HDTL을 이용한 병렬 자바 프로그램의 모니터링과 검사

(Monitoring and Checking Concurrent Java Programs with HDTL)

조 승 모 † 김 형 호 †† 차 성 덕 ††† 배 두 환 †††

(Seung-Mo Cho) (Hyung-Ho Kim) (Sung-Deok Cha) (Doo-Hwan Bae)

요 약 정형 명세를 이용하여 구현된 프로그램이 수행 중에 명세를 만족시키는지 모니터링하고 검사하는 기법에 대한 연구들이 기존에 많이 수행되어 왔다. 이들은 주로 요구사항 명세언어로 시제논리 혹은 그것의 확장을 사용하게 된다. 이때 대부분의 연구는 실제 구현된 시스템이 가지는 동적인 변화를 제대로 요구사항 명세에 기술하고, 검사할 수 있는 언어를 제공하지 못하고 있다. 본 연구에서는 동적 시스템의 특성 명세언어로 기존에 제안했던 HDTL을 사용하여 동적인 자바 프로그램의 수행을 모니터링하고 검사하는 프레임웍을 제안한다.

키워드 : 정형 명세, 동적 시스템 검증, 자바

Abstract There have been many researches about monitoring and checking the implementations during run-time using formal requirement specification. They usually adopt temporal logics or their extensions to specify the requirements for the implementations. However, most of the systems fail to support the specification of requirements for dynamic systems - systems whose components are created and removed during run-time. Unlike analysis or design models, most actual implementations are dynamic, so the notion of instances should be employed in the property specification language. In this paper, we show how we can monitor and check Java programs using our temporal logic for dynamic systems (HDTL). We suggest a framework in which the execution of Java programs are monitored and checked against given HDTL requirements.

Key words : formal specification, dynamic system verification, Java

1. 서 론

정형 명세를 이용한 소프트웨어 시스템의 검증은 오래 연구되어 왔다. 예를 들어 병렬 시스템이나 반응 시스템(reactive systems) 의 경우, 시제논리(temporal logic)[1]로 그 시스템이 만족해야 할 요구사항을 기술하고, 시스템의 모델이 실제 그 요구사항을 만족시키는지를 검증하는 방법론이 사용되어 왔다. 이 과정에서 많은 연구들이, 검증의 알고리즘적인 복잡도의 문제를 해결하기 위해 수행되었다. 하지만, 아직도 검증하려는 시스템의 모델이 실제와 유사한 복잡도를 가질 때, 만족스럽게 검증해 내는 단계까지는 가지 못했다고 말해야 할 것이다. 대부분의 연구는 실제 시스템을 그대로 검증하지 못하고, 많은 추상화(abstraction)을 거쳐 만들어진 모델에만 적용될 수 있다.

따라서 정형 명세를 이용하는 최근의 다른 연구의 흐름은, 명세의 검증을 모델이 아니라, 실제 코드의 수행에 직접 적용하는 것이다. 이 경우, 시스템의 모든 수행을 검사할 수 없으므로, 시스템이 명세를 만족한다는 것을 보일 수는 없다. 하지만, 명세를 어기는 부분이 발생하면 그것을 찾아줄 수 있으므로, 시스템의 정확성에 대한 확신(confidence)를 높이는데 도움이 될 수 있다.

이는 특히 런타임 모니터링[2, 3, 4]에 유용하게 적용될 수 있다. 즉 우리가 시스템이 꼭 만족해야 하는 요구사항을 정형적으로 기술했다고 할 때, 현재의 정형기법

† 비 회 원 : 한국과학기술원 첨단정보기술연구센터
seung@salmosa.kaist.ac.kr
†† 비 회 원 : 한국과학기술원 전산학과
hhkim@salmosa.kaist.ac.kr
††† 종신회원 : 한국과학기술원 전산학과 교수
cha@salmosa.kaist.ac.kr
bae@salmosa.kaist.ac.kr
논문접수 : 2001년 10월 9일
심사완료 : 2002년 2월 14일

으로는 그것이 실제 코드에서 만족되는지를 검증하기가 힘들다. 또한 테스팅 역시 100%의 coverage를 달성하지 못하므로, 안심할 수 없다. 이러한 경우 정형적 요구사항 명세에 기반한 런타임 모니터링을 수행하면, 적어도 현재 수행되고 있는 프로그램의 수행(execution)은 요구사항을 만족하고 있음을 알 수 있다. 그 밖에도 이 기법은 테스팅 과정에서 오라클(oracle)[5]로 사용할 때도 같은 식으로 적용된다.

이렇게 구현이 만족해야 하는 요구사항을 정형적으로 명세하고 검사하려는 연구들이 기존에 여럿 존재하였다. 하지만, 실제 구현언어에 기존의 명세언어를 사용할 경우, 둘 사이의 표현력의 차이가 문제가 된다. 특히 병렬 시스템을 대상으로 하는 경우, 요구사항 명세에 많이 사용되는 시제논리가, 실제 구현과 연결되어 사용될 수 있는지의 문제가 있다. 특히 시제논리를 사용할 때, 대부분의 구현은 검증에 사용되는 모델들과 달리, 동적(dynamic) 으로 시스템의 구조가 변한다는 문제를 해결해야 한다. 우리는 이 문제를 [6]에서 다룬 바 있다. [6]에서 우리는 새로운 시제논리인 HDTL (Half-order Dynamic Temporal Logic) 을 제안하였고, 그 분석 기법을 제시하였다.

이 논문은 HDTL의 한 응용으로, JAVA 프로그램이 만족해야 할 요구사항이 수행 중에 지켜지고 있는지를 체킹하는 기법을 제안한다. 이 연구를 위해, 우리는 새로이 MAUDE[7]로 HDTL checker를 구현했다. 또한 실제 수행되는 자바 프로그램의 수행과정을 분석하기 위해, 자바에서 표준으로 제공되는 JPDA (Java Platform Debugger Architecture) [8]를 사용하였다. JPDA를 사용하여, 프로그램의 수행 중에 생성되는 이벤트들을 수집하여, 그 이벤트 트레이스가 주어진 요구사항에 위배되지 않는지를 검사한다.

본 문서의 구성은 다음과 같다. 2장에서는 관련 연구들에 대해 정리하였다. 이는 병렬 객체지향 프로그램의 모니터링에 관련된 최근의 세가지 연구 결과에 대한 정리와 비교, JPDA에 대한 간략한 소개, 그리고 HDTL의 기본적인 사항들을 포함한다. 3장에서는 MAUDE와, 그로 구현된 HDTL checker에 대해 설명한다. 4장에서는 병렬 자바 프로그램의 수행 모델과, 우리가 모니터링하고자 하는 시스템이 만족해야 하는 가정에 대해서 설명하고, 행한 실험에 대해 정리하였다. 그리고 5장에서 결론을 맺는다.

2. 관련 연구

1.1 정형명세 기반 객체지향 시스템 모니터링

이 장에서, 우리는 최근에 수행된 자바, 혹은 병렬 객체 지향 시스템에 대한, 정형명세에 기반한 모니터링에 관련된 세가지 연구들에 대해 살펴보고자 한다. 각각은 유사하지만, 약간씩 다른 접근 방법을 취하고 있다. 먼저 각 방법론을 설명하고, 이들 서로간의, 그리고 우리가 제안하는 방법과의 비교를 하도록 하겠다.

먼저 들 수 있는 것은, MaC 시스템[2]이다. 이 연구에서는 자바 프로그램의 수행을 모니터링하고, 정형 명세 언어로 작성된 명세가 만족되고 있는지를 검사하는 것을 목적으로 한다.

사용자는 모니터링하고자 하는 프로그램을 보고, 어떤 사건들을 모니터링할 것인지를 결정하여, 그것을 PEDL (Primitive Event Definition Language)라는 언어로 기술한다. 이는 구현과 관련되는 부분과, 그보다 윗 레벨의 요구사항을 분리하기 위한 방법이다. 예를 들어, '기차가 지나가기 전에는 항상 차단기가 내려와 있어야 한다' 라는 요구사항을 기술한다고 하면, 기차가 지나가는 사건, 차단기가 내려온다는 사건 등이, 자바 프로그램의 어떤 변수, 어떤 코드와 연결되는지를 구체적으로 기술하는 것이 PEDL의 역할이다.

기술된 PEDL 명세는 모니터링할 대상 시스템을 수정하는데 사용된다. 즉 수행되는 프로그램의 바이트코드를 변경하여, 특정한 사건이 발생할 때마다 event를 보내도록 만들어 주는 것이다. 이러한 변경은 PEDL을 해석한 후에 자동으로 수행된다.

또 하나의 부분이 MEDL (Meta Event Definition Language) 이다. 이는 시스템의 요구사항을 PEDL에서 정의한 사건과 조건들을 사용해서 나타내는 부분이다. MEDL 명세는 run-time checker를 자동으로 생성하는데 사용된다. 시스템이 발생시킨 이벤트들을 여기서 검사하여 MEDL 명세에 어긋나는 동작이 일어나지 않는지를 체크한다. MEDL은 일반적인 Linear Temporal Logic (LTL) 에 기반을 두고 설계되었으나, 표현력을 위해 몇가지 변경이 가해졌다. 사건 뿐 아니라 조건을 기술 할 수 있게 한 것, 그리고 변수를 사용할 수 있게 한 것 등이 중요한 변경점이다.

우리는 MaC 시스템의 문제점으로 크게 두가지를 지적하고자 한다.

첫째, 모니터링 할 시스템의 구성이 정적이라고 가정하고 있다는 점이다. 여기서는 각 클래스들은 하나의 인스턴스(객체) 만을 가진다는 것으로 가정된다. 예를 들어, 한 멤버함수에 대한 호출은, 그것이 어떤 객체에 대한 호출이건 모두 같은 것으로 취급된다. 따라서 동적인 시스템의 경우, 정확한 모니터링이 이루어지지 못한다. 우리는 이것을 가장 큰 문제점으로 본다. 예를 들어, 다음 PEDL 명세의 일부를 보자.

```
event queryResend
        =startM(Client.retryGetData(int));
```

이는 queryResend라는 사건(event)를 정의하는 부분이다. 이 사건은 Client라는 클래스의 retryGetData라는 메소드가 시작될 때 (startM), 발생하는 것으로 정의된다. 따라서 시스템에 Client라는 클래스의 객체가 여럿 존재할 때, 어떤 객체의 retryGetData 메소드가 호출되었던지 상관없이, queryResend 사건이 발생하게 된다.

둘째, MaC이 사용하는 명세언어인 MEDL과 PEDL은 표준적인 특성명세 언어와 많이 다르다. 구현에 의존적인 PEDL은 어쩔 수 없는 부분이지만, MEDL까지도 변수 등의, 일반적인 메모리 중심 언어(imperative language)에서 볼 수 있는 특징을 가지고 있다. 이는 물론 표현력의 확장이라는 목표에서 나온 것이다. 하지만, 이는 이 언어가 추론(reasoning), 모델체킹 등의 다른 검증 방법에도 사용되는 것을 막는 역할도 하게 된다. 즉 여기서 작성된 MEDL 요구사항 명세는 이 방법론에 따라 구현을 모니터링하는 용도로 밖에 쓰이지 못한다. 뒤에 살펴볼 다른 연구들이 표준적인 시제논리를 사용함으로써, 같은 명세를 다른 분석 방법에서도 사용할 수 있는 점과 구별된다고 할 수 있다.

두번째로 살펴볼 것은 NASA의 AMES 연구소에서 수행중인 Java PathExplorer [3]이다. 이곳에서는 기존에 Java 시스템의 모델체킹에 대한 연구도 수행한 바 있다.

이 시스템은 MaC 에서 많은 영향을 받았다고 밝히고 있다. 가장 큰 부분은, MaC처럼 두가지 명세언어를 사용한다는 점이다. 따라서 전체적인 시스템 구조는 유사하다.

PathExplorer는 Compaq에서 제작된 Jtrek[10]이라는, 자바 바이트코드를 수정하는 도구를 사용한다. 따라서 여기서 PEDL에 해당되는 부분은 Jtrek을 위한 스크립트 언어이다. MaC과 달리, 이 시스템에서는 표준적인 특성명세 언어인 LTL을 그대로 사용한다. 그리고 이 명세언어를 수행하기 위해서 rewrite logic system인 MAUDE를 사용한다.

이 방법의 문제점으로 우선 들 수 있는 것은 LTL을 사용했다는 점이다. 물론 이는 MaC의 MEDL에 비해 표준적인 특성명세언어라는 장점을 가진다. 하지만 LTL은 기본적으로 propositional logic에 기반하고 있으므로, 표현력에 한계가 있다. 이로 인해 우선, 이 방법론 역시, 동적인 시스템을 다룰 수 없다는 한계가 있다. 우리의 관점에서 볼 때, 다른 검증기법에서도 사용할 수 있는 표준적이면서도 적당한 표현력을 갖춘 언어가 필요하다고 보인다.

마지막으로 살펴볼 시스템은 스위스의 EPFL에서 개발한 MOTEL[4]이다. 이 시스템은 병렬 객체지향 시스템의 모니터링을 위한 것으로, 특정 구현언어에 한정적인 시스템은 아니다. 여기서는 CORBA를 사용하는 환경을 가정하고, 상용 CORBA 환경이 제공하는 기구들을 사용하여 시스템을 모니터링한다.

이 시스템이, MaC이나 PathExplorer와 다른 가장 큰 차이점은, 시스템의 특성을 기술하기 위한 추상화 수준(abstraction level)이 미리 결정되어 있다는 점이다. 위의 두 시스템에서는 사용자가, 관찰할 사건들을 정의하도록 되어 있는데 반해, 여기서는 그것이 미리 정해져 있다.

MaC 에서의 PEDL 과 같이, 모니터링 프로그램이 관찰하고자 하는 사건(event)에 대한 정의를 사용자가 직접 할 수 있게 하는 것을 2-layered approach 라고 부를 수 있다. 반면, MOTEL의 경우, 사건에 대한 정의 자체가 미리 정의되어 있다. 즉 시스템에서 일어나는 사건은, 객체들간의 상호작용(메시지 전달)만으로 한정된다. 즉 시스템의 요구사항은 메시지 전달 수준에서 표현되어야지, 그보다 자세한 수준에서는 표현할 수 없다는 말이다. 즉 예를 들면, 한 메소드 내부에서의 변수값의 변화 등은 이벤트로 고려할 수 없다. 이를 위해서, 이 연구에서는 병렬 객체 지향 시스템의 수행모델을 정의하고, 그 모델상에서 발생하는 사건들을 유형별로 정의하고 있다. 따라서 이런 방법을 우리는 1-layered approach라고 칭한다.

이러한 두가지 방법 - 사용자에게 사건의 정의를 맡기는 것 vs. 미리 정의된 추상화 수준을 제공하는 것 - 은 각각의 장단점을 가진다. 전자는 사용자에게 많은 선택폭을 주어, 엄밀한 모니터링이 가능하게 하고, 더 많은 특성을 기술할 수 있게 한다는 장점이 있다. 반면, 후자는 더 손쉬운 명세작성을 가능하게 하고, 작성된 명세가 이해하기 쉬우며, 모니터링에 사용된 명세를 다른 검증방법에도 사용할 수 있는 가능성을 제시한다. 우리의 연구에서는 후자의 접근방법을 취하였다.

MOTEL에서 사용하는 특성명세 언어는 LTL에 기반하고 있다. 거기에, 사건의 발생 횟수, 그리고 객체 개념을 포함하기 위한 확장이 더해져 있다. 즉 위의 두 방법과 달리, MOTEL에서는 한 클래스의 서로 다른 객체에 대한 호출을 구분할 수 있도록 되어 있다. 예를 들어, 다음과 같은 명세를 보자.

```
∀ o1∈C1. ∃ o2∈C2,
□ ( o_inReq(*, o1, m1, *) →
      ¯ o_inReq(o1, o2, m2, *) U o_inReq(o1,
        o2, m3, *) )
```

o_inReq(a, b, c, d) 는, [4]에서 정의하고 있는 미리 정의된 사건들 중의 하나로서, 객체 a에서 객체 b의 메소드 c를, d라는 파라메터를 가지고 호출하는 사건을 가리킨다. 따라서, 이 명세는, 모든 C1 클래스에 속하는 객체 o1 에 대해, C2 클래스의 객체 o2가 존재하여, 그 때 o1의 메소드 m1 이 호출되었다면, o1이 o2의 메소드 m3를 호출하기 전에는 m2를 호출할 수 없다는 것을 의미한다. 여기서 보듯, 한정자를 사용하여 자연스럽게 동적인 시스템 - 객체의 수가 여럿이고 변할 수 있는 - 의 요구사항 명세를 나타내고 있음을 알 수 있다.

그러나, 문제는 이러한 확장에 대한 정형적인 뒷받침이 되어 있지 않다는 점이다. 그들의 LTL 확장은 객체에 대한 한정자를 이용해 객체라는 개념을 수용하고 있다. 논문에서는 그들의 명세 언어를 일반적인 tableau method를 이용하여 오토마타로 변환, 시스템의 수행을 체크할 수 있다고 기술되어 있다. 하지만, 논문에서 설명하고 있는 오토마타로의 변환 과정에서는 일반적인 LTL의 경우만 설명하고 있고, 한정자가 포함되었을 경우에 어떻게 변환되는지에 대한 설명이 없다. 우리는 같은 문제를 풀려고 했고, 일반적인 tableau method로 한정자를 다루는 방법을 찾기 못했기에, HDTL이라는 새로운 시제논리를 제안하고, 새로운 분석방법을 제시하였다. 따라서 이렇게 구체적인 언급이 없는 것은 그 가능성을 회의하게 한다. 실제로 저자들에게 문의한 결과, 이 부분에 대해서는 논문에 문법만 제시했을 뿐, 의미와 수행에 관한 이론/구현은 만들지 못한 것으로 확인하였다. 이런 면에서 우리는 MOTEL 연구가, 이론적인 뒷받침이 부족한 것으로 평가한다.

우리의 연구는 다음과 같은 접근방법을 취한다.

우선 우리는 MOTEL에서와 같이, 모니터링을 위한 미리 정해진 추상화 수준을 제공한다. 우리의 결정은, 자바의 메소드 호출 사건들을 체크하는 것이다. 따라서 시스템의 event trace는 다음과 같은 정보를 가지게 된다.

sender, receiver, method-id [, 1^{st}-arg, 2^{nd}-arg, …]

또한 우리는 별도의 instrumentation을 위한 도구를 사용하지 않고, 자바에서 기본적으로 제공하는 JPDA를 사용하고자 한다. 이는 모니터링 프로그램을 만들기 위한 표준적인 방법이라는 의미가 있다. 우선 얻을 수 있는 장점은 어떤 형태의 instrumentation도 필요없다는 점이 있다.

JPDA를 사용함으로써 얻을 수 있는 다른 장점은, JPDA가 특정한 도구가 아니라, 하나의 명세(specification) 이라는데 기인한다. Sun이 발표한 자바 명세에 맞는 자바가상기계(JVM)가 여러가지 존재할 수 있는 것과 마찬가지로, JPDA도 현재 Sun에서 JDK와 함께 제공하는 것 외에도 다른 것이 존재할 수 있다. 그럴 경우, 여러 JPDA의 구현을 모두 사용할 수 있고, 그 중에 성능이나 편리성 면에서 우수한 것을 선택할 수 있다. 또 한 가지 장점은 JPDA의 명세가 발전되어 나감에 따라, 더 다양한 기능이 추가될 것을 기대할 수 있다는 점이다. 예를 들어, 현재의 JPDA에는 실시간성과 관련된 부분이 생략되어 있다. 후에 JPDA가 확장되어 이 부분이 추가된다면, 우리의 방법론은 그 결과를 쉽게 이용할 수 있다.

마지막으로, 우리의 연구는 HDTL을 특성명세언어로 사용한다. 이는 LTL에 대한 확장이라는 점에서는 MaC에서 제안하는 MEDL과 같지만, 그 확장이 기존의 전통적인 시제논리에서 크게 벗어나지 않는다는 점에서 다르다. 즉 HDTL은 변수, 대입(assignment) 등의 메모리 중심 언어적인 특징을 가지지 않으므로, 모니터링 외에도 모델검사나 추론 등의 방법과도 함께 사용될 수 있다. HDTL은 LTL보다 높은 표현력을 가지고, 객체 개념을 쉽게 표현할 수 있다. 또한 우리는 정형명세 시스템인 MAUDE로 HDTL의 의미를 정의하였다. 따라서 HDTL은 확실한 정형적 의미를 가진다고 할 수 있다. 현재 우리는 HDTL을 사용하는 모델체킹 기법에 대하여 연구를 수행 중이다.

우리의 방법과 다른 연구들을 비교해 보면, 표 1과 같다.

표 1 비교

	MaC	Java PathExplorer	MOTEL	Ours
theoretical integrity	high	high	low	high
standard language	no	yes	yes	yes
layer(s)	two	two	one	one
standard technology	no	no	CORBA	JPDA
dynamism	no	no	no	yes

2.2 JPDA(Java Platform Debugger Architecture)

JPDA[8]는 Java2 SDK ver. 1.3 부터 포함된 기술로 자바 프로그램을 위한 디버거, 모니터, 혹은 유사한 프로그램을 제작하는 것을 지원하기 위한 기술이다. 이를 이용하면, 사용자는 자바 가상기계의 동작을 여러 가지 방법으로 관찰하거나, 동작에 영향을 줄 수 있다. 예를 들어, 코드의 특정 위치에 도달했을 때, 수행을 일시 정지시키고 디버거 프로그램으로 제어를 넘겨, 현재의 상태를 관찰하거나, 상태를 변경시킴으로써 수행을 변화시키는 등의 일이 가능하다. JPDA는 3개의 레벨에서 프로토콜과 API를 정의함으로써 디버거 혹은 프로파일링을 지

원하는 도구를 개발하기 위한 프레임웍을 제공한다.

JPDA는 다음과 같은 세가지 부분으로 구성된다.

1. Java Virtual Machine Debug Interface(JVMDI): Java Virtual Machine 이 지원해야 할 부분. JVMDI를 지원하는 JVM (Java 2 SDK 등)은 디버거의 back end를 지원하게 된다. 이 부분이 VM에 질문하고 제어하게 된다. 따라서 이 부분이, 디버거의 front end와, 공유메모리 혹은 소켓 등을 이용하여 통신하게 된다.
2. Java Debug Wire Protocol (JDWP): front end와 back end 사이의 통신 프로토콜을 정의한다.
3. Java Debug Interface (JDI): 유저 코드 수준에서 사용자가 사용할 정보와 요구 등을 정의하는 100% 자바 인터페이스이다. 이는 front end에서 구현된다.

바로 JVMDI나 JDWP를 사용할 수도 있지만, 일반적으로는 JDI 를 사용해서 모든 디버거를 구현할 수 있다. 우리는 JDI에서 정의된 인터페이스를 사용하여, 자바 프로그램에서 수행되는 메소드 호출들을 사건(event)으로 수집(logging)하여 검사하였다.

2.3 HDTL

HDTL은 동적 시스템의 요구사항 기술을 위한 시제논리이다. 이는 기존의 시제논리 LTL에 동결 한정자(freeze quantifier)를 추가함으로써, 시스템의 동적 구성요소들을 지시할 수 있도록 한 확장이다. HDTL에 대한 자세한 사항은 [6]에 기술되어 있다. 이 장에서는 간단한 구문과 의미에 대한 소개만을 제시한다.

우리는 동적 시스템의 동작(behavior)를 이벤트의 순차(sequence)로 정의한다. 그리고 각 이벤트는 메시지 송신자(sender), 수신자(receiver), 그리고 레이블(label)의 세가지 구성요소를 가진다. 그러한 시스템에 대해, HDTL 구문은 다음과 같다.

term := snd(x) | rcv(x) | label(x) | *l*

formula := $t_1 = t_2$ | false | f_1 => f_2 | O f | f_1 U f_2 | x.f

HDTL식을 이루는 기본적인 term은 메시지의 레이블(l), 혹은 변수들의 값에서, 각 구성요소들을 추출한 것이 된다. 변수들은 동결 한정자("x.") 에 의해, 이벤트들에 바인딩 된다. 예를 들어 다음과 같은 HDTL 식을 보자.

x.(label(x) = "borrow")

이 식은 식의 최외곽에 동결 한정자를 가진다. 따라서, 한 이벤트 순차가 주어졌을 때, 그 순차의 맨 처음 이벤트가 변수 x에 바인딩 된다. 그러므로 이 식은, 이 벤트 순차의 맨 처음 이벤트의 레이블이 "borrow" 인 경우 참이 된다. 거기에 다음과 같은 식으로 식이 변형되면,

◇x.(label(x) = "borrow")

이는 식의 맨 외곽에 eventually 시제 연산자가 붙어 있으므로, 이벤트 순차 중에 언젠가 레이블이 "borrow" 인 이벤트가 나타나면 참이 된다. 이를 이용하여 예를 들면 다음과 같이 동적 시스템에 대한 특성을 기술할 수 있다.

□x.(label(x) = "request" → ◇y.(label(y) = "ok" ∧ snd(y) = rcv(x) ∧ rcv(y) = snd(x)))

이는 request 라는 메시지를 받으면, 언젠가 그 메시지를 보낸 쪽으로 ok 라는 메시지를 보내야 한다는 특성을 기술한 것이다. 여기서 두 메시지의 송신자와 수신자를 비교함으로써, 두 메시지가 서로 관계 있는 것임을 기술하고 있다.

3. MAUDE를 이용한 HDTL 체커

MAUDE[7]는 rewriting logic과 equational logic을 모두 지원하는 대수 명세 (algebraic specification) 지원 시스템이다. MAUDE 시스템은 초당 수백만 이상의 개서(rewriting)을 수행하는 효율성과, 메타언어(meta-language)적인 특징으로 인해, 여러 언어(논리 언어, 계산 모델, 정리증명, 프로그래밍 언어 등)에 대한 수행환경으로 많이 사용되고 있다. 이 논문에서는 equational logic에 대한 개서만을 사용한다.

개서는 등식(equation)을 다루는 강력한 기법이다. 이를 위해서는 개서규칙 (rewrite rules) 들이 필요하다. 규칙들은 등식의 형태를 띄고 있으나, 방향성을 가진다. 즉 한 수식은 반복되는 개서를 통해 정규식(normal form)에 이르기까지 변형되고, 그 후에는 변하지 않는다. 따라서 개서규칙이 항상 수식들을 유한한 변환단계를 거쳐 정규식으로 변환시키는지 (terminating), 그리고 그 하나의 식에 대해 유일한 정규식만이 존재하는지 (Church-Rosser)가 개서규칙의 중요한 특성이 된다.

예를 들면, propositional logic에 대해서 우리는 다음과 같은 MAUDE 프로그램(개서규칙)을 작성할 수 있다. 이 프로그램은 Hsiang[9]에 의해 모든 propositional logic 수식을 유일한 정규식으로 유한한 단계로 변환시킨다는 것이 증명되었다.

```
fmod PROPOSITIONAL-CALCULUS is

  sort Formula .
  subsort Qid < Formula .

  ops true false : -> Formula .
  op _/\_ : Formula Formula
```

```
        -> Formula [ assoc comm prec 15 ].
  op _++_ : Formula Formula
        -> Formula [ assoc comm prec 17 ].

  vars X Y Z : Formula .
eq true /\ X = X .
  eq false /\ X = false .
  eq X /\ X = X .
  eq false ++ X = X .
  eq X ++ X = false .
  eq X /\ (Y ++ Z) = X /\ Y ++ X /\ Z .

  op _\/_ : Formula Formula
        -> Formula [assoc prec 19] .
  op !_ : Formula -> Formula [prec 13] .
  op _->_ : Formula Formula
        -> Formula [prec 21] .
  op _<->_ : Formula Formula
        -> Formula [prec 23] .
  eq X \/ Y = X /\ Y ++ X ++ Y .
  eq ! X = true ++ X .
  eq X -> Y = true ++ X ++ X /\ Y .
  eq X <-> Y = true ++ X ++ Y .
endfm
```

MAUDE의 프로그램은 module 단위로 구조화된다. equational logic을 위한 MAUDE 모듈은 "fmod <name> is <body> endfm" 의 형태를 띈다. 그리고 body에는 여러 정의들이 들어가는데, 여기에는 sort, operation, variables, rewrite rules 등이 포함된다.

이 프로그램은 propositional logic에 대한 개서를 수행한다. 즉 모든 tautology 식들에 대해서는 true 값으로 개서하고, 그외의 다른 수식들은

AND('/\')와 XOR('++') 만으로 이루어진 정규식으로 변환시킨다.

QID는 MAUDE에 기본적으로 포함되는 모듈로서, 'a, 'b, 'id 등 quatation mark로 시작하는 상수를 정의하는 모듈이다. 연산자들은 sort 들간의 관계와 의미를 정의하는 개서규칙으로 정의된다. assoc, comm, prec 등은 정의하는 연산자들의 특성을 정의하는 것으로, assoc 은 그 연산자가 결합법칙을 만족함을, comm은 교환법칙을 만족하는 것을 나타내고, prec n은 그 연산자의 우선순위(n)를 나타낸다. 위와 같은 프로그램이 있을 때, 다음과 같은 개서를 수행하여 식을 정규형태로 변환할 수 있다. ('red'는 'reduce'의 약자로, 개서규칙들을 적용하라는 MAUDE 명령이다.)

```
red 'a -> 'b /\ 'c <-> ('a -> 'b) /\ ('a
      -> 'c) . ***> true
red 'a <-> ! 'b .                    ***> 'a ++ 'b
```

이상의 프로그램은 propositional logic의 의미를 MAUDE로 나타낸 것이다. 이를 확장하여 HDTL과 이벤트 트레이스간의 일치관계를 구하는 MAUDE 프로그램을 작성할 수 있다. 이를 위해서는 우선 HDTL의 정의에 따라서 각 연산자들([],<>, 동결 한정자, 등)을 MAUDE로 정의하는 것이 필요하다. 일부만 나타내보면 다음과 같은 식이 된다.

```
ops ([]_) (<>_) : Formula -> Formula [prec 11].
op _U_ : Formula Formula -> Formula [prec 14].
op o_ : Formula -> Formula [prec 11] .
op Frz : Variable Formula -> Formula .
```

여기서 always([]), eventually(<>), next(o) 연산자는 인수를 하나 가지는 것으로 정의되었고, until(U), 동결 한정자(Frz) 연산자는 두개의 인수를 가지는 것으로 정의되었다. 이때, 밑줄(_) 기호는 인수가 들어가야 할 자리를 나타내는데 사용되었다.

MAUDE를 이용해, 이벤트 트레이스의 검사를 수행하기 위해서, 우리는 [10]에서 LTL에 대해 수행한 것과 같이, 이벤트가 입력되는대로, 검사해야 할 HDTL 식을 바꿔가는 방식을 취했다. 이는 예를 들면, 다음과 같은 개서규칙에 의해 정의된다.

```
eq Consume([] X, E) = [] X /\ Consume(X, E).
```

이 규식은 하나의 이벤트 E 가 들어왔을 때, 그 다음 이벤트에서 체킹해야 하는 LTL식 []X 가 어떻게 바뀌는지를 보여준다. 즉 다음 이벤트에서는 여전히 원래의 식 [] X 를 만족시켜야 하고, 거기에 현재 이벤트 E는 식 X를 만족시켜야 한다. 따라서, []p 라는 식이 있을 때, q 라는 이벤트가 들어오면 Consume([] p, q) => [ ]p ∧ Consume(p, q) => []p ∧ false => false 와 같은 순서로 개서되어, 에러를 알리게 된다.

```
eq  Consume(Frz(V, X), E)
      = Consume(Replace(X, V, E), E) .
```

위 식은 동결 한정자가 있을 때, 식이 어떻게 변환되는가를 나타낸 것이다. Replace(X, V, E)는 식 X내의 변수 V의 값을 E로 치환하라는 연산자이다. 그 의미는 별도로 재귀적으로 정의된다. 따라서, 이러한 변환에 의해 동결 한정자의 의미에 따라 HDTL 식이 변환되면서, 입력으로 주어진 이벤트 트레이스가 HDTL 명세와 일치하는지를 검사하게 된다.

이렇게 변환하는 과정에서, 위의 [] p와 같은 경우처럼, 중간에 false 로 판명되면, 이 이벤트 트레이스는 주어진 HDTL 식을 만족시키지 않는 것으로 판단할 수 있다.

이상의 과정을 그림으로 나타내면 그림 1과 같다. 즉 초기에 검사하고자 하는 HDTL 요구명세 F0가 주어졌

을 때, 그 식을 사건이 입력될 때마다, 원래 식을 만족시키기 위해 그 다음 상태에서 만족해야 하는 식으로 변환시켜 가면서 검사하게 된다.

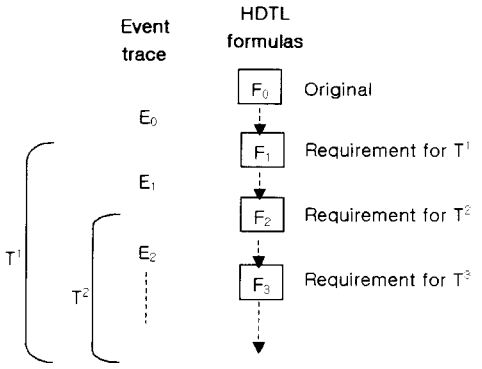

그림 1 HDTL을 이용한 트레이스 검사 과정

4. 병렬 Java 프로그램의 검사

이 장에서는 병렬 자바 프로그램의 수행 모델과, 우리가 수행한 실험에 대해 기술한다.

병렬 자바 프로그램을 만들기 위해서 사용되는 것은 스레드(thread) 클래스이다. 시스템을 구성하는 능동적 객체(active objects)들을 만들기 위한 클래스는, 스레드 클래스를 상속하게 된다. 그리고 그 클래스는 자신의 수행 스레드를 위한 run 메소드를 정의하게 된다. 이 클래스의 객체가 생성되면, 새로운 스레드가 생성되고 그 스레드가 run 메소드를 수행하게 된다.

능동적 객체간의 통신은 메소드 호출의 형태로 행해진다. 객체 A가 객체 B의 메소드를 호출할 경우, 객체 A의 스레드 위에서 객체 B의 메소드가 수행되게 된다. 이를 통해서, 객체 A는 객체 B에게 데이타를 전달 할 수 있고, 객체 B의 상태를 변화시킬 수 있다. 따라서 본 연구에서는 이러한 메소드 호출을 단위로 하여 시스템의 수행을 모니터링하고, 특성을 기술한다. 따라서 모니터링의 대상이 되는 이벤트 순차에 속하는 각 이벤트들은, 다음과 같은 세가지 정보의 모음이 된다.

< caller-object, callee-object, method-id >

이렇게 모니터링하는 것은, 시스템의 수행을 관찰할 때, 메소드보다 낮은 단위의 수행은 무시함을 의미한다. 즉, 메소드 내부의 상세한 사항은 무시하고, 시스템에서 메소드들이 어떤 순서로 호출되는지 만을 살펴봄으로써, 의미있는 요구사항을 검사할 수 있다고 가정하는 것이다. 이는 다시 말하면 시스템이 적정한 수준까지 모듈화 되어 있어야 한다는 요구이다. 즉 하나의 메소드는 가능하면 하나의 기능만을 수행하도록 설계되어야 한다. 그렇지 않다면, 메소드 호출을 단위로 모니터링하여서는 아무런 의미있는 정보를 얻을 수 없게 된다.

예를 들어, 극단적으로 시스템이 하나의 메소드로 구성된 경우, 그 메소드가 수행되었다는 것은 아무런 정보가 되지 못한다. 반대편의 극단으로, 모든 코드들이 하나의 메소드를 이룬다면, 메소드를 단위로 모니터링 하는 것으로, 시스템의 모든 코드들이 제대로 된 순서로 수행되는지를 검사할 수 있다. 하지만, 이 경우 메소드 수행에 대한 정보들이 너무 방대해지므로, 검사하는데 자원을 많이 필요하게 된다. 또한 요구사항을 정확하게 기술하는 데 필요한 정보들이 너무 많으므로, 실질적으로 불가능하게 된다.

제시한 방법론을 시험하기 위해, 우리는 간단한 채팅 프로그램을 대상으로, 그 실행을 모니터링하고, 생성된 트레이스가 HDTL 명세를 만족시키는지 검사하였다.

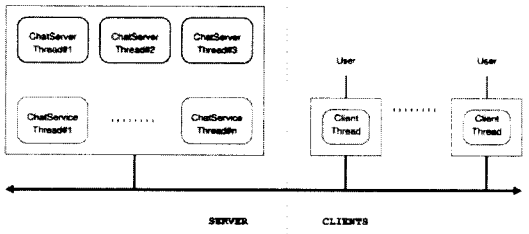

그림 2 예제 : 채팅 시스템

위 그림은 예제 시스템인 채팅 프로그램의 구조를 나타낸다. 이 시스템은 클라이언트-서버 구조를 가지는데, 클라이언트는 자바 애플릿의 형태로 사용자와의 UI를 담당하며, 서버에 정보를 보내고 받으며 수행된다. 서버쪽에서는 들어오는 요구들을 처리한다. 이를 위해, 서버쪽에서는 접속하는 사용자 각각에 대해, 하나씩의 스레드를 만들어 요구를 처리한다. 또한 사용자들을 연령별로 3개의 그룹으로 나누어, 각각의 서버 스레드에서 처리하도록 하였다. 이러한 특성으로 인해, 우리는 서버쪽을 모니터링 하도록 하였다. 서버쪽에서 동적으로 스레드를 만들어 수행하므로, 기존의 정적 시스템을 가정한 모니터링 방법(예를 들어, MaC)으로는 시스템의 동작이 제대로 되고 있는지를 정확하게 구별할 수 없게 된다.

서버를 구성하는 스레드는 두가지이다. 하나는 ChatServerThread 로, 연령별로 하나씩 생성되어 사용자의 접속요구를 처리하게 된다. 또 하나는 ChatService

Thread 로, 접속하고자 하는 사용자가 있을 때마다 생성된다. 이는 클라이언트의 요구를 처리하고, 메시지를 보내는 역할을 한다. 즉 서버쪽에서의 클라이언트의 대변자라고 할 수 있다.

객체간의 통신은 이들 스레드 사이의 메소드 호출로 일어난다. 예를 들어, 클라이언트 A가 다른 클라이언트 B에게 메시지를 보낸다고 하면, 이는 A 클라이언트를 나타내는 ChatServiceThread 객체의 run 메소드 부분에서, B 클라이언트의 ChatServiceThread 객체의 메시지 함수를 호출하는 것으로 나타난다.

JPDA에서 정의하는 옵션을 사용하여 서버쪽을 수행시키고, 그와 병렬적으로 모니터링 프로그램을 수행시키면, 서버에서 일어나는 일들이 트레이스의 형태로 저장되게 된다. 이를 MAUDE로 작성된 HDTL 체커의 입력으로 사용하여 특성이 만족되는지를 검사하였다. 실제로 생성된 트레이스 로그는 다음과 같은 형태가 된다.

```
Evt(P('null), P('myChat.ChatServerThread@23), Msg('run))
Evt(P('myChat.ChatServerThread@23),
P('myChat.ChatServiceThread@44), Msg('init))
Evt(P('myChat.ChatServerThread@23),
P('myChat.ChatServiceThread@44), Msg('startClient))
Evt(P('null), P('myChat.ChatServiceThread@44), Msg('run))
Evt(P('myChat.ChatServiceThread@44),
P('myChat.ChatServiceThread@44), Msg('sendMessage))
Evt(P('myChat.ChatServerThread@23),
P('myChat.ChatServiceThread@44), Msg('getInRoom))
Evt(P('myChat.ChatServerThread@23),
P('myChat.ChatServiceThread@44), Msg('getMyName))
Evt(P('myChat.ChatServerThread@23),
P('myChat.ChatServiceThread@44), Msg('newUser))
Evt(P('myChat.ChatServiceThread@44),
P('myChat.ChatServiceThread@44), Msg('sendMessage))
Evt(P('myChat.ChatServiceThread@44),
P('myChat.ChatServiceThread@44), Msg('stringToBoolean))
Evt(P('myChat.ChatServiceThread@44), P('myChat.Room@52),
Msg('init))
Evt(P('myChat.ChatServiceThread@44), P('myChat.Room@52),
Msg('getUsers))
Evt(P('myChat.ChatServiceThread@44),
P('myChat.ChatServiceThread@44), Msg('exitARoomW))
Evt(P('myChat.ChatServiceThread@44),
P('myChat.ChatServiceThread@44), Msg('sendMessageAll))
Evt(P('myChat.ChatServiceThread@44),
P('myChat.ChatServiceThread@44), Msg('sendMessage))
Evt(P('myChat.ChatServiceThread@44),
P('myChat.ChatServiceThread@44), Msg('newUser))
...
```

이는 MAUDE의 입력으로 사용하기 위해 간단한 전처리기(preprocessor)에 의해 가공된 형태이다. 한 트레이스는 위와 같이 Event의 리스트로 나타난다. 여기서 Evt, P, Msg 등은 event, process id, message id 들을 나타내는 constructor 이다. 즉, 다음과 같은 이벤트는

```
Evt(P('myChat.ChatServerThread@23), P('myChat.Chat
ServiceThread@44), Msg('newUser))
```

myChat 이라는 패키지의 ChatServerThread라는 클래스의 객체 중, 23이라는 ID를 가지는 객체가, Chat ServiceThread 클래스의 ID 44인 객체에게 newUser 라는 메소드를 호출하는 사건을 나타낸다. (자바의 객체들은 모두 유일한 ID를 가지고, 그 값은 uniqueID 라는 메소드에 의해 알 수 있다.)

실험에서 사용한 특성 명세는 다음과 같다. 이는 '한 사람이 한 채팅룸에 들어가면, 그 방에서 나오거나 방의 주인에 의해 쫓겨나지 않는 한, 다른 채팅룸에 들어갈 수 없다' 라는 의미를 가진다.

```
F0 = []x.(label(x)='enterRoom') ->
    o( ! y.(label(y)='enterRoom' /\
    rcv(x)=rcv(y) )
            U z.((label(z)='exitARoom' \/
            label(z)='kickOut')   /\
            rcv(z)=rcv(x) ) ) )
```

이 명세는, 일단 'enterROOM' 라는 메소드가 호출되면, 같은 객체의 (즉 "rcv(z)=rcv(x)") 다른 메소드 'exitARoom'나 'kickOut'가 호출되기 전까지는, 같은 객체의 'enterRoom' 가 호출될 수 없다는 것을 의미한다. 우리가 3장에서 구현한 이벤트 체커는 이런 HDTL명세를 입력으로 받아, 이벤트 순차와 비교하여 그것이 이 명세를 위배하지 않는지 판단한다.

예를 들어, 이런 명세를 위배하는 트레이스가 있다고 할 때, 그것을 어떻게 검사하는지 그 과정을 살펴보자. 다음과 같은 두 이벤트 E0, E1가 연달아 들어오면 이는 위의 제약조건을 위배하게 된다.

```
E0 = Evt(P('myChat.ChatServiceThread@20),
P('myChat.Room@52), Msg('enterRoom))
E1 = Evt(P('myChat.ChatServiceThread@44),
P('myChat.Room@52), Msg('endterRoom))
```

위에서 정의한 명세 F0는 이 이벤트들이 들어옴에 따라, 다음과 같이 바뀐다. (그림 1.) 우선, 처음에 F0는 Consume(F0, E0)로 바뀐다. 이것이 다음에 검사할 요구사항 명세가 된다.

```
F1 = Consume(F0, E0)
   = ! y.(label(y)='enterRoom' /\
   rcv(x)=rcv(y) )
```

```
U z.((label(z)="exitARoom" \/
label(z)="kickOut")   /\
rcv(z)=rcv(x) )
```

이는 다시

F2 = Consume(F1, E1)으로 바뀌게 된다. 그런데 이 때, 3장에서 기술한 HDTL 체커를 사용하면, Consume (F1, E1) = false 임을 알 수 있게 된다. 따라서 우리는 E1 이 입력되는 시점에서, 이 트레이스는 원래의 HDTL 요구사항을 위배하고 있음을 알게 된다.

2장에서 살펴본 바와 같이, 기존의 정형적 명세 기반의 모니터링 시스템으로는 이러한 특성을 제대로 기술할 수 없다. 예를 들어 MaC 같은 경우, A라는 객체가 enterRoom이라는 메시지를 수신하는 이벤트와, B라는 객체가 enterRoom 메시지를 수신하는 이벤트를 분간하지 않으므로, 이런 명세를 기술할 수 없다. HDTL은 이 두 이벤트를 분리하고, 여러 이벤트간의 송신자, 수신자의 ID를 비교할 수 있게 하여, 동적 시스템의 수행을 모니터링할 경우 필요한 정형적 요구사항을 상세히 기술하도록 해 준다. 우리는 MAUDE로 작성된 HDTL 검사 프로그램을 통해, 수집된 자바 프로그램의 이벤트 트레이스가 위의 명세를 만족함을 확인할 수 있었다.

5. 결 론

시스템을 이루는 정형 명세를 사용하여 시스템이 반드시 만족하여야 할 요구사항을 기술하였다고 하더라도, 최종적인 구현이 그 요구사항을 만족하는지를 보장하는 것은 쉬운 일이 아니다. 따라서 본 연구에서는 그러한 한가지 방법으로서, 실제 구현이 수행되는 런타임에 정형명세와 구현이 일치하는지를 알아볼 수 있는 방법을 제시하였다.

이를 위해 기존에 동적 시스템을 위해 제안하였던 HDTL을, 구현언어인 JAVA 프로그램에 대한 정형적 요구사항 기술언어로서 사용하였다. 이를 통해, 기존의 정형 명세 기반 모니터링에 대한 연구들과 달리, 동적으로 시스템의 구성이 바뀌는 실제적인 시스템에 대한 요구사항도 기술하고 검사할 수 있게 되었다.

한 이벤트 트레이스가 HDTL 명세를 만족하는지를 검사하기 위해, 우리는 MAUDE라는 개서 시스템을 사용하였다. MAUDE로 HDTL의 의미를 기술하고, 이벤트 트레이스와 HDTL 명세의 일치 여부를 개서규칙을 통해 정형적으로 기술함으로써, 간단하고 효율적인 체킹 알고리즘을 구현할 수 있었다.

향후 가능한 연구방향의 하나는 테스트 오라클로서의 사용이다. 현재 일반적인 테스팅에서는, 테스트 케이스를 수행시켰을 때, 시스템이 제대로 수행되었는지를 판단하는 오라클을 사용하는 경우가 아직 많지 않다. 이는 오라클을 기술하는 과정의 어려움과 오라클 기술을 위한 적당한 언어가 존재하지 않기 때문이다. 우리의 연구와 같이, 런타임 모니터링과 체킹을 수행하는 여러 연구들은, 동시에 병렬 프로그램에 대한 테스트 오라클로서 기능할 수 있을 것으로 생각된다. 이 경우, HDTL의 표현력이 충분한지의 문제, 그리고 기존의 테스팅 프로세스에 어떻게 결합되어야 하는가의 문제 등을 해결해야 한다.

참 고 문 헌

[1] A. Pnueli, "The temporal logic of programs," Proc. 18th IEEE Symposium on Foundation of Computer Science, 1977.

[2] M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan, "Java-MaC: a Run-time Assurance Tool for Java Programs," Proc. First Workshop on Runtime Verification (RV'01), Paris, France, 23 July 2001

[3] Klaus Havelund, Grigore Rosu, "Monitoring Java Programs with Java PathExplorer," Proc. First Workshop on Runtime Verification(RV'01), Paris, France, 23 July 2001

[4] Dietrich, F., Logean, X., Koppenhofer, S. & Hubaux, J.-P., "Modelling and testing object-oriented distributed systems with linear-time temporal logic," Technical Report SSC/1998/011, Swiss Federal Institute of Technology, Lausanne.

[5] Laura K. Dillon and Y. S. Ramakrishna, "Generating oracles from your favorite temporal logic specifications," Proceedings of the Fourth ACM SIGSOFT Symposium on Foundations of Software Engineering, Oct. 1996.

[6] Seung Mo Cho, Hyung Ho Kim, Sung Deok Cha and Doo Hwan Bae, "Specification and Validation of Dynamic Systems using Temporal Logic," IEE Proceedings-Software, Vol. 148, No 4, Aug. 2001.

[7] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada, "Maude: Specification and Programming in Rewriting Logic," Maude System documentation, March, 1999.

[8] http://java.sun.com/j2se/1.3/docs/guide/jpda/

[9] Hsiang, J., "Refutational Theorem Proving using Term Rewriting Systems," PhD thesis, University of Illinois at Champaign-Urbana, 1981.

[10] Cohen, S., Jtrek. Compaq, http://www.compaq.com/java/download/jtrek.

조 승 모

1994년 한국과학기술원 전산학 학사. 1996년 한국과학기술원 전산학 석사, 2002년 한국과학기술원 전산학 박사. 현재 한국과학기술원 첨단정보기술연구센터(AITrc) 연수연구원. 관심분야는 정형기법, 테스팅

김 형 호

1996년 서강대 전산학 학사. 1998년 한국과학기술원 전산학과 석사. 현재 한국과학기술원 전산학과 박사과정 재학 중. 관심분야는 컴포넌트 기반 소프트웨어 공학, 객체지향 기술, 소프트웨어 메트릭

차 성 덕

1983년 University of California at Irvine 전산학 학사. 1986년 University of California at Irvine 전산학 석사. 1991년 University of California at Irvine 전산학 박사. 1990년 ~ 1991년 Hughes Aircraft Co. 연구원. 1991년 ~ 1994년 The Aerospace Corp. 연구원. 현재 한국과학기술원 전산학과 부교수

배 두 환

1980년 서울대학교 조선공학과 학사. 1987년 위스콘신-밀워키대학 전산학 석사. 1992년 플로리다 대학 전산학 박사, 1992년 ~ 1994년 플로리다대학 전산학과 조교수. 1995년-1996년 한국과학기술원 정보 및 통신공학과 조교수. 현재 한국과학기술원 전산학과 부교수