체지향 모델의 개념이 등장하였다. 따라서, 효율적인 소프트웨어 메커니즘으로 유기적인 프로세서 환경을 지원하기 위해 객체지향의 개념과 병행성의 개념을 결합한 병행 객체지향 프로그래밍 기법이 요구되었다.

객체지향 프로그래밍은 상속성(inheritance)과 캡슐화(encapsulation)를 통해 재사용성(reusability)을 증대시켜 프로그래밍의 전체적인 성능을 향상시키는데 목적이 있다. 하지만 병행 프로그래밍 기법에 객체지향 개념이 도입할 때 병행성과 상속성의 충돌을 유발시키는 상속 변칙(inheritance anomaly)이라는 문제가 발생된다. 상속 변칙은 상속 계층(inheritance hierarchy)내에 존재하는 모든 클래스의 재정의를 요구하며 캡슐화의 손상을 초래하고 나아가 클래스나 메소드의 재사용을 불가능하게 한다. 본 논문에서는 상속 변칙 문제를 해결하기 위해 캡슐화된 객체의 내부 상태들이 객체에 대한 외부 인터페이스의 한 부분으로 이용될 수 있는 상태 추상화 개념을 도입하였다. 또한, 메소드들을 효과적으로 상속할 수 있는 상속 인터페이스 메커니즘을 설계하였으며 이를 통해 전형적인 상속 변칙 문제를 해결하였다.

Ⅱ. 병행 객체지향 프로그래밍 언어의 특성

1. 병행 객체지향 프로그래밍

병행 객체지향 프로그래밍 기법은 병행 프로그래밍 기법과 객체지향의 개념을 결합한 기법으로 병행적으로 존재하는 객체들 사이의 병행 실행을 허용하거나, 한 객체 내부에서 다수의 스레드들이 병행으로 실행되도록 허용하여 프로세서의 처리 능력을 증가시킨다[1]. 병행 프로그래밍 기법에 객체 지향개념을 도입함으로써 캡슐화를 통해 특정 객체의 내부에 대한 참조를 최대한 특수화하여 객체들 사이의 통신량을 감소시킬 수 있다. 따라서, 객체들간의 독립적인 병행 수행 가능성을 최대화 할 수 있다. 또한 병행성을 통해 객체 내부의 수행방식을 추상화(abstraction)함으로써 복잡한 병렬성의 문제를 최대한 완화시킬 수 있다.

객체지향 개념과 병행성의 결합은 크게 세 가지로 분류된다. 첫째, Simula 1과 같이 기존의 순차적인 객체지향 프로그래밍 언어를 확장하여 병행 처리하는 방법으로 객체의 병행적인 특성을 이용하여 병행 객체지향 프로그래밍 언어로 확장된다[1][2]. 둘째, ACTOR, POOL, Maude, ABCL/1과 같이 병행 프로그래밍 언어에 객체지향 개념을 도입하는 방법으로 이는 오류 및 유지, 보수의 영향력을 최소화하기 위해 병행 코드를 확장한다[3][4]. 셋째, 라이브러리를 설계하는 방법으로 객체지향 소프트웨어의 재사용성 및 호환성을 효과적으로 지원할 수 있게 한다[5].

2. 상속성과 병행 객체

객체지향 프로그래밍 언어에서 객체들은 독립적으로 활동하며 다른 객체들과 메시지 전달을 통해 상호 작용한다. 이를 위해 객체지향 프로그래밍 언

어는 캡슐화와 상속성을 사용한다. 캡슐화란 연관된 데이터와 프로세스를 결합시켜 캡슐로 만들어 객체들의 내부 정보를 최대한 숨기는 것이다. 상속성이란 클래스의 생성 시 클래스 계층 구조의 경로를 따라 부모 클래스로부터 자식 클래스로 공통된 성질을 갖는 정적인 속성들과 동적인 메소드를 전달하여 기존의 소프트웨어를 재사용할 수 있도록 하는 메커니즘이다[6].

독립적인 객체들이 병행 수행되기 위해서는 병행 수행될 부분과 수행 시점을 프로그램에 명시하는 방법이나 병행 수행될 부분의 크기 및 종류 또는 병행 수행되는 프로세스들 사이의 상호 작용 제어 등의 문제가 명시되어야 한다. 병행 객체지향 프로그래밍에서 병행 객체는 제어 스레드(control thread)의 소유 여부에 따라 능동 객체(active object)와 수동 객체(passive object)로 구분된다. 능동 객체는 자신의 제어 스레드를 가지며 다른 객체에 의해 영향을 받지 않는 독립적인 객체다. 수동 객체는 자신의 제어 스레드를 가지지 못하며 다른 객체로부터 요구 메시지를 수용했을 때만 활성화된다. 병행성은 능동 객체의 생성 및 능동 객체와 수동 객체의 상호 작용에 의해 이루어진다[7].

병행 프로그래밍 기법에 객체지향의 개념을 도입할 때 클래스는 순차 클래스, 모니터 클래스, 병행 클래스로 분류된다[7][8]. 순차 클래스는 객체의 내부에 스레드를 가지지 못하며 외부의 병행 객체들이 동시에 순차 객체의 메소드를 호출할 때 순차 객체의 일관성을 보장하기 위한 상호배제 동기화를 제공하지 못한다. 모니터 클래스는 객체 내부에 병행 스레드를 가지지는 못하지만 외부 병행 객체들이 동시에 메소드를 호출할 경우 객체의 일관성을 보장하기 위해 세마포어를 사용한다. 병행 클래스는 모니터 클래스와 비슷한 구조를 가지지만 동기화를 위해 내부 메시지 큐의 순차적인 메시지 선택을 통해 상호 배제 동기화를 제공한다.

그러나, 효율적인 병행 프로그래밍을 위해 객체지향 개념을 도입하는 것은 다음과 같은 부수적인 문제들을 발생시킨다. 즉, ABCL/1이나 POOL-T와 같은 언어는 병행 객체들의 상속성을 지원하지 못하므로 코드의 재사용이 불가능하다. 또한, ConcurrentSmalltalk나 Orient84/K와 같은 언어는 상속성을 지원하지만, 병행성을 지원하지는 못한다[6][9][10]. 따라서, 이와 같은 문제들은 병행 객체지향 언어가 가질 수 있는 장점들을 활용할 수 없게 하며 그로 인해 프로그램의 전체적인 성능을 저하시키게 된다.

Ⅲ. 병행 객체지향 언어의 상속 변칙

1. 상속 변칙의 특성

병행 객체지향 프로그래밍 언어에서는 재사용과 병행 수행을 위해 객체의 동기화 제약조건(synchronization constraint)을 명시해야 한다. 예를 들면 경계 버퍼(bounded buffer)의 경우 'full'인 상태에서 버퍼에 대한 저장과 'empty'인 상태에서 버퍼로부터의 제거는 허용되지 않는다는 동기화 제약조건이 명시되어야 한다. 이와 같이 객체의 동기화에 관련한 객체의 행위가 제어되는 부분은 동기화 메소드 코드 내에 동기화 코드(synchronization code)로서 표현된다. 따라서 동기화 코드는 동기화 제약조건과 일관성을 유지해야 한다.

상속 변칙은 병행 프로그래밍에 객체 지향의 개념을 도입할 때 병행성과 상속성의 충돌로 발생한다[11]. 즉, 동기화 클래스를 보유한 클래스 선언을 재사용할 때 재사용에 따른 여러 가지 형태의 변화를 지원할 동기화 코드가 없기 때문에 발생된다. 상속 변칙은 상속 계층에 존재하는 한 클래스의 내부 메소드 변경이 그 부모 클래스는 물론 자식 클래스의 내부 메소드에 대한 재정의를 요구한다. 그러므로, 병행성과 상속성의 혼합은 객체지향의 중요한 개념 중의 하나인 캡슐화의 손상을 초래하며 이는 클래스나 메소드의 재사용을 불가능하게 한다.

2. 상속 변칙의 분석

Matsuoka와 Yonezawa는 허용할 수 있는 메시지 집합에 대한 객체 상태의 변화로 부클래스(subclass)의 정의에 종속되어 각각 서로 다른 형태로 발생하는 상속 변칙을 상태 분할(state partitioning), 과거 민감성(history-only sensitive), 상태 변경(state modification)으로 분류하였다[12][13][14].

상속 변칙의 발생을 정의하기 위해 경계 버퍼

(bounded buffer)를 사용할 수 있다. 경계 버퍼에서의 연산을 위해 두 개의 메소드를 정의할 수 있다. 즉, 외부로부터 하나의 데이터를 버퍼 내부로 저장하는 연산을 수행하는 메소드와 버퍼로부터 하나의 데이터를 외부로 제거하는 메소드를 정의할 수 있다. 경계 버퍼의 상태를 살펴볼 때, 가득 찬 상태를 나타내는 'full'과 빈 상태를 나타내는 'empty', 그리고 'full'도 'empty'도 아니면서 하나 이상의 데이터를 갖는 'mid'라는 상태를 가질 수 있다. 경계 버퍼의 동기화 제약 조건을 살펴볼 때, 'full'인 상태에서 데이터를 저장하는 메소드가 허용될 수 없고, 'empty'인 상태에서 데이터를 제거하는 메소드가 허용될 수 없다는 동기화 제약조건을 가질 수 있다. 경계 버퍼에서 발생할 수 있는 상속 변칙의 예는 다음과 같다.

(1) 상태 분할 상속 변칙

상태 분할 상속 변칙은 하나의 상태가 두 개 이상의 상태로 분할될 때 발생된다. 병행 객체지향 언어에서 하나의 객체는 어떤 상태들을 가진다. 이 상태들은 집합으로 표현될 수 있으며 이 집합은 동기화 제약 조건에 따라 부분집합으로 분리될 수 있다. 클래스 내에 새로운 메소드를 추가함으로써 기존의 상태가 분할될 때 이 분할이 상속 계층 내의 모든 클래스에 대해 적절하게 설명될 수 있어야 하므로 동기화 제약 조건의 분할을 요구하게 된다.

경계 버퍼의 예에서, 하나의 데이터를 제거하는 메소드를 두 번 연속 수행하는 새로운 메소드가 부클래스에 추가될 수 있다. 이 때 'mid'상태는 두 개로 분할되어야 한다. 즉, 버퍼가 하나의 데이터를 가진 상태와 하나 이상의 데이터를 가진 상태로 분할되어야 한다. 또한, 하나의 데이터를 가진 상태에서는 두 개의 데이터를 연속으로 제거하는 추가된 새로운 메소드가 허용될 수 없다는 동기화 제약 조건이 추가되어야 한다. 따라서, 분할된 상태를 설명하기 위해 상속 계층에 존재하는 모든 클래스 내의 메소드들에 대한 재정의를 요구한다. 그림 1은 상태 분할 변칙을 통해 상태가 재정의되어야 하는 상태 분할 상속 변칙의 발생을 나타낸 것이다.

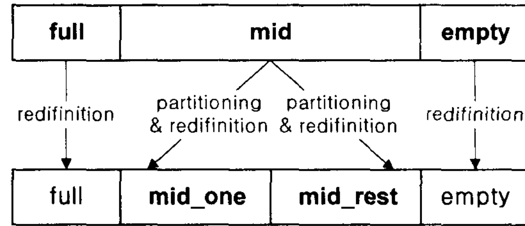

그림 1. 상태 분할 변칙 발생으로 인한 상태의 재정의

Fig. 1 State redefinition caused by state partitioning anomaly

(2) 과거 민감성 상속 변칙

과거 민감성 상속 변칙은 과거 정보(historical information)에 기반을 둔 동기화를 설명하지 못할 때 발생된다. 만약 현재의 상태가 과거의 상태나 과거의 실행에 종속된다면 이는 과거 민감성이 된다. 현재의 클래스 상태가 과거 상태에 의존할 경우 현재의 클래스 변수들은 과거 상태와 식별되지 못한다. 그러므로, 한 객체가 메소드를 실행함으로써 그 결과가 반영될 수 있는 새로운 변수의 도입이 필요하게 된다. 이 새로운 변수의 도입은 상속 계층 내에 존재하는 클래스들에 대한 재정의를 요구하는 과거 민감성 상속 변칙을 발생시킨다.

경계 버퍼의 예에서, 한 데이터를 제거하는 메소드와 동일한 연산을 수행하지만, 하나의 데이터를 저장하는 메소드의 허용 직후에는 수행될 수 없다는 조건을 가진 새로운 메소드가 부클래스에 추가될 수 있다. 따라서, 과거 정보를 반영하는 새로운 변수의 도입이 요구된다. 즉, 현재 수행한 메소드의 결과를 반영하여 미래에 수행할 메소드가 확인할 수 있는 새로운 변수가 추가되어야 한다. 과거 정보를 반영하는 변수의 추가로 인해 부모 클래스의 재정의가 요구되며 과거 민감성 상속 변칙이 발생하게 된다. 그림 2는 과거 민감성 상속 변칙의 발생으로 상태가 재정의되어야 하는 상황을 나타낸 것이다.

(3) 상태 변경 상속 변칙

상태 변경 상속 변칙은 상태들을 구분하기 위해 필요한 메소드를 추가함으로써 발생된다. 특정 메

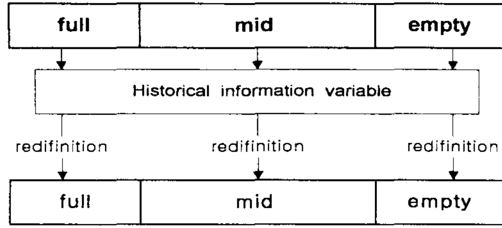

그림 2. 과거 민감성 상속 변칙 발생으로 인한 상태의 재정의

Fig. 2 State redefinition caused by history-only sensitive anomaly

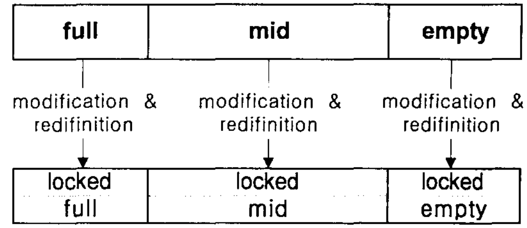

그림 3. 상태 변경 상속 변칙 발생으로 인한 상태의 재정의

Fig. 3 State redefinition caused by state modification anomaly

소드가 부클래스 내에 혼합될 때 부모 클래스 내의 상태 집합을 변경하게 된다. 따라서, 부모 클래스의 상태를 분할하기보다는 변경하기를 요구함으로써 상속 계층 내의 모든 클래스에 대한 재정의를 요구하게 된다.

경계 버퍼의 예에서, 객체의 잠금(locking) 능력을 향상시키기 위해 잠금 기능을 수행하는 lock() 메소드와 잠금을 해제하는 free() 메소드를 가진 클래스가 경계 버퍼에 혼합(mix-in)될 수 있다. 한 객체가 lock() 메소드를 허용하게 되면 free() 메소드가 허용될 때까지 다른 메시지의 수용을 모두 중지하게 된다. 과거 민감성 메소드인 lock()과 free() 메소드는 상태를 분할하기보다는 이미 정의된 동기화 제약 조건을 변경하는 것을 의미한다. 따라서, 상태 변경 변칙이 발생하게 된다. 그림 3은 상태 변경 상속 변칙의 발생으로 인한 상태의 재정의를 나타낸 것이다.

3. 상속 변칙 해결에 대한 기존의 연구와 문제점

임의의 상황에서 동기화 코드의 상속을 허용할 수 있고, 상속 변칙 문제를 최소화하기 위한 연구가 진행되어 왔다[11][12].

첫째, 재정의 요구 코드들을 최소화하고 상속성의 결합을 위해 ABCL/1 언어를 확장하는 연구가 수행되었다. ABCL/1 언어에서 메소드들은 primary, constraint, transition으로 표현된다. 객체들의 상태 천이를 캡슐화하여 과거 민감성 상태를 포함하게 되는 재정의를 지역화하고 메소드 정의의 다른 부분에서 동기화 코드를 분리해 냄으로써 재정의 요구를 감소시키고자 하였다. 하지만, 이 방법의 문제점은 다중 메소드가 재정의로 인해 영향을 받을 수 있다는 점이다. 또한 미흡한 캡슐화로 인해 추상형 동기화 상태에서 객체의 메소드 집합에 대한 연산을 수행할 수 없다는 문제점을 가진다.

둘째, 저수준의 프리미티브 관점에서 구현된 라이브러리 루틴의 집합으로 동기화 방식을 제공하는 Eiffel II에 기초를 둔 연구가 이루어졌다. 병행 객체는 추상 프로세스 클래스에서 상속을 통해 동기화 코드를 각 메소드에 연결한다. 하지만, 자식 객체에 대한 동기화 루틴에서는 이 연결을 무시함으로써 객체 중심의 동기화 관점에서 상속의 영향을 지역화하고자 하였다. 이 연구는 서로 다른 동기화 제약 조건의 요구에 대해 서로 다른 동기화 방식을 적용함으로써 병행 객체의 코드 재사용을 증진시킬 수 있게 하였다. 하지만, 클래스 정의에 의해 사용자가 위임하게 되면 수퍼 클래스는 프로그램 내에서 이 위임을 가정해야 하기 때문에 캡슐화를 손상시킨다는 문제점을 가진다. 또한 동기화 방식은 프리미티브로서 내장되어 있지 않고 사용자 수준의 코드를 통합하기 위한 어떠한 최적화도 수행되지 않기 때문에 사용자 정의 코드의 실행에 있어서는 오버헤드를 발생시킨다.

셋째, 병행 시스템의 모델링을 위해서 재기록 논리(rewriting rule)를 토대로 한 Maude 언어에 대한 연구가 이루어졌다. 이 언어로 구축한 병행 시스템은 함수적 모듈과 시스템 모듈로 구분되는 모듈 연산의 개념에서 유도되었으며, 술어와 재기록 규칙으로 구성된다. 또한, 객체지향 모듈이 시스템

모듈로 변환된 후 변환된 모듈의 재기록 규칙에 따라 병행 재기록 수행과 연산이 동시에 이루어지게 함으로써 객체 지향 모듈을 병행 객체 지향 모듈로 전환할 수 있게 하였다. 재기록 규칙에 포함된 부가 조건이 가드처럼 동작하기 때문에 상태 분할 변칙이 발생하지 않는다. 하지만, 이 언어의 특성상 강력한 패턴 매칭 능력에 기반을 두기 때문에 실제 응용에 있어서는 비효율적라는 문제점을 가진다.

Ⅳ. 추상형 상태를 이용한 상속 인터페이스 설계

1. 추상형 상태의 특성

병행 객체지향 언어에서 각 객체들의 상태 정보는 인스턴스 변수나 상태 변수의 집합으로 표현된다. 이 변수들은 캡슐화되어 보호되기 때문에 외부로부터 은닉(hiding)된다. 그러나 객체의 내부 상태 정보가 절대적으로 필요한 상황이 있다. 스택의 예를 들면, 빈 상태에서는 'pop' 되지 않아야 하고 가득 찬 상태에서는 'push'되지 않아야 하는 경우다. 이러한 상황들을 해결하기 위한 방법으로 인스턴스 변수를 외부에서 접근 가능한 형태로 만드는 경우가 있지만 이는 캡슐화의 손상을 가져온다. 다른 방법으로, 내부 상태 정보를 반영하는 폴링(polling) 메소드를 첨가하는 방법이 있다. 이는 객체의 상태가 폴링 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 데이터 타입(ADT)의 경우 상태 정보는 외부로부터 관찰되지 못하는 반면, 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 대해 가시적이 되게 하는 것이다. 각각의 객체는 연관된 추상형 상태의 집합으로 선언되며 이 집합은 외부에 대해 가시적이기 때문에 프로그래머에 의해 변경될 수 있다.

경계 버퍼 객체의 경우 추상형 상태는 { empty, mid, full }로 표현될 수 있다. 또한, 경계 버퍼 객체에 데이터가 절반 이상 존재하는 경우와 그렇지 않은 경우를 나타내기 위해 { mid }를 { midlow, *midhigh* }로 분할하여 추상형 상태의 집합을 { empty, midlow, midhigh, full }로 변경해 표현할 수 있다.

2. 상속 인터페이스의 설계

병행 객체가 어떤 상태에 있을 때 내부적 일관성을 유지하기 위해 동기화 제약조건을 통해 객체의 전체적인 메시지 집합에서 일부를 수용할 수 있다. 그림 4는 효율적인 상속 인터페이스 설계를 위해 조건부 동기화를 가지는 경계 버퍼를 나타낸 것이다.

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

그림 4. 조건부 동기화를 가지는 경계 버퍼
Fig.4. Bounded buffer with conditional synchronization

그림 4에서 put() 메소드는 경계 버퍼 객체 내부에 하나의 데이터를 저장하는 연산을 수행하며 버퍼가 'full'인 상태에서는 허용될 수 없다는 제약조건 'while(size<MAX_SIZE)'을 가진다. get() 메소드는 경계 버퍼 객체로부터 하나의 데이터를 제거하는 연산을 수행하며 버퍼가 'empty'인 상태에서는 허용될 수 없다는 제약 조건 'while(size>0)'을 가진다.

상속성은 인스턴스 변수들의 집합과 형태를 상속하는 기능과 메소드의 구현을 상속하는 기능을 가진다. 또한, 상속성은 새로운 클래스에 대해 서로 다른 부분만을 기입하는 코드 재사용과 기본 클래스에 부가적인 기능을 혼합하는 코드 혼합(code

mix-in)을 위해 사용된다. 따라서, 유사한 객체들을 효율적으로 처리하게 위해 중복되는 인터페이스의 선언을 피할 수 있는 상속 인터페이스의 설계가 요구된다. 그림 5는 경계 버퍼에 대한 상속 인터페이스 모듈을 나타낸 것이다.

```
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) {
 while(int size; size<MAX_SIZE; size++)
 {
 in = (in + 1) / MAX_SIZE;
 item[in] = x;
 }
 }
 void get()
            {
 while(int size; size > 0; size--)
 {
 out = (out + 1)/MAX_SIZE;
 return item[out];
 }
 }
 }
```

그림 5. 경계 버퍼의 상속 인터페이스 모듈
Fig. 5 Inheritance interface module of bounded buffer

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 동기화 제약 조건으로 'size < MAX_SIZE'를 가지며 추상형 상태 집합이 { empty, mid }일 때 경계 버퍼에 하나의 데이터를 저장하고 { full }일 때 저장할 수 없게 한다. 또한, get() 메소드는 동기화 제약 조건으로 'size > 0'을 가지며 추상형 상태집합이 { mid, full }일 때 경계 버퍼로부터 하나의 데이터를 제거하고 { empty }일 때는 제거할 수 없게 한다.

상속 인터페이스 내의 한 상태는 새로운 인터페이스 내에서 둘이나 둘 이상으로 분할 될 수 있다. 또한, 상속 인터페이스 내의 어떤 상태들은 새로운 인터페이스 내에서 제거될 수 있다. 이와 같은 사실에 입각해 추상형 상태는 필요에 따라 새로운 인터페이스 내에서 변경 수 있다. 그림 6은 추상형 상태의 변경의 예를 나타낸 것이다. 이 때 { mid }는 { low, high }로 분할되었으며 따라서, 전체적인 추상형 상태는 { empty, low, high, full }로 변경되었다. 이 예에서는 추상형 상태가 { low, high, full }인 경우에만 get()이 수행될 수 있다.

```
class bbuf-hl : interface {
int in, out, size, item[];
private:
empty() { return put(); }
low() {return get(); }
high() {return get(); }
full() { return get(); }
public:
void bbuf(){
if (in=out) {low(), high(); }
void get();
}
}
```

그림 6. 경계 버퍼에서 추상형 상태의 변경
Fig. 6 Abstract state modification in bounded buffer

상태의 변경 후 상태 값들이 메소드의 body 내에서 재정의되기 때문에 대부분의 코드들이 재정의되어야 하는 것처럼 보인다. 하지만, 만약 새로운 상태를 동적으로 결정할 필요가 있다면 메소드 실행이 완료된 후에 그 연산을 수행하게 할 수 있으므로 코드의 재정의를 최소화할 수 있다. 이를 지원하는 bbuf-hl에 대한 구현은 그림 7과 같다.

```
class bbuf-hl-1 : bbuf-hl {
int in, out, size, hilow, low, high ;
public :
void get( hilow() ) { mid; }
void put( hilow() ) { mid; }
int hilow() {
if(out >= size/2) { bbuf = high;}
else { bbuf = low; }
}
}
```

그림 7. 경계 버퍼에서 추상형 상태 변경의 구현
Fig. 7 Implementation of abstract state modification in bounded buffer

bbuf-hi-1 인터페이스는 hilow()의 조건식에 따라 { mid }상태를 { high, low }로 나눈다. 따라서, put()과 get()의 행위는 그에 따라 변경되어야 한다. 하지만 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 재설정된다.

V. 상속 변칙의 해결

1. 상태 분할 상속 변칙 문제의 해결

상태 분할 상속 변칙 문제의 해결은 추상형 상태 분할과 데몬 역할을 수행하는 메소드에 의해 해결될 수 있다. 즉, 새로운 상태가 동적으로 결정될 필요가 있을 때 메소드의 실행이 완료된 후에 연산을 수행하게 함으로써 해결될 수 있다. 그림 8은 추상형 상태의 분할을 이용한 경계 버퍼에서의 상태 분할 상속 변칙 문제의 해결을 나타낸 것이다.

get()을 연속해서 두 번 수행하는 get2() 메소드를 추가하는 경우, 추상형 상태 { 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 때만 허용될 수 있다. { empty }상태일 때는 동기화 제약 조건에 의해 get() 메소드 자체가 허용되지 않는다. 그림 8은 추상형 상태의 분할과 set_state() 메소드를 데몬 메소드로 사용함으로써 상태 분할 변칙 문제의 해결을 나타낸 것이다.

```
class bbuf2 : interface{
int in, out, size, item[];
public:
void bbuf(){in == out;}
void get2() {
out+=2;
if(in=out=0) { empty;}
else if(in=out+1) { one; }
else { mid; }
return (int)
}
}
class bbuf2i : bbuf2 {
int in, out, size, item[],v1, v2, b;
pubic:
void get2() {
out = (out + 1) / size; v1 = item[out];
out = (out + 1) / size; v2 = item[out];
out = out - 2;
bbuf( set_state() )
{ return(v1, v2); }
get(){mid;}
put(){mid;}
}
void set_state(){
if(in = size){ bbuf(full); }
else if(in = 0){bbuf(empty);}
else if(in = 1){bbuf(one);}
else {bbuf(mid);}
};
```

그림 8. 경계 버퍼에서 상태 분할 변칙의 해결
Fig. 8 Solve the state partitioning anomaly problem in bounded buffer

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, midput, fullput }이다. 여기서 추상형 상태 { midput }과 { fullput }은 각각 { mid }와 { full }을 나타내지만, 직전에 put() 메소드를 허용하여 { mid }와 { full }이 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() 메소드가 허용되지 않으므로 gget()의 수행과는 관계가 없다. 추상형 상태 집합이 { midput, fullput }일 때 put() 메소드의 실행은 허용하지만 get() 메소드의 실행은 허용하지 않음으로써 민감성 상속 변칙 문제를 해결할 수 있다. 그림 9는 경계 버퍼에서의 과거 민감성 상속 변칙 문제의 해결을 나타낸 것이다.

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 상속될 때 lock 인터페이스에는 새로운 lock() 메소드와 free() 메소드가 추가된다. free() 메소드의 수

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

그림 9. 경계 버퍼에서 과거 민감성 상속 변칙의 해결
Fig. 9 Solve the history-only sensitive anomaly problem in bounded buffer

행 후에만 상위 메소드 put()과 get()이 수행될 수 있으므로 put()과 get()의 실행 여부는 현재의 lock()과 free()의 수행에 의존한다. 따라서 허용할 수 있는 상태들을 제한함으로써 put()과 get()에 대한 이전의 구현은 새로운 인터페이스를 따르게 되며 그로 인해 재사용될 수 있게 된다. 그림 10은 경계 버퍼에서의 상태 변경 상속 변칙 문제의 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.

VI. 결 론

본 논문에서는 상속 변칙 문제의 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 변칙 문제를 해결하였다.

```
class lock : interface{
bool locked;
public:
 void new(){locked=0};
 void lock() { if (free) {locked}; }
 void free() { if (locked) {free}; }
}
class locki : lock {
public:
 void locki();
 void new(int n){return free()};
 void put(int item) while(!locked &&
(in<MAX_SIZE)){in++;}
 void get() while(!locked&&(in>=out+1)){out++;}
}
```

그림 10. 경계 버퍼에서 상태 변경 상속 변칙의 해결
Fig. 10 Solve the state modification anomaly problem in bounded buffer

추상형 상태 분할과 데몬 메소드를 사용해 새로운 상태가 동적으로 분할될 필요가 있을 때 메소드의 실행이 완료된 후에 연산이 수행되게 함으로써 상태 분할 상속 변칙 문제를 해결할 수 있었다. 추상형 상태의 집합이 발생해서는 안될 과거 정보를 포함하는 상태를 제한함으로써 과거 민감성 상속 변칙 문제를 해결할 수 있었다. 또한, 허용 가능한 상태를 제한함으로써 이전에서의 연산에 대한 구현을 새로운 인터페이스의 상속을 따르게 하여 재사용이 가능하게 함으로써 상태 변경 상속 변칙 문제를 해결하였다. 향후, 상속 인터페이스의 확장을 통해 추상형 상태의 변경이라는 프로그래머 수준의 오버헤드를 최소화할 수 있는 동기화 메커니즘의 구현에 대해 지속적으로 연구를 수행할 계획이다.

참고문헌

[1] M. Karaorman, J. Bruno, Introducing Concurrency to a Sequential Language, Communication of the ACM, Vol. 36, No. 2, pp.103-116, 1993
[2] K. P. Lohr, Concurrency Annotations, OOPSLA'92, Vol. 27, pp.327-340, 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, pp.45-54, 1997
[4] G. Agha, C. J. Gallsen, ActorSpace an Open Distributed Programming paradigm, 4th ACM PPOPP, pp.23-32, 1993
[5] T. Chikayama, KLIC : A Portable Parallel Implementation of a Concurrent Logic Programming Language, Parallel Symbolic Languages and system international workshop PSLS'95, pp.286-294, 1996
[6] 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
[7] L. R. Welch, COCOON : Creator of Concurrent Object-Oriented Systems, Ada Letters, Vol. 17, No. 6, pp.32-38, 1997
[8] D. Caromel, Toward a Method of Object-Oriented Concurrent Programming, Communication of the ACM, Vol. 36, No. 9, pp.90-102, 1993
[9] B. Meyer, Eiffel : Programming for Reusability and Extensibility, SIGPLAN Notices, Vol. 22, No. 2, pp.85-94, 1987
[10] D. G. Kaufra, K. H. Lee, Inheritance in Actor Based Concurrent Object-Oriented Language, ECOOP'89, pp.131-145, 1989
[11] C. Barry, L. Leung, P. Peter, K. Chui, Behavior Equation as Solution of Inheritance Anomaly in Concurrent Object-Oriented Languages, IEEE'96 Proceedings of PDP'96, pp.360-366, 1996
[12] G. Agha, P. Wenger, A. Yonezawa, Research Direction in Concurrent Object Oriented Programming, MIT express, pp.107-150, 1993
[13] U. Lechner, C. Lengauer, F. Nick, M. Wirsing, How to Overcome the Inheritance Anomaly, ECOOP'96, LNCS 1089, 1996
[14] S. Ferenczi, Guarded Methods vs. Inheritance Anomaly : Inheritance Solved by Nested Guarded Method Calls, ACM SIGPLAN Notices, Vol. 30, No. 2, pp.49-58, 1995

이 광(Gwang Lee)
1993년 2월 조선대학교 컴퓨터공학과 졸업(공학사)
1995년 2월 조선대학교 대학원 컴퓨터공학과 졸업(공학석사)
1998년 2월 조선대학교 대학원 컴퓨터공학과 박사과정 수료
1997년 2월~1999년 3월 청주과학대학 컴퓨터과학과 전임강사
1999년 4월~현재 청주과학대학 컴퓨터과학과 조교수
*주관심분야 : 병렬 객체지향 시스템, 객체지향 데이터베이스 시스템

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