프로그램 변환을 통한 Java 다중 스레드 프로그램의 결정적 테스팅

(Deterministic Testing of Java Multi-Threaded Programs through Program Transformation)

정 인 상 †

(In Sang Chung)

요 약 동시에 수행 가능한 여러 개의 스레드들로 구성된 병렬 프로그램은 본질적으로 비결정성을 내포하고 있다. 따라서, 이러한 비결정성에 기인한 비반복성때문에 순차적 프로그램을 위해 개발되었던 기존의 테스팅 테스팅 방법이나 디버깅 방법을 병렬 프로그램에 그대로 적용할 수 없다. 이 논문에서는 Java 다중 스레드 프로그램의 재수행성을 보장하기 위하여 대상이 되는 프로그램을 원시코드 수준에서 변환하는 방법을 제시한다. 일단 변환 규칙에 따라 변경이 된 프로그램은 주어진 동기화 메소드 시퀀스를 강제적으로 반복 수행할 수 있어 오류의 원인을 찾는 노력을 줄일 수 있다. 또한, 재실행하고자 하는 동기화 메소드 시퀀스가 실제 주어진 프로그램에서 수행 가능한지를 판별하는 방법을 제시한다.

Abstract Concurrent programs, which are composed of threads that can operate concurrently, are intrinsically nondeterministic. Therefore, we can not directly apply classical testing or debugging techniques developed for sequential programs to concurrent programs because of the nonreproducibility due to nondeterminism. In this paper, we present a source tranformation method to guarantee the reproducibility of multi-threaded programs written in Java programming language. Once a mutli-threaded program has been transformed according to the transformation rules, we can force the program's execution to follow the given sequence of synchronized methods repeatedly and reduce the efforts to find the source of the errors. In addition, we present a method for checking the feasibility of the synchronized method sequence to be replayed.

1. 서 론

동시에 수행 가능한 여러 개의 스레드들로 구성된 병렬 프로그램은 스레드들간의 병행성으로 시스템의 전체적인 성능을 향상시키고 주어진 자원의 활용도를 높일 수 있다. 이러한 장점 때문에 과거에 제한된 범위 내에서만 사용되었던 병렬 프로그램은 지금은 군사, 의료, 통신과 같은 분산화, 고성능, 고신뢰도를 요구하는 분야에서 널리 사용되고 있는 추세이다.그러나, 병렬 프로그램을 구성하는 스레드들의 상호작용으로 인한 복잡도로 말미암아 순차적인 프로그램보다는 이해하기 어려우며 관리하기 어렵다.

특히, 병렬 프로그램은 비결정성(nondeterminism)을 내포하고 있어 프로그램을 분석하거나, 수정, 테스팅 및 디버깅과 같은 작업을 어렵게 하는 요인이 된다. 예를 들어, 병렬 프로그램의 비결정성 때문에 프로그램의 검증이 어려운 경우를 생각해보자. 비결정성을 내포한 프로그램은 순차적인 프로그램과는 달리 동일한 입력에 대하여 반복 수행 했을 경우에 서로 다른 프로그램 경로를 선택할 수 있으며 따라서 서로 다른 결과를 산출할 수 있다. 만약, 프로그램의 오류가 있다고 판단되어 검증하려고 할 때 이러한 비결정성에 기인한 비반복성은 순차적인 프로그램의 검증을 위해 개발되었던 기존의 테스팅 방법이나 디버깅 방법을 그대로 적용할 수 없다. 왜냐하면, 프로그램 검증 시에 주어진 입력에 대하여 프로그램을 실행하였을 때 올바른 결과를 산출했

· 본 연구는 2000년도 한성대학교 교내연구비 지원과제임.
† 종신회원 : 한성대학교 정보전산학부 교수
insang@hansung.ac.kr
논문접수 : 1999년 9월 14일
심사완료 : 2000년 4월 3일

을지라도 항상 정확하게 동작한다고 말할 수 없기 때문이다. 또한, 한번 발생된 오류를 테스팅이나 디버깅을 위한 재수행에서 다시 재현시키는데 어려움이 있다.

최근 병렬처리 기술에 대한 요구에 부응하여 병렬 프로그램의 특징을 고려한 검증 기법에 관한 연구가 여러 측면에서 활발하게 진행되고 있다. 병렬 프로그램을 검증하는 방법은 정적 분석 방법[1,2], 동적 분석 방법, 디버깅 방법[3-6] 및 여러 방법들의 장점을 혼합한 방법[7]으로 분류할 수 있다. 정적 분석 방법은 프로그램을 실제 수행하지 않고 원시코드나 설계 명세서 등을 분석하여 프로그램을 검증하는 방법을 말한다. 동적 분석 방법은 정적 분석 방법과는 달리 프로그램의 입력 데이타를 제공하여 실제로 프로그램을 수행시켜서 오류를 찾아내는 방법이며 때로 정적 테스팅 방법으로 언급되기도 하는 정적 분석 방법과 구분하기 위하여 동적 테스팅 방법이라고도 한다. 병렬 프로그램 디버깅 방법은 정적 분석 방법이나 테스팅 방법을 통하여 발견된 오류의 위치를 국부화하기 위하여 실제 제공되는 입력 데이타에 대하여 프로그램을 반복 수행하여 프로그램의 상태를 검사하는 방법을 말한다.

이 논문에서는 Java 언어로 구현된 다중 스레드 프로그램을 대상으로 프로그램 디버깅을 위해 사용될 수 있는 결정적 수행(deterministic execution) 방법을 제시한다. 결정적 수행 방법이란 비결정적으로 행동하는 병렬 프로그램을 원하는 순서에 따라 강제적으로 수행하게 하는 방법을 말한다. Java 다중 스레드 프로그램도 다른 병렬 프로그램과 같이 비결정성을 지니고 있기 때문에 효과적인 프로그램의 검증을 위해서는 프로그램의 재수행성이 보장되어야 한다. 이를 위하여 Java 다중 스레드 프로그램에 대한 테스트케이스를 입력 데이타와 동기화 메소드(synchronized method)들의 시퀀스의 쌍으로 정의하고 주어진 입력 데이타에 대하여 프로그램을 실행할 때 동기화 메소드 시퀀스에 정의된 순서대로 프로그램을 수행할 수 있도록 하는 방법에 주안점을 둔다. 제안된 방법을 쉽게 자동화 할 수 있도록 원시코드 수준에서 주어진 동기화 메소드 시퀀스를 강제적으로 반복 수행할 수 있도록 Java 다중 스레드 프로그램을 변환하는 방법을 제시한다. 변환 규칙에 의하여 변환된 프로그램은 이전의 프로그램 수행을 재현할 수 있으므로 오류의 원인을 찾는 노력을 줄일 수 있다.

테스트케이스를 구성하는 동기화 메소드 시퀀스는 프로그램을 수행하거나 해당 명세를 분석하여 생성할 수 있다[8-13]. 만약 명세를 기반으로 동기화 메소드 시퀀스를 생성하는 경우에는 동기화 메소드 시퀀스의 정당성(validity)은 검사할 수 있으나 실행성(feasibility)을 검사하기 어렵다는 문제점이 있다. 이러한 문제점을 해결하기 위한 방법도 이 논문에서 제시한다.

이 논문의 구성은 다음과 같다. 2장에서는 병렬 프로그램의 검증을 위해 개발되었던 기존의 재수행성을 보장하는 방법들에 대하여 간략하게 설명한다. 3장에서는 Java 다중 스레드 프로그램의 재수행성을 보장하기 위한 프로그램 변환 방법을 기술하고 예를 보인다. 4장에서는 주어진 동기화 메소드 시퀀스가 실제 프로그램에서 수행이 가능한지를 판단하는 방법을 소개한다. 5장에서는 결론을 맺고 향후 연구 방향을 설정한다.

2. 관련 연구

보다 효과적으로 병렬 프로그램을 검증하기 위해서는 프로그램의 재수행성을 보장해야 한다. 지금까지 제안된 재수행성을 보장하기 위한 대부분의 결정적 수행 방법들은 다음과 같은 단계들로 구성된다.

단계 1) 탐침코드(probe) 삽입 단계: 프로그램을 실행할 때 생성되는 실행 트레이스(execution trace)를 기록하기 위한 탐침코드를 프로그램에 삽입하는 단계이다.

단계 2) 실행 트레이스 기록 단계: 탐침 코드가 삽입된 프로그램을 수행하여 실제로 프로그램 수행 동안에 발생한 모든 사건들을 기록하는 단계이다.

단계 3) 재현 단계: 단계 2)에서 기록된 실행 트레이스를 바탕으로 프로그램을 재수행하는 단계이다. 이 단계에서 프로그램의 재수행성을 보장하기 위해 원래의 프로그램에 프로그램의 재수행성을 보장하도록 하는 부가적인 코드들이 삽입(instrumentation)된다.

초기에 제시된 재수행 보장 방법들은 프로그램 수행 중에 발생하는 모든 동기화 혹은 정보 교환 사건들에 대해서 그들의 발생순서 뿐만 아니라 실제 프로그램 상태들을 기록함으로써 재수행을 보장하였다. 그러나, 이러한 방법은 기록해야 하는 양이 지나치게 많기 때문에 실제로 사용하기에는 어려움이 따른다. 이러한 문제점을 극복하기 위하여 LeBlanc와 Mellor-Crummey가 제안한 Instant Replay이라 불리는 방법[5]과 Tsai와 그 동료들이 제안한 방법[6]을 들 수 있다.

Instant Replay 방법은 프로그램 수행 중에 발생하는 모든 사건에 대해서 사건들의 상대적인 순서만을 기록함으로써 차후에 재수행을 보장한다. 이전의 방법이 모든 사건에 대해서 그 내용까지 기록했던 점에 비추어 볼 때 필요한 정보의 양을 획기적으로 줄인 방법이다. 원래 이 방법은 공유자원을 사용하는 병렬 프로그램을 대상으로 제안되었다. 프로그램 수행 중에 발생하는 각

사건에 대해서 그 사건이 사용하는 공유 자원의 종류와 그 자원을 자신이 몇번째로 사용하였는가를 기록한다. 이 정보를 바탕으로 재수행 할 때 각 사건들은 자신이 공유자원을 사용할 차례가 될 때까지 기다린다. 이런 방법으로 이 전에 수행된 프로그램의 수행 행태를 재현할 수 있다.

Tsai 등이 제안한 재수행 보장 방법은 Instant Replay 방법을 기반으로 Ada 태스킹 프로그램을 결정적으로 수행(deterministic execution)하기 위해 개발된 방법이다. 이 방법은 프로그램 수행 중에 비결정성을 유발할 수 있는 문장들을 파악하고 이 문장들 주의에 탐침코드(probe)를 삽입하여 태스크 간 상호 작용을 기록한다. 이와 같이 기록된 태스크 간의 상호 작용 순서를 재수행할 때 재현할 수 있도록 원래의 프로그램을 원시코드 수준에서 변환한다. 변환된 프로그램의 각 태스크는 기록된 정보를 바탕으로 자신이 실행할 순서가 되었는지를 판단한다. 만약, 해당 태스크가 실행될 차례가 아니라면 태스크의 실행을 보류하고 자신의 순서가 될 때까지 기다린다. 이와 같은 방식으로 Ada 태스킹 프로그램의 재실행을 보장한다. 이 밖의 방법으로 Occam 언어로 작성된 프로그램의 재실행을 보장하기 위해 개발된 방법이 있다[3].

이 논문에서는 대표적인 병렬 객체 지향 프로그래밍 언어인 Java 언어로 작성된 다중 스레드 프로그램의 재수행성을 보장하는 방법을 소개한다. 특히, 스레드 간의 통신이 모니터 등을 통하여 이루어지는 프로그램을 대상으로 하며 프로그램의 재수행을 위한 단계들 중에서 재현 단계(단계 3)를 위해 Java 다중 스레드 프로그램을 변환하는 방법에 중점을 두고 기술한다. 이 논문에서 제시한 방법은 실행 트레이스를 기록할 필요가 없이 외부로부터 직접 스레드들의 상대적인 수행 순서만을 입력받아 프로그램의 재수행성을 보장할 수 있다. 이 점은 명세로부터 테스트케이스를 추출하는 경우에 매우 유용하게 사용될 수 있다. 또한, 이 논문에서 제안한 결정적 실행을 위한 변환 방법은 Java 언어 뿐만 아니라 객체의 병렬 수행을 지원하는 다른 객체지향 언어로 작성된 프로그램에도 적용할 수 있도록 다양한 설계 패턴을 이용하였다.

3. Java 다중 스레드 프로그램의 결정적 수행 방법

이 장에서는 Java 다중 스레드 프로그램의 비결정성으로 인한 프로그램의 검증의 어려움을 해결하는 하나의 방법으로 프로그램의 실행을 결정적으로 반복 수행할 수 있게 하는 프로그램 변환 절차에 대해 기술한다. 또한, 이 변환 절차를 생산자/소비자 예에 적용하여 그 결과 및 효과에 대해 기술한다.

3.1 Java 다중 스레드 프로그램의 비결정성

병렬 프로그램의 비결정성은 내재적 비결정성(inherent nondeterminism)과 프로그래머가 유도하는 비결정성(programmer-induced nondeterminism)으로 분류할 수 있다. 내재적 비결정성은 두 개의 프로세스가 임의의 프로세스에 메시지를 송신하는 순서가 결정적이지 않을 때 발생한다. 예를 들면, 어떤 주어진 입력 값에 대해 프로그램을 실행할 때 프로세스 A가 프로세스 B보다 먼저 프로세스 C에 메시지를 전송할 수 있지만 동일한 입력 값에 대해 프로그램을 다시 실행할 때 프로세스 B가 프로세스 A보다 먼저 메시지를 프로세스 C에 전송할 수 있다. 이 경우에 프로세스 C가 메시지를 수신하는 순서에 따라 프로그램의 실행결과가 달라질 수 있다. 두 번째 형태의 비결정성, 즉 프로그래머가 유도하는 비결정성은 프로세스나 타스크 내에서 고의적으로 비결정성을 유발하는 프로그램 문장들을 사용함으로써 야기된다. 이러한 비결정성을 유발하는 프로그램 문장들은 Ada언어에서 찾아 볼 수 있는데 대표적인 것으로 "selective-wait" 문장을 들 수 있다. 그러나, Java 언어에서는 한 스레드내에서 비결정성을 유발하는 문장들이 없으므로 이 논문에서는 스레드의 수행 순서에 의해 야기되는 내재적 비결정성만을 고려한다.

Java 다중 스레드 프로그램의 (내재적) 비결정성으로 인한 문제를 이해하기 위해 (부록 1)에 주어진 생산자/소비자 프로그램을 예로 사용한다. 이 프로그램은 생산자 스레드와 소비자 스레드로 구성되어 있으며 두개의 데이타만을 저장할 수 있는 버퍼 객체를 사용한다. 생산자 스레드와 소비자 스레드는 임의의 순서대로 데이타를 생산하고 가져가게 할 수 있도록 하였다. 특히, 버퍼에서 사용되는 여러 공유 변수들을 상호 배타적으로 사용하기 위해 버퍼가 제공하는 put(int) 메소드와 int get() 메소드를 동기화 메소드로 선언하였다.

생산자 스레드는 버퍼의 현재 위치에 값을 저장하고 다음에 수행되는 put(int) 연산을 위해 버퍼의 위치를 증가시킨다. 또한, 현재 버퍼에 있는 데이타들의 수를 계산하여 버퍼가 비어 있을 경우에만 소비자 스레드가 get()연산을 하도록 하였으며 버퍼가 데이타로 채워진 경우에는 생산자 스레드가 put() 연산을 하지 못하도록 하였다. 그림 1은 이와 같은 버퍼를 이용하여 생산자 스레드와 소비자 스레드를 각각 하나씩 생성하여 수행하는 클래스를 보여준다.

```
class ProducerConsumer {
    public static void main(String args[]) {
        Buffer buf = new Buffer();
        Producer p = new Producer(buf, "Producer");
        Consumer c = new Consumer(buf);
        p.start();
        c.start();
    }
}
```

그림 1 ProducerConsumer 클래스

만약 이 프로그램에 대해 입력 데이타 (1, 2, 3)으로 put() 연산을 세 번 수행하고 동시에 get() 연산을 세 번 수행한다면 get() 연산으로부터 기대되는 출력 값은 (1, 2, 3)일 것이다. 예를 들어, 다음과 같은 동기화 메소드 순서로 프로그램이 수행된다고 가정하자:

"(put(1), get(), put(2), get(), put(3), get())".

이 때 get() 연산의 수행 결과는 기대한대로 (1, 2, 3)의 순서대로 출력될 것이다. (부록 1)에 주어진 프로그램은 대부분의 경우에 올바른 결과를 산출한다. 그러나, 만약 프로그램이 다음과 같이 주어진 동기화 메소드 순서에 따라 수행되는 경우를 고려해보자:

"(put(1), put(2), put(3), get(), get(), get())".

이 경우에는 프로그램의 출력이 (1, 2, 3) 대신 (3, 2, 3)이 출력될 것이다. 이와 같이 오류를 검출한 동기화 메소드 시퀀스는 프로그램의 비결정성 때문에 똑같은 입력 데이타를 사용하여 해당 프로그램을 여러번 반복 수행한다 할지라도 다시 재현되리라는 보장이 없다. 따라서, 오류의 원인을 파악하고 국부화 하기 위해서는 주어진 동기화 메소드 시퀀스에 나타난 순서에 따라 프로그램을 결정적으로 수행하는 방법이 필요하다.

3.2 결정적 수행을 위한 Java 다중 스레드 프로그램의 변환

Java 다중 스레드 프로그램을 변환하는 목적은 주어진 동기화 메소드 시퀀스에 따라 프로그램을 강제적으로 수행할 수 있도록 프로그램의 수행을 제어하도록 하는 것이다. 이를 위해서는 주어진 동기화 메소드 시퀀스에 정의된 순서에 따라 각 스레드에 의해 수행될 메소드들을 자신의 순서가 될 때까지 수행을 보류할 수 있는 방법이 필요하다.

이 논문에서는 프로그램의 재수행성을 보장하기 위해 제어 객체를 도입하고 프로그램을 구성하는 각 스레드들을 제어 객체와 통신할 수 있도록 변환한다. 예를 들어, 프로그램이 스레드 T1, T2, T3 세 개의 스레드로 구성되어 있을 때 변환한 후의 프로그램 구조는 그림 2에서와 같이 각 스레드가 제어 객체(C)와 통신할 수 있도록 T1′, T2′, T3′로 변환된다. 제어 객체가 수행하는 주요 작업은 스레드들을 주어진 동기화 메소드 시퀀스에 기술된 순서에 따라 스레드의 수행을 제어하는 것이다. 각 스레드는 메소드를 실행하기 전에 제어 객체에 수행할 수 있는지를 문의한다. 만약, 요청한 스레드가 수행될 순서가 아니라면 제어 객체는 해당 스레드의 수행을 순서가 될 때 까지 보류한다. 이와 같은 제어 객체는 그림 3에서 보인 바와 같이 두 개의 동기화 메소드를 제공하는 클래스로부터 쉽게 생성할 수 있다.

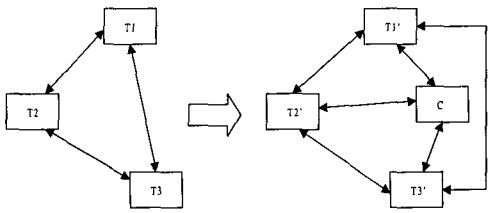

그림 2 결정적 수행을 위한 프로그램 변환

```
class ReplayController {
    ...
    public synchronized void requestPermit(ThreadID id)
    throws InterruptedException {
        while (!currentThreaID.equals(id)) wait();
    }
    public synchronized void completeMethod(ThreadID
    id) throws InterruptedException {
        currentThreadID=(ThreadID)threadID.
        elementAt(++currentID); notifyAll();
    }
}
```

그림 3 제어 객체 클래스

제어 객체는 내부적으로 프로그램을 구성하는 스레드들의 id를 저장하기 위해 큐(threadID)를 사용한다. 실제 큐에 재수행하고자 하는 동기화 메소드 시퀀스를 저장하는 대신 스레드들의 id를 저장하는 것으로 똑같은 효과를 가져올 수 있다. 왜냐하면 Java 프로그램에서는 각 스레드 간의 통신으로 인한 비결정성은 존재하지만 각 스레드는 순차적으로 실행되기 때문이다. 예를 들면, 부록 1에 있는 생산자/소비자 프로그램에서 생산자 스레드 id를 "producer"라하고 소비자 스레드 id를 "consumer"라 가정하자. 이 때 큐에 스레드 id 시퀀스

가 "(producer, producer, producer, consumer, consumer, consumer)"로 저장되어 있다면 이 시퀀스는 "(put(1), put(2), put(3), get(), get(), get())"으로 주어진 메소드 시퀀스에 대응된다는 것을 쉽게 알 수 있다.

프로그램을 구성하는 각 스레드는 동기화 메소드를 수행하기 전에 requestPermit(ThreadID) 메소드를 통하여 제어 객체에 해당 메소드를 실행할 수 있는 지를 문의한다. 제어 개체는 threadID 큐에 기술된 스레드 시퀀스 순서에 따라 스레드로 부터의 요청을 심사한다. 만약, 요청한 스레드가 현재 수행할 순서가 아니라면 스레드의 수행을 wait() 문장을 사용하여 보류하고 순서가 일치하는 경우에는 스레드의 수행을 진행하도록 하여 메소드를 호출하도록 한다. 메소드의 실행이 완료된 후에는 이 사실을 completeMethod(ThreadID) 메소드를 통하여 제어 객체에 알리고 수행이 보류된 스레드를 notifyAll()을 통하여 수행이 가능한 상태로 만든다. 또한, 제어 객체는 스레드 큐로부터 다음에 수행될 스레드를 식별한다.

이와 같은 제어 객체와 스레드 간의 통신을 위해서 각 스레드가 어떤 객체에 정의된 메소드를 호출하기 전에 제어 객체의 requestPermit(ThreadID)를 호출하는 문장을 삽입해야 하며 메소드 호출 문장 다음에는 메소드의 수행이 완료되었다는 사실을 제어 객체에 알리기 위해 completeMethod(ThreadID) 문장을 삽입할 필요가 있다. 그림 4는 이와 같이 변환한 스레드를 보여준다.

```
class TransformedThread extends Thread {
    Controller controller = ReplayController.getController();
    //*제어 객체 controller 생성
    ...
    public void run() {
        ...
        controller.requestPermit(getName()); //*제어 객체
        에 obj.method() 실행 요청 obj.method(...);
        controller.completeMethod(getName()); //*제어객
        체에 obj.method() 완료 사실 알림
        ...
    }
}
```

그림 4 스레드 변환 예

그림 4에서 볼 수 있듯이 requestPermit(ThreadID) 메소드와 completeMethod(ThreadID) 메소드는 인자로 스레드 id를 요구하기 때문에 getName() 메소드를 사용하여 현재 수행중인 스레드의 id를 식별하도록 하였다. 그림 5는 이와 같은 과정을 따라 생산자 스레드를 변환한 예를 보여준다(//*로 표시된 문장은 변환과정에서 삽입된 문장을 의미한다.). 소비자 스레드도 마찬가지 방식으로 변환된다.

```
class Producer extends Thread {
        private Buffer buf;
        ReplayController rc =
        ReplayController.getController(); //*
        public Producer(Buffer buf, String name) {
            super(name);
            this.buf = buf ;
        }
        public void run() {
            for (int i=1; i<4;i++) {
                try { //*
                rc.requestPermit(getName()); //*
                } catch(Interrupted-
                Exception e) {} //*
                buf.put(i);
                try { //*
                rc.completeMethod
                (getName()); //*
                } catch(Interrupted-
                Exception e) {} //*
            }
        }
}
```

그림 5 생산자 스레드 변환

이러한 스레드 변환에서 주의할 점은 프로그램을 구성하는 모든 스레드는 똑같은 제어 객체를 참조할 수 있어야 한다는 사실이다. 이를 위해 Singleton 설계 패턴[14]을 사용하여 ReplayController 클래스가 단 하나의 객체만을 생성할 수 있도록 하였다. 클래스가 하나의 객체만을 생성하게 하려면 클래스에서 제공하는 생성자를 private이나 protected로 선언하여 외부로부터의 접근을 제한해야 한다. 또한, 클래스의 유일한 객체를 외부로부터 참조할 수 있게 하는 메소드가 필요하다.

그림 6은 그림 3의 ReplayController 클래스를 단 하나의 객체만을 생성할 수 있도록 보완한 것이다. 여기에서 볼 수 있듯이 단 하나의 객체만을 생성하기 위해 생성자를 private으로 선언하여 외부로부터 직접 객체를 생성하지 못하도록 하였으며 실제 객체 생성을 클래스 메소드 getController()에서 수행하도록 하였다. 클래스 메소드 getController()는 ReplayController 클래스의 유일한 객체를 참조하는 정적 변수 rc를 사용하고 이 변수를 클래스의 유일한 객체로 초기화 하는 작업을 수행한다.

```
class ReplayController {
        ...
        private static ReplayController rc;
        private ReplayController() { ... }
        public static ReplayController getController() {
            if (rc == null) {
                rc = new ReplayController();
            }
            return rc;
        }
        public synchronized void requestPermit
         (ThreadID id) throws InterruptedException {
            ...
        }
        public synchronized void completeMethod
         (ThreadID id) throws InterruptedException {
            ...
        }
}
```

그림 6 Singleton 패턴을 이용한 ReplayController 클래스 설계

지금까지 기술한 Java 다중 스레드 프로그램의 결정적 수행을 위한 프로그램 변환 절차를 요약하면 다음과 같다.

단계 1) 우선 결정적으로 수행할 동기화 메소드 시퀀스들을 명세나 기존의 프로그램 수행 결과로부터 추출한다. 동기화 메소드 시퀀스를 명세로부터 추출하는 경우와 프로그램을 직접 수행하여 추출하는 경우에 따라 추출된 동기화 메소드 시퀀스의 실현성(feasibility) 문제가 발생할 수 있으므로 이 단계에 대해서는 4장에서 자세히 설명하기로 한다.

단계 2) 단계 1)에서 추출된 동기화 메소드 시퀀스를 결정적으로 수행할 제어 객체를 설계한다. 그림 6에서 보여진 제어 객체를 위한 ReplayController 클래스는 일반적인 Java 다중 스레드 프로그램을 위해 설계된 것이므로 이 단계에서 대상 프로그램에 도입하여 사용할 수 있다.

단계 3) 프로그램을 구성하는 각 스레드 내에서 동기화 메소드 시퀀스를 이루는 메소드를 포함하고 있을 때 그림 4와 그림 5에서와 같이 해당 메소드 호출 문장의 앞과 뒤에 메소드의 실행을 제어 객체에 요구하는 문장(requestPermit)과 메소드의 실행을 완료되었다는 사실을 제어 객체에 알리는 문장(completeMethod)을 삽입한다. 부록 3은 이와 같은 과정을 거쳐 변환된 생산자/소비자 프로그램을 보여준다.

단계 4) 단계 1)에서 추출한 동기화 메소드 시퀀스들에 대하여 단계 2)와 단계 3)을 통해 변환된 프로그램을 수행시켜 수행 결과를 분석한다. 만약, 동기화 메소드 시퀀스가 변환되기 전의 프로그램을 수행함으로써 얻어진 트레이스인 경우에는 이 단계에서 해당 동기화 메소드 시퀀스를 반복 수행하여 프로그램의 오류를 국부화 시킬 수 있다. 이와 달리 명세로부터 동기화 메소드 시퀀스를 추출한 경우에는 명세에서 타당한 시퀀스가 실제 프로그램에 구현이 되었는지를 판별하는 작업(실현성 검사)을 이 단계에서 수행한다.

3.3 생산자/소비자 프로그램의 재수행 예

3.2절에서 기술한 방법에 따라 생산자/소비자 프로그램을 재수행성을 보장하도록 변환했다고 가정하자. 또한, 변환된 프로그램의 수행 과정을 설명하기 위해 "(put(1), put(2), put(3), get(), get(), get())" 메소드 시퀀스에 해당하는 스레드 수행 순서 "(producer, producer, producer, consumer, consumer, consumer)"가 주어졌다고 가정하자. 또한, 프로그램을 실제 수행할 때 수행되는 스레드의 순서는 메소드 시퀀스 "(put(1), get(), put(2), get(), put(3), get())"에 해당하는 "(producer, consumer, producer, consumer, producer, consumer)"이라고 가정하자.

표 1 변환된 생산자/소비자 프로그램의 수행 예

수행할 스레드	실행을 요청한 스레드	실행 유무	실행이 보류된 스레드	실제 실행된 스레드
생산자	생산자	실행	-	생산자
생산자	소비자	보류	-	-
생산자	생산자	실행	소비자	생산자
생산자	소비자	보류	-	-
생산자	생산자	실행	소비자	생산자
소비자	소비자	실행	-	소비자
소비자	소비자	실행	-	소비자
소비자	소비자	실행	-	소비자

프로그램이 수행을 시작하면 제어 객체는 현재 생산자 스레드가 도착하기를 기다리고 있으므로(current-Thread=producer) 생산자 스레드의 수행을 허락할 것이다. 생산자 스레드의 수행이 완료되면 complete-Method() 메소드를 호출하여 제어 객체로 하여금 다음에 수행할 스레드를 식별하고 대기하고 있는 스레드를 깨운다.

그러나, 소비자 스레드가 생산자 스레드보다 먼저 제

어 객체에 get() 메소드의 수행을 요청했다고 가정했으므로 이 경우 제어 객체는 생산자 스레드가 도착하기를 기다리고 있으므로 생산자 스레드가 도착할 때까지 소비자 스레드의 수행을 보류한다. 생산자 스레드의 실행이 완료되면 다음에 수행될 스레드를 파악한 후에 대기하고 있는 소비자 스레드를 깨운다. 소비자 스레드는 현재 수행할 수 있는 스레드가 소비자 스레드이므로 깨어난 소비자 스레드는 진행을 계속할 수 있다. 이와 같은 방식으로 주어진 메소드 시퀀스의 재수행성을 보장한다. 표 1은 위의 과정을 알기 쉽게 간략화 하여 도식화 한 것이다.

4. 동기화 메소드 시퀀스의 실현성 검사

Java 다중 스레드 프로그램과 같은 병렬 프로그램에서는 스레드들 간의 상호 작용이 프로그램의 수행 결과에 영향을 미치게 되며 프로그램에 대한 효과적인 테스팅을 위해서는 스레드 또는 프로세스들간의 상호 작용 순서를 테스팅할 수 있는 방법이 필요하다. 따라서 병렬 프로그램에서는 단순히 입력 데이타가 테스트케이스를 구성하는 순차적인 프로그램의 테스팅과는 달리 병렬 프로그램의 테스트 케이스는 보통 입력 데이타와 동기화 메소드 시퀀스의 쌍으로 정의된다. 이와 같은 병렬 프로그램의 특성을 고려하여 동기화 메소드 시퀀스를 구하는 방법에 대한 연구가 현재 활발하게 진행되고 있다.

동기화 메소드 시퀀스를 구하는 방법은 프로그램 코드를 바탕으로 하는 방법과 명세를 분석하여 얻는 방법으로 구분할 수 있다. 코드 기반 병렬 프로그램 테스팅 방법에서는 단순한 입력 데이타만을 가지고 프로그램을 실제 수행시켜서 프로그램이 수행중에 행한 동기화 메소드 시퀀스를 얻은 후 이 시퀀스를 이용해서 재수행시키거나 얻어진 시퀀스를 변형해서 수행시키는 방법을 채택하고 있다. 이와 같은 방법은 프로그램 수행 중에 발생하는 다양한 메소드 시퀀스를 체계적으로 검사하기에 적합하지 않다는 단점이 있다.

한편으로, 요구 명세로부터 병렬 프로그램을 위한 테스트케이스를 자동으로 도출해내는 방법에 대한 연구들도 진행되고 있다. 여기서는 입력 데이타와 메소드 시퀀스와의 관계를 잘 표현하고 있는 명세 언어가 별로 없다는 문제점이 있는데 현재 사용하고 있는 명세 언어들 중에서 상태전이도[8]와 MSC[15]를 기반으로 병렬 프로그램의 테스트케이스를 구성하기 위한 메소드 시퀀스를 도출하는 방법[9-11]이 제시되었다. 그러나, 명세를 기반으로 메소드 시퀀스를 구하는 경우에는 메소드 시퀀스의 정당성(validity)은 검사할 수 있으나 명세의 실현성(feasibility)을 검사할 수 없다는 문제점이 있다.

예를 들어, 부록 1에 있는 생산자/소비자 프로그램을 테스팅하기 위해 "(put(1), put(2), put(3), get(), get(), get())" 메소드 시퀀스에 해당하는 스레드 수행 순서 "(producer, producer, producer, consumer, consumer, consumer)"가 명세로부터 추출되었다고 가정하자.또한, 버퍼 클래스의 put() 메소드에 있는 조건문 "if (c>size)"에 의해 오류가 발생하므로 올바른 조건문 "if (c==size)"으로 수정하고 재수행성을 보장하도록 프로그램을 변환했다고 가정하자. 실제 이 시퀀스에 따라 프로그램을 강제적으로 수행시키면 데드락 상태에 이른다는 것을 알 수 있다. 이는 생산자 스레드가 세 번째 put(3) 메소드를 수행할 때 버퍼가 모두 채워져 있는 상태이므로 wait()문장에 의해 소비자 스레드가 get() 메소드를 실행할 때까지 대기 상태에 들어가기 때문이다. 즉, 스레드 수행 순서에 따라 소비자 스레드는 세 번째 put(3) 메소드의 수행이 완료된 후에 수행이 되므로 생산자 스레드는 결코 일어나지 않을 사건을 기다리는 상황이 발생한다. 실제 오류를 수정한 프로그램은 "(put(1), put(2), put(3), get(), get(), get())"와 같은 데드락을 발생할 수 있는 메소드 시퀀스를 허용하지 않는다.

따라서, 이와 같이 병렬 프로그램을 테스팅하기 위해 명세로부터 메소드 시퀀스를 추출하는 경우에는 추출된 메소드 시퀀스가 해당 프로그램에서 수행가능성이 있는지를 판별할 수 있는 수단이 필요하다. 그러나 임의의 프로그램에 대해 주어진 메소드 시퀀스를 수행 가능한지를 판단하는 문제는 풀 수 없는 문제(undecidable problem)이기 때문에 이를 근사적으로 해결할 수 밖에 없다.

이 논문에서는 이 문제를 해결하기 위하여 스레드가 실행을 요청한 메소드를 어느 시간 동안 수행하지 못한다면 주어진 메소드 시퀀스를 실현가능성이 없다고 판정하는 timed-wait 설계 패턴[9]을 이용하였다. Java 언어에서는 이러한 경우를 위해 wait(long msec) 메소드를 제공하며 이 메소드는 스레드를 msec 동안 실행을 보류시키고 스스로 깨어나게 하는 작업을 수행한다. 그림 7은 wait(long msec)를 이용하여 설계한 requestPermit() 메소드의 구현 내역을 보여준다.

```
public synchronized void requestPermit(String id)
                                    throws InterruptedException {
    if (!curThread.equals(id)) {
        long waitTime = maxWaitMillis;
        long startTime = System.currentTimeMillis();
```

```
        for (;;) {
            try {
                wait(waitTime);
            } catch(InterruptedException e) {}
            if (curThread.equals(id)) break;
            else {
                long now = System.currentTime-
                        Millis();
                long timeSoFar = now - startTime;
                If (timeSoFar >= maxWaitMillis) {
                System.out.println("Provably Infeasible
                  Sequence");
                System.exit(0);
            }
            else  waitTime = maxWaitMillis -
                        timeSoFar;
        }
    }
}
```

그림 7 수정된 requestPermit() 메소드의 구현 내역

그림 7에서 maxWaitMillis 변수는 스레드가 메소드의 실행을 요청하고 요청이 받아들여질 때까지 기다리는 최대 시간이다. 만약 이 시간 동안 요청이 받아들여지지 않는다면 프로그램이 주어진 메소드 시퀀스를 수행할 수 없다고 판단한다.

그림 7에 있는 requestPermit()를 사용하여 오류가 제거된 생산자/소비자 프로그램을 "(producer, producer, producer, consumer, consumer, consumer)"로 주어진 스레드 순서에 따라 강제적으로 수행하는 경우를 고려해보자. 이 경우에 그림 2에 정의된 requestPermit() 메소드를 사용하는 경우와 달리 생산자 스레드가 세 번째 put(3)을 수행할 때 주어진 시간 동안 소비자 스레드가 get() 메소드를 수행하지 않으므로 이 메소드의 수행가능성이 없다는 사실을 발견하고 프로그램의 수행을 종료시킨다는 사실을 알 수 있다. 따라서, 이 시퀀스는 명세적으로는 타당하지만 프로그램에서는 구현되지 않은 시퀀스임을 알 수 있다. 부록 2는 앞에서 기술한 모든 상황을 고려하여 설계한 제에 객체 클래스(RepalyController)를 보여준다.

5. 결론 및 향후 연구 방향

이 논문에서는 Java 다중 스레드 프로그램의 비결정성으로 인한 문제를 해결하기 위한 프로그램 변환 방법을 제안하였다. 주어진 동기화 메소드 시퀀스를 프로그램이 결정적으로 수행할 수 있도록 프로그램의 수행을 제어하기 위한 제어 객체를 설계하였다. 제어 객체를 설계할 때 singleton 설계 패턴을 이용하여 프로그램을 구성하는 각 스레드들이 하나의 공통적인 제어 객체만을 접근할 수 있도록 하였다. 또한, 명세로부터 메소드 시퀀스를 추출하는 경우를 위해 timed-wait 설계 패턴을 기반으로 주어진 메소드 시퀀스가 실제 프로그램에서 수행이 가능한지를 판별할 수 있도록 하였다. 이 논문에서 제안한 프로그램 변환 방법은 Java 다중 스레드 프로그램과 같은 일반적인 병렬 객체 지향 프로그램에 쉽게 적용할 수 있으리라 기대한다.

이 연구와 관련하여 추진 중인 연구는 다음과 같다. 현재 이 논문에서 제안한 방법은 명세적으로 타당한 모든 시퀀스가 프로그램에 구현되어야 한다는 것을 가정하고 있다. 즉, 명세적으로 타당한 시퀀스를 결정적으로 수행할 수 없다면 해당 프로그램은 오류가 있다고 가정한다. 그러나, 이러한 가정은 실제 병렬 프로그램을 구현할 때 명세에 나타난 시퀀스들의 일부만을 구현하여도 프로그램이 여전히 명세를 만족하는 경우를 처리하지 못한다. 예를 들어, 전화를 거는 과정을 고려해보자. 호출자(caller)가 전화를 걸 때 송신음을 듣는 사건(receive_free_tone)과 피호출자가 수신음을 듣는 사건(receive_ringing_tone)이 발생하는 순서는 비결정적이다. 즉, 호출자가 송신음을 듣는 사건이 피호출자가 수신음을 듣는 사건보다 먼저 발생할 수 있을 뿐만 아니라 피호출자가 수신음을 듣는 사건이 호출자가 송신음을 듣는 사건보다 먼저 발생할 수 있다. 이러한 경우에는 해당하는 사건들 중 어느 사건이 먼저 발생해도 프로그램의 기능에 영향을 주지 않기 때문에 실제 프로그램을 개발할 때 가능한 모든 시나리오를 고려하지 않고 임의의 한 시나리오만을 고려하여 구현할 수 있다. 이 문제를 해결하는 한 방법으로써 주어진 시퀀스로부터 의미적으로 동일한 시퀀스 집합을 생성하는 방안을 생각해 볼 수 있다. 이 방법은 반드시 주어진 시퀀스를 프로그램이 수행하지 않는다 할지라도 생성된 시퀀스 집합에 포함되어 있는 시퀀스들 중 하나만을 프로그램이 수행하면 오류로 처리하지 않는다. 이에 대한 연구가 현재 진행중이다[16].

참 고 문 헌

[1] S. M. Shatz and W.K. Cheng, "A Petri Net Framework for Automated Static Analysis of Ada Tasking Behavior," *Journal of Systems and Software*, vol. 8, pp. 343-350, 1988.

[2] R. N. Taylor, "A General Purpose Algorithm for Analyzing Concurrent Programs," *Communications of ACM*, Vol. 26, no. 5, pp. 362-376, 1983.

[3] A. Cimitle and U. De Carlini "Replay-based

Debugging of Occam Programs," Journal of Software Testing, Verfication and Reliability, vol. 3, pp. 83-100, 1993.

[4] J. H. Griffin, H. J. Wasserman, and McGarvan, "A Debugger for Parallel Processes," *Software Practice and Experience*, vol. 18, no. 12, pp. 1179-1190, 1988.

[5] T. J. LeBlanc and J. M. Mellor-Crummey, "Debugging Parallel Programs with Instant Replay," *IEEE Trans. on Computers*, vol. C-36, no. 4, pp. 471-482, 1987.

[6] K. C. Tai, R. H. Carver and E. E. Obaid, "Debugging Concurrent Ada Programs by Deterministic Execution," *IEEE Trans. on Soft. Eng.*, vol. 17, no. 1, pp. 45-63, January 1991.

[7] M. Young and R. N. Taylor, "Combining Static Analysis with Symbolic Execution," *IEEE Trans. on Soft. Eng.*, vol. 14, no. 10, pp. 1499-1511, 1988.

[8] H. AboElFotoh, O. Abou-Rabia, and H. Ural, "A Test Generation Algorithm for Systems Modelled as Nondeterministic FSMs," *IEE Software Eng. Journal*, pp. 184-188, July 1993.

[9] I. S. Chung, et al., "Testing of Concurrent Programs based on Message Sequence Charts," *Proc. of Int'l Symp. on Soft. Eng. for Parallel and Distributed Systems*, pp. 72-82, May 1999.

[10] I. S. Chung, et al., "Testing of Concurrent Programs After Specification Changes," *Proc. of Int'l Conf. on Software Maintenance*, Oxford, England, pp. 199-210, August 1999.

[11] J. Grabowski, D. Hogrefe, I. Nussbaumer, and A. Spichiger, "Test Case Specification Based on MSCs and ASN.1," *Proc. of the Seventh SDL Forum 1995*, pp. 307-322, 1995.

[12] D. Rosenblum, "Specifying Concurrent Systems with TSL," *IEEE Software*, pp. 52-61, May 1991.

[13] K. C. Tai and R. H. Carver, "A Specification-Based Methodology for Testing Concurrent Programs," *Proc. of European Software Eng. Conf.*, pp. 154-172, 1995.

[14] E. Gamma et al., Design Patterns: Elements of Reusable Object-Oriented Software Addison-Wesley, 1995.

[15] ITU-T Recommendation Z.120: Message Sequence Chart (MSC), September 1994.

[16] I. S. Chung, et al., "A New Approach to Deterministic Execution Testing for Concurrent Programs," *Proc. of Int'l Workshop on Distributed System Validation and Verfication,* Taiwan, pp. E59-E66, March 2000.

[17] D. Lea, Concurrent Programming in Java, Addison-Welsely, 1997.

부 록 1: 생산자/소비자 프로그램

```
class Producer extends Thread {
        private Buffer buf;
        public Producer(Buffer buf, String name) {
                super(name);
                this.buf = buf ;
        }
        public void run() {
                for (int i=1; i<4;i++)  buf.put(i);
        }
}
class Consumer extends Thread {
        private Buffer buf;
        public Consumer(Buffer buf), String name {
                super(name);
                this.buf = buf;
        }
        public void run() {
                int value = 0;
                for (int i = 1 ; i < 4 ; i++ ) value = buf.get();
        }
}
class Buffer {
        private int[] contents = new int[2];
        private int n = 0;
        private int ip = 0;
        private int op = 0;
        private int size  = 2;

        public synchronized int get() {

                if (n ==0)  try {
                wait();
                } catch (InterruptedException e) {}

                int item = contents[op];
                op = (op+1) % size;
                n--;
                System.out.println(item);
                notifyAll();
                return item;
        }
        public synchronized void put(int value) {

                if (n>size)  try {
                wait();
                } catch (InterruptedException e) {}

                contents[ip] = value;
                ip = (ip+1) %size;
                n++;
                notifyAll();
        }
}
public class ProducerAndConsumer {
```

```
        public static void main(String args[]) {
        Buffer buf = new Buffer();
        Producer p1 = new Producer(buf, "Producer");
        Consumer c1 = new Consumer(buf, "Consumer");

        p1.start() ;
        c1.start() ;
    }
}
```

부 록 2: 제어 객체 클래스(ReplayController)

```
package Controller;

import java.io.*;
import java.util.*;


class InfeasibleException extends Exception {
        public InfeasibleException(String message) {
                super(message);
        }
}

public class ReplayController {

        private static ReplayController ctrl;
        private Vector threadID = new Vector();
        private Enumeration venum;
        private String currentThread;
        private int currentID = 0;
        private int sizeTrace = 0;
        private Hashtable timeInfo = new Hashtable();
        private long maxWaitMillis;


        public synchronized static ReplayController getController() {
                if (ctrl== null) return new ReplayController();
                return ctrl;
        }

        // trace 파일로부터 스레드 수행 순서를 threadID 큐로 읽는 메소드 public
        synchronized void readTrace(String trace, long t) throws IOException {

                FileReader tf = new FileReader(trace);
                BufferedReader btf = new BufferedReader(tf);
                String s, s1;

                while ((s = btf.readLine()) != null) {
                        threadID.addElement(s1=new String(s));
                        if (!timeInfo.containsKey(s1)){
                            timeInfo.put(s1, String.valueOf
                    ((long)t));
                        }
                }

                btf.close();

                maxWaitMillis = t;
                sizeTrace = threadID.size();
                if (sizeTrace >0)
                        currentThread = (String)threadID.
                                                elementAt(curID);
        }

        // 스레드가 메소드의 수행을 요청할 때 사용하는 메소드: timed-wait 설계 패턴
        사용        public synchronized void reqPermit(String id)
                        throws InterruptedException {

                if (!currentThread.equals(id)) {
                        long waitTime = maxWaitMillis;
                        long startTime =
                                        System.currentTimeMillis();
                        for (;;) {
                                try {
                                        wait(waitTime);
                                } catch(InterruptedException e) {}

                                if (currentThread.equals(id))
                                                        break;
                                else {
                                    long now =
                                            System.currentTimeMillis();
                                    long timeSoFar = now -
                                                        startTime;
                                    if (timeSoFar >=
                                                        maxWaitMillis) {
                                        try {
                                        throw new Infeasible-
                                                Exception("Infeasible");
                                        }
                                        catch(Infeasible-
                                                        Exception e) {
                                         System.out.println(e);
                                         System.exit(0);
                                        }
                                    } else
                                        waitTime =
                                            maxWaitMillis - timeSoFar;
                                }
                        }
                }

        }

        // 스레드가 메소드의 수행을 완료한 후에 호출하는 메소드
        public synchronized void completeMethod(String id)
                        throws InterruptedException {
                curThread = (String)threadID.elementAt(++curID);
                System.out.println(id + "    " + curThread);

                notifyAll();
        }

}
```

부 록 3: 변환된 생산자/소비자 프로그램

```
import Controller.*;
import java.io.*;

class Producer extends Thread {
        private Buffer buf;
        Controller ctrl = ReplayController.getController(); //제어 객체를 얻기 위한 probe
        public Producer(Buffer buf, String name) {
                super(name);
                this.buf = buf ;
        }
```

```
        public void run() {
                for (int i=1; i<4;i++) {
                        try {
                            ctrl.reqPermit(getName()); //실행을 요청하는 probe
                            buf.put(i);
                            ctrl.completeMethod(getName());//실행이 완료되었음을
                            알려주는 probe
                        }  catch (InterruptedException e) { }
                }
        }
}

class Consumer extends Thread {
        private Buffer buf;
        Controller ctrl = ReplayController.getController(); //제어 객체를 얻기 위한 probe
        public Consumer(Buffer buf, String name) {
                super(name);
                this.buf = buf;
        }

        public void run() {
                int value = 0;
                for (int i = 1 ; i < 4 ; i++ ) {

                        try {
                            ctrl.reqPermit(getName()); //실행을 요청하는 probe
                            value = buf.get();
                            ctrl.completeMethod(getName());//실행이 완료되었음을
                            알려주는 probe
                        } catch (InterruptedException e) {}
                }
        }
}

class Buffer {
        private int[] contents = new int[2];
        private int n = 0;
        private int ip = 0;
        private int op = 0;
        private int size  = 2;

        public synchronized int get() {

                if (n ==0)  try {
                                wait();
                } catch (InterruptedException e) {}

                int item = contents[op];
                op = (op+1) % size;
                n--;
                System.out.println(item);
                notifyAll();
                return item;
        }
        public synchronized void put(int value) {

                if (n>size)  try {
                                wait();
                } catch (InterruptedException e) {}

                contents[ip] = value;
                ip = (ip+1) %size;
                n++;
                notifyAll();
        }
}
```

```
public class ProducerAndConsumer {
        public static void main(String args[]) {
        Controller ctrl = ReplayController.getController(); // 제어 객체를 얻는 probe
        try {
                ctrl.readTrace(args[0], Integer.parseInt(args[1])); //스레드 수행 순서를
                threadID 큐로 읽는 probe
        } catch (IOException e) {}

        Buffer buf = new Buffer();
        Producer p1 = new Producer(buf, "Producer");
        Consumer c1 = new Consumer(buf, "Consumer");

        p1.start() ;
        c1.start() ;
    }
}
```

정 인 상
정보과학회논문지 : 소프트웨어 및 응용
제 27 권 제 2 호 참조