프로그래밍 언어의 관점에서 본 소프트웨어/하드웨어의 정형 검증 방법 소개†

고려대학교 방기석 · 유희준* · 최진영**

1. 서 론

2000년대에 들어 컴퓨터 프로세서의 성능이 발전하고 연산 능력이 눈에 띄게 향상됨에 따라 산업체 전반에 걸쳐 많은 부분이 컴퓨터에 의해 제어되는 시스템으로 교체되고 있다. 이러한 추세와 더불어, 컴퓨터 시스템의 많은 부분이 소프트웨어화 되고 있다.

하드웨어로 동작되던 부분이 빠른 연산 속도에 의해 소프트웨어에 의해 제어되고, 이에 따라 유연성 및 유지 보수의 향상이라는 장점을 가져 오고 있다. 그러나, 소프트웨어는 개발에 사용하는 프로그래밍 언어의 표현 및 연산 능력에 크게 좌우되고 있으며, 개발자의 경험과 축적된 기술에 의해 성능에서 큰 차이를 가져오게 된다. 특히, 시스템의 크기가 복잡하고 분산화되고 있는 요즘 소프트웨어 시스템의 안정성 및 프로그래밍 언어의 정확성에 대한 논란이 크게 제기되고 있다. 이러한 문제점으로 인해 안정성이 증명된 하드웨어 시스템을 그대로 유지하고 개선하려는 움직임도 있다. 하지만, 하드웨어 역시 프로그래밍 언어를 사용해서 하드웨어 칩의 설계를 하는 것이 최근 동향이다. 특히, 내장형 시스템과 같이 하나의 칩 속에 모든 기능이 탑재되는 SOC 분야에서는 프로그래밍 언어 기술이 매우 중요하다. 내장형 시스템은 그 내부에 하드웨어와 소프트웨어를 모두 갖고 있다. 즉, 설계시에 하드웨어로 설계할 부분과 소프트웨어로 설계할 부분을 나누게 된다. 때문에 각 부분을 설계하기 위한 프로그래밍 언어의 선택과 그 안정성의 증명은 전체 시스템의 성능 및 안정성과 매우 밀접하게 연결되어 있다. 일반적으로 시스템의 안정성을 증명하기 위해 정형 기법[1]을 사용하고 있다. 정형 기법은 수학과 논리학에 기반하기 때문에 표현이 명확하고, 기계적이고 논리적인 증명 절차에 따라 가능한 모든 경우에 대한 증명을 수행하여 신뢰할 수 있는 시스템의 구현에 도움을 준다. 그러나, 정형 검증을 위해 설계하는 명세 언어는 일반적인 프로그래머나 시스템 개발자들에게는 매우 어렵게 느껴지는 특정한 모델링 언어들이다. 따라서, 정형 검증을 위해 프로그래머들의 설계와 동일한 별도의 검증 모델을 구현하고, 이 모델에 대한 안정성 증명을 수행하는 것이 일반적인 절차이다. 이러한 번거로움과 난해함 때문에 산업체에서 정형 기법을 적용하는 것이 매우 큰 어려움으로 남아있다. 결국, 이해하기 쉬운 모델링 언어의 개발 혹은 설계자에 의해 구현된 코드나 설계도에 대한 정형 검증을 가능하도록 하는 연구가 절실하게 필요하다. 이런 필요성에 따라 근래에는 설계자에 의해 구현된 하드웨어 및 소프트웨어 코드를 직접 정형 검증하는 방법에 대한 연구가 전세계적으로 학계 및 산업체에 의해 매우 활발히 진행되고 있다. 마이크로소프트나 인텔, 그리고 NASA와 같은 기관에서는 별도의 연구 과제와 연구 그룹을 결성하여 이미 오래 전부터 연구에 몰두하고 있으며 그 결과 좋은 도구가 개발되어 실제로 자사의 제품에 대하여 적용하고 안정성 검증에 사용되고 있다. 그러나 국내에서는 프로그래밍 언어의 안정성에 대한 중요성은 인식하고 있으나, 산업체에서의 적용은 아직 미비한 수준이다. 이에, 현재 개발되고 있는 안정성 검증 도구와 연구동향을 살펴보고, 국내 컴퓨터 산업에의 적용 가능성을 전망해 보도록 한다.

† 본 연구는 한국과학재단 목적기초연구(R01-2000-00287) 지원으로 수행되었음.

* 학생회원
** 종신회원

2. 컴퓨터 시스템 안정성 보장을 위한 방법

컴퓨터 시스템은 크게 하드웨어와 소프트웨어 부분으로 나눠볼 수 있다. 하드웨어와 소프트웨어 시스템이 유기적으로 연결되어 하나의 시스템으로 동작하는 것이 컴퓨터 시스템이다. 따라서, 두 부분 중 어느 한 곳에서라도 오류가 발생한다면 그것은 전체 시스템에 이상을 가져오게 된다. 따라서, 컴퓨터 시스템의 안정성을 보장하기 위해 시뮬레이션과 테스팅[2] 같은 기법을 예전부터 많이 사용하고 있다. 일반적으로 시뮬레이션 혹은 테스팅은 시스템 설계자들이 특정한 시스템을 설계/구현한 후 시스템이 사용자의 요구사항을 만족시킬 수 있는지를 확인하기 위해 특정한 동작 시나리오를 설정하고, 그 시나리오에 따라 구현한 시스템을 동작시켜 원하는 결과를 가져오는지를 보는 방법을 사용한다. 그렇기 때문에 얼마나 많은 시나리오를 사용하는가가 테스팅에 있어서 그 정확성을 결정하는 매우 큰 요인이 된다. 즉, 가능한 한 많은 수의 테스팅 시나리오를 설정하고 여러 번 반복해서 테스팅을 수행함으로써 더욱 정확한 결과를 산출해 낼 수 있는 것이다. 그러나, 사람이 만들어 낼 수 있는 시나리오에는 한계가 있으며, 아무리 많은 수의 테스팅을 수행한다 하더라도 그것이 시스템의 안정성을 100 % 보장한다고는 말할 수 없는 것이다. 따라서, 근래에는 최대한 많은 수의 시나리오를 자동으로 생성해 내기 위한 연구가 많이 진행되고 있다.

하지만 그것 역시 한계는 분명 존재한다. 이러한 한계를 극복하기 위해 연구된 것이 정형 기법[1]이다. 정형 기법은 테스팅과 달리 시스템을 수학/논리학에 기반한 표현을 이용하여 논리적으로 명세한다. 이렇게 명세된 시스템을 역시 논리적인 절차에 따라 기계적인 증명을 수행하여 시스템의 정확성을 보장하게 한다. 정형 기법은 논리에 기반하기 때문에 표현이 명확하고, 수학적 증명 단계를 거치기 때문에 증명 결과에 대해서는 확실히 믿을 수 있다. 이러한 정형 기법에 대해 간단히 알아보도록 한다.

정형 기법은 컴퓨터 시스템을 개발하기 위해 수학적으로 분석하고 설계하는 기술을 의미한다[3]. 이 정의에서 말하듯이 정형 기법은 수학과 논리학에 기반을 둔 방법으로 하드웨어나 소프트웨어 시스템을 명세하고 시스템이 만족해야 할 특성 역시 수학적인 논리로 표현하여 시스템이 특성을 만족하는지를 증명한다. 이러한 정형 기법은 수학적 모델을 설명하기 위한 문법, 그리고 문법의 의미를 설명하는 의미론 및 의미 관계를 포함한다. 따라서 정형 기법은 자연어가 내포할 수 있는 애매모호함이나 불확실성을 최소한으로 줄일 수 있다. 또한 설계된 시스템이 처음에 의도된 요구사항과 동일한지를 수학적 성질을 이용하여 증명할 수 있기 때문에 개발초기에 큰 실수를 발견할 수 있다. 그러나 정형 기법도 다른 많은 기술들과 마찬가지로 적용에 한계가 있다. 이 한계는 현재의 기술 수준에서 발생하기도 하고 이론적인 한계에서도 야기되기도 한다. 현재 사용되는 정형 기법에서 실제로 문제가 되는 것은 어떻게 실시간적인 특성을 정확히 표현하고 증명을 할 수 있는가 하는 문제이다. 또한 정형 기법의 이론적인 한계는 다음의 두 가지 관점에서 보일 수 있다. 첫째, 실세계와 수학적인 세계의 경계는 무엇인가? 둘째, 수학의 내부적인 한계는 무엇인가? 이러한 질문은 정형 기법이 개발되고 발전하면서 끊임없이 제시되고 있다. 수학적인 기반을 갖고 개발되고 있기 때문에 그 한계는 halting 문제와 같은 수학에서의 한계와 거의 비슷하다. 이러한 한계를 잘 이해하는 것은 정형 기법을 안전에 민감한 시스템에 적용하는데 매우 중요하다.

정형 기법은 크게 정형 명세(Formal Specification)와 정형 검증(Formal Verification)의 두 가지로 구분할 수 있다[4]. 정형 명세[4]는 시스템이 만족해야 할 요구사항과 그 요구사항을 만족할 수 있는 설계를 기술하는 것을 말한다. 정형 검증[4]은 명세가 정확한지, 즉 설계가 요구사항을 만족하는지를 검사하는 것이다.

2.1 정형 명세

정형 기법은 크게 정형 명세와 정형 검증으로 나눠서 생각할 수 있다. 정형 명세는 증명하고자 하는 시스템을 논리적인 언어를 이용하여 모델링하는 것이다. 이 모델은 실제 시스템을 추상화 한 것으로 특정한 명세 언어를 이용하여 시스템의 성질을 그대로 보존하게 된다. 정형 명세는 요구 명세(requirement specification)와 설계 명세(design specification)로 구성된다. 요구 명세는 시스템이 무엇을 해야 하는가를 정의해 놓은 명세로서 간결하게 표현되고, 요구사항을 직접 표현한다. 설계 명세는 요구사항을 만족할 수 있는 설계를 기술하는데 그 목적이 있다. 그러므

로 설계 명세는 시스템의 행위를 구현할 목적으로 작성되며 세부적인 사항을 포함한다.

2.2 정형 검증

정형 검증은 위에서 말한 바와 같이 정형적으로 명세된 시스템의 안정성을 증명하는 방법이다. 일반적으로 명세 자체의 정확성을 검증하거나 명세된 시스템이 일련의 특성을 만족하는지를 증명함으로써 그 시스템의 안정성을 보장하게 된다. 정형 검증에는 크게 자동 증명(theorem proving)과 모델 검증(model checking)이 있다. 자동 증명 방법에서는 시스템 모델과 명세를 적당한 논리를 이용해서 표현하고 시스템 모델이 명세를 만족함을 수학적인 방법으로 증명을 통해서 보인다. 이 방법은 수학과 논리학을 사용한 증명이기 때문에 매우 강력하고 완전하지만 매우 어려워서 전문가가 아니면 이해하기가 어렵고 규모가 큰 프로그램의 경우에는 적용하기가 어렵다. 반면에 모델검증[5]은 시스템의 동작을 유한상태 기계의 형태로 명세하고 그 시스템이 만족해야 할 특성을 CTL[6]이나 LTL[6]과 같은 시제 논리로 표현한다. 그 후에 표현된 특성이 유한상태 기계로 명세된 시스템에서 만족하는지를 검사하는 방법이다. 시제 논리는 일반적으로 자동 증명에서 사용되는 논리에 비해 제한적이지만 간단하고 명확해서 유용하게 사용될 수 있다. 이 방법은 복잡한 시스템의 동작을 유한상태 기계로 표현함으로써 간단하게 표현할 수 있고, 이해하기가 매우 쉽다. 또한 모델검증용 도구가 제공되어 명세된 시스템에 대해 자동으로 검증을 수행할 수 있다. 그러나 이 방법 역시 시스템의 동작이 복잡하고 규모가 큰 경우에는 상태폭발 문제를 야기할 수 있다.

3. 모델 검증 방법을 이용한 컴퓨터 시스템의 정형 검증의 문제점

위에서 말한 바와 같이 정형 기법은 컴퓨터 시스템의 안정성 및 숨겨진 오류를 발견하는데 매우 큰 효과를 보여주고 있다. 하지만 논리 증명 방법은 그 증명 단계나 명세 방법이 매우 어려워서 실제 시스템의 설계자들이 이해하기가 곤란하다. 따라서 이해하고 적용하기 쉬운 모델 검증 방법을 주로 사용하게 된다. 위에서 살펴본 바와 같이 모델 검증은 시스템을 유한 상태 전이 시스템으로 모델링하고 그 모델이 만족해야 할 특성을 시제 논리로 명세하여 모델이 특성을 만족하는지 검증한다. 또한 모델이 특성을 만족하지 못할 경우 모델의 전이를 추적하여 왜 특성을 만족하지 못하는지 반례를 자동으로 생성하여 오류의 수정을 용이하도록 도와준다. 하지만 이런 장점에도 불구하고 현재 모델 검증이 컴퓨터 시스템, 특히 범용 프로그래밍 언어로 설계된 소프트웨어의 검증에 사용되기는 매우 어렵다. 그 이유는 다음과 같이 요약할 수 있다[7].

첫째, 상태 폭발의 문제(state explosion problem)가 야기될 수 있다. 유한 상태 전이 시스템 모델은 원래 시스템의 요소들이 늘어날 경우 지수적으로 그 상태가 증가하게 된다. 이런 상태 폭발 문제를 해결하기 위해 많은 방법이 제기되고 있지만 대부분이 하드웨어 요소들에 대한 기법이며 소프트웨어 시스템의 경우는 하드웨어보다 복잡하여 그 추상화의 문제가 더 심각하다.

두번째 문제는 모델의 구현에 있다. 대부분의 컴퓨터 시스템 개발자는 C, Java와 같은 범용 프로그래밍 언어로 구현을 한다. 하지만 정형 검증을 위한 명세 언어는 그런 범용 언어와는 매우 다른 문법적인 특성 및 의미론을 바탕으로 하고 있다. 따라서, 실제 프로그램을 정형 검증 도구에 적용하기 위해서는 일단 범용 프로그래밍 언어로 구현된 모델을 다시 정형 검증 도구용 입력 언어로 변환시키는 작업이 필수적이다. 이런 변환 작업 도중에 많은 오류가 발생할 수 있고 시간의 낭비가 일어나게 된다.

세번째 문제는 요구 명세의 문제이다. 컴퓨터 시스템의 요구사항을 현재 사용되는 시제 논리를 이용하여 완벽히 표현하는 것은 매우 어렵다. 비록 모델 검증의 명세 언어가 시제논리에 기반하여 매우 논리적으로 정의되어 있지만, 복잡한 이벤트의 특성을 표현하기에는 그렇게 쉽지 않다. 또한 모델 검증의 명세언어는 상태적인 특성을 수학적 모델에 적용하기 위해 개발된 것이기 때문에 실제 프로그램 코드를 설계하기에는 적합하진 않다. 정형 검증 도구의 입력언어들은 대부분 정적인 상태의 변화나 지역적 변수의 변화 같은 특성을 검사하기에 적합하다. 그러나 실제 프로그램 코드에서는 이런 특성 외에도 많은 동적인 특성을 지니고 있기 때문에 정형 검증을 하려는 사용자는 반드시 이런 차이점을 극복할 수 있도록 코드를 추상화해야 한다. 특히, 객체 지향형 소프트웨어의

경우 수행중에 동적으로 생성되는 객체나 쓰레드의 생성과 소멸 같은 특성은 모델링 언어로 표현하기에는 매우 어렵다.

마지막으로 출력 결과의 해석에서 문제가 발생할 수 있다. 모델이 특성을 만족하지 못했을 경우 모델 검증 도구는 오류에 대한 역추적을 가능하도록 하는 반례를 만들어 준다. 그러나 모델의 크기가 매우 클 경우 그 반례 역시 수백 또는 수천 라인으로 증가하게 된다. 이런 큰 추적코드를 수동으로 실제 코드에 적용해서 오류의 위치를 찾아내는 것은 매우 어렵다. 또한 오류 추적 코드는 요약된 모델에서 매우 낮은 수준으로 생성되기 때문에 실제 코드를 추적하는 데엔 많은 차이가 존재한다. 즉, 실제 복잡한 소스 코드의 한 단계의 전이가 요약된 오류의 수십 줄의 단계에 해당하는 경우가 매우 많다. 이런 많은 문제점 때문에 현재 컴퓨터 시스템의 설계 코드를 새롭게 명세하지 않고 코드 자체를 정형 검증하는 연구가 활발히 진행되고 있다. Microsoft의 SLAM 프로젝트[9]나 NASA의 JPF[10], Kansas 주립대학의 Bandera [7, 8] 프로젝트와 같은 경우는 소프트웨어 소스 코드의 정형 검증 도구로 매우 좋은 효과를 가져오고 있다. 또한 Lucent의 Bell 연구소 역시 자체적으로 Feaver [11] 라는 도구를 개발하여 Lucent에서 개발하는 교환기 시스템의 소스 코드를 모두 검증하고 있다. 하드웨어 시스템의 설계는 Verilog[11]와 같은 하드웨어 전문 설계 언어인 HDL을 사용하는 것이 일반적이다. HDL로 설계된 모델을 Verilog나 SMV[5]와 같은 정형 검증 도구를 이용해서 모델 검증을 수행하는데 이 때 역시 위와 같은 문제가 생긴다. 그래서, 차세대 HDL로 사용할 수 있는 SystemC[13]와 같은 범용 언어를 이용해서 하드웨어를 명세하고 이것을 자동으로 정형 검증 도구의 입력 언어로 변환하여 모델 검증을 수행하는 연구가 해외 뿐 아니라 국내에서도 활발히 진행되고 있다.

4. 검증된 시스템 설계 방법

4.1 소프트웨어 설계언어의 정형 검증

4.1.1 Bell Lab의 SPIN과 Feaver

Bell Lab에서는 Lucent에서 개발하는 교환기 및 각종 시스템의 안정성을 보장하기 위해 많은 연구를 수행해 오고 있다. 그 중 가장 눈에 띄는 연구 결과가 SPIN[14, 15]이다. SPIN은 통신 프로토콜과 같은 분산 환경의 소프트웨어 시스템의 정확성 및 안정성을 증명하기 위해 개발된 모델 검증이다. SPIN은 명세하고자 하는 시스템의 동작을 오토마타와 같은 형식으로 모델링한다. 이 때 사용하는 언어가 Promela[14, 15]이다. Promela는 시스템을 프로세스의 형태로 추상화 하고, 프로세스들이 병렬적으로 수행하면서 채널을 통해 서로간에 동기화 및 통신을 수행하게 된다. 이렇게 추상화 된 시스템 모델의 동작을 보기 위해 Promela의 시뮬레이션을 SPIN이라는 모델 검증 도구를 사용해서 수행한다. 그 결과 시스템의 각 프로세스가 동작하는 모습을 화면으로 매우 쉽게 확인할 수 있다. 그리고, 시뮬레이션으로 확인하기 어려운 요구사항이나 특성은 선형시제논리[6]를 이용해서 명세한 후 SPIN에서 제공하는 검증 도구를 이용해서 논리적으로 증명을 하게 된다. 증명 결과 모델링 된 시스템이 특성을 만족하지 못하면 오류가 발생하는 경로를 모두 저장해서 역추적할 수 있는 반례를 생성해 준다. 이 반례에 따라 Promela 모델을 동작시키면 어떤 상황에서 오류를 발생하는지를 쉽게 찾을 수 있고, 그 상태에서 오류 수정이 매우 용이하다. 이 도구는 현재 해외에서 개발된 소프트웨어 검증 도구 중 성능이 우수한 것으로 평가되어 있으며, 통신 프로토콜 뿐 아니라 범용 프로그래밍 언어로 개발된 소프트웨어 및 운영체제, 그리고 분산 시스템 등에서 활발히 적용되고 있다. 그러나, SPIN 역시 소프트웨어 시스템을 그대로 모델 검증하기 위해서 지켜야 할 제약이 존재한다. 위에서 밝힌 바와 같이 Promela라는 특정한 언어를 통해 시스템을 추상화 해야 하고, 이 과정에서 실제 소스 코드에 약간의 제한을 가해야 한다. 또한 범용 프로그래밍 언어로 구현된 시스템을 Promela로 모델링 할 때 모든 것이 완벽히 추상화 되는 것이 아니다. 따라서, 설계자들에 의해 구현된 소스 코드를 검증하기 위해서는 매우 많은 수작업이 필요하다.

이러한 단점을 극복하고, 소스 코드의 검증을 더욱 정확히 하기 위해 Bell 연구소에서는 Feaver[11] 라는 도구를 개발하였다. Feaver는 Lucent에서 개발한 교환기의 정확성을 보장하기 위해 연구되었다. 이 교환기의 안정성을 증명하기 위해 교환기가 만족해야 하는 요구사항들에 대해 정형적으로 명세된 데이타베이스를 구현하고 그 데이타베이스에 저장된 요

구사항들을 만족하는지를 기계적으로 증명하기 위해 Feaver가 개발되었다. 그러나, 증명 과정에서 사람의 개입을 줄이고, 기계적으로 수행하는 것이 매우 중요하다. 이를 위해 Feaver는 설계자가 구현한 C 코드 그 자체를 검증 시스템의 입력으로 한다. 그리고 이 코드로부터 동작 모델을 자동으로 추출하고 그 모델에서 오류가 발생할 수 있는 동작 시나리오를 추출해 낸다. 이 시나리오에 따라 시스템이 동작했을 때 요구사항 데이타베이스에 명시된 요구사항들을 위배하는지를 검증하게 된다. 이 검증에 사용되는 도구가 SPIN이다. 즉, 소스 코드로부터 동작 시나리오를 SPIN의 입력 언어인 Promela로 자동으로 추출해 낸다. 이 코드는 물론 이해하기는 어려울 수 있지만, 소스 코드에서 갖고 있는 거의 대부분의 정보를 포함하기 때문에, 전문가에 의해 모델링된 코드보다도 더 정확한 추상화가 가능하다. Bell 연구소에서는 Feaver 시스템을 ANSI-C 로 구현된 범용 소프트웨어의 정형 검증 도구로 활용하고 있으며, 그 성능도 매우 좋은 것으로 현재 평가받고 있다.

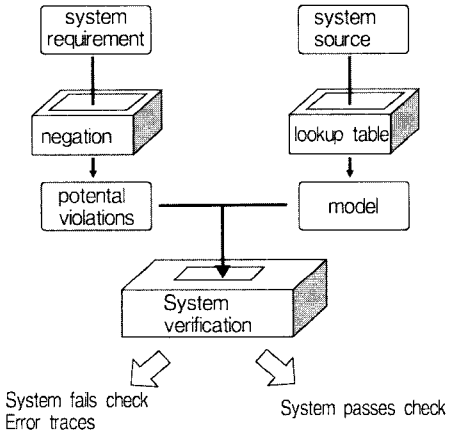


그림 1 Feaver의 동작 과정

Feaver를 이용해서 안정성이 증명된 시스템으로 Lucent의 PathStar™[11]라는 접근 서버 시스템이 있다. Bell 연구소에서는 이 접근 서버의 제어장치의 안정성을 증명하기 위해 Feaver를 사용했다. 이 제어장치는 약 1600 줄 정도의 C 언어로 구현되어 있다. 그리고, 각각 독립적으로 수행하는 여러 프로세스들이 동시에 수행되는 시스템이다. 따라서, 제어 장치의 코드로부터 검증을 위한 추상 모델을 추출하기 위해서는 각각의 프로세스들의 유기적인 동작이 그대로 유지되어야 한다. Bell 연구소에서는 이러한 복잡한 과정을 Feaver를 이용해서 해결하였으며, 현재 검증된 PathStar™ 접근 서버 시스템을 동작시키고 있다. 이러한 안전한 시스템을 구현하는데 대략 18개월이 소요되었다고 한다. 물론 이 기간동안 시스템을 설계하고, 개선하고, 유지보수하는 시간이 모두 포함된다. 그 기간동안 약 300개 이상의 버전이 새롭게 나왔으며 그 크기 역시 다양하게 진화되었다. 이 과정에서 약 75개의 오류를 발견해 낼 수 있었다. 많은 오류가 시스템의 안정성에 영향을 미치는 것이었으며, 전통적인 테스팅 기법 및 전문 테스터들에 의해 몇 가지 오류를 발견할 수 있었다. 그러나 Feaver 시스템을 활용함으로써 테스터들이 발견할 수 없었던 더 많은 오류를 자동으로 찾아낼 수 있었다.

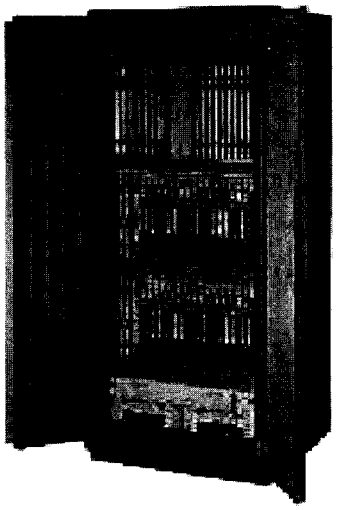


그림 2 PathStar ™

4.1.2 Microsoft의 SLAM 프로젝트

Microsoft에서는 Microsoft에서 제작하는 많은 소프트웨어의 정확성을 검증하기 위해 SLAM Project [9, 16]를 수행하고 있다. SLAM은 C 언어로 설계하는 소프트웨어를 그 대상으로 하고 있다. SLAM 프로젝트에서는 소프트웨어의 안정성 특성 증명을 위한 사용자의 개입이나 추상화 작업을 필요로 하지 않는다. 대신 C 언어로 구현된 코드를 추상화 한 Boolean 프로그램[17]을 자동으로 추출하고, 그 추상 모델이 특성을 만족하는지를 증명하는 방법을 개

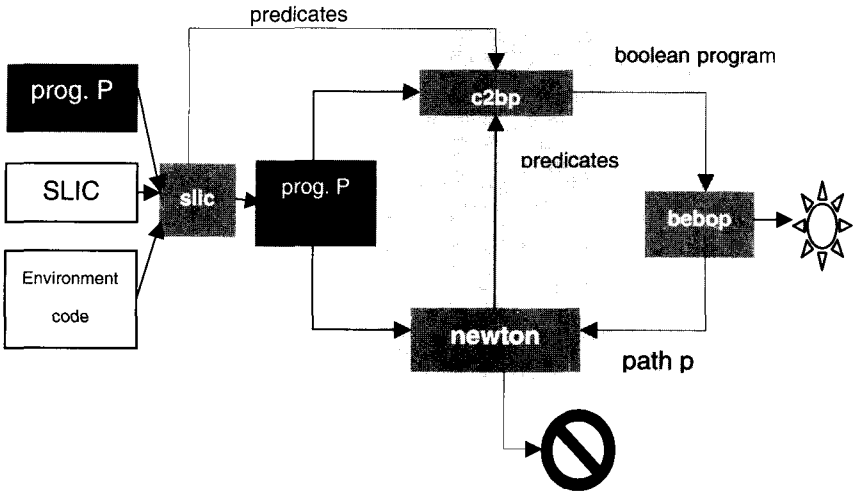


그림 3 SLAM Toolkit의 동작

발하고 있다. SLAM toolkit은 다음의 세 가지 도구를 순차적으로 사용한다.

- C2BP : 프로그램 코드를 분석하여 SLIC[18]이라는 중간 명세 언어로 변환한 다음 이것으로부터 프로그램 코드의 동작을 나타내는 Boolean 프로그램으로 변환시켜주는 도구이다.
- Bebop[19] : 변환된 boolean 프로그램을 자료 흐름 분석 알고리즘을 이용해 순회하여 내부에 갖고 있는 오류 상태를 찾아내는 모델 체커이다.
- Newton : Bebop을 통해 발견된 오류사항을 제거한 안전한 모델을 생성해 주는 도구이다. 이 모델에 따라 오류가 발생하지 않는 C 코드를 재생성할 수 있는 기반이 된다. 위에서 사용되는 SLIC 코드는 C 코드를 분석하여 추상화 한 모델이다. 이 과정은 SLIC 전처리기에 의해 미리 생성된다. SLAM toolkit을 엄밀히 말하면 논리적인 증명 방법을 사용한 검증을 수행하지는 않는다. 다만 Boolean 프로그램을 분석하고 순회하여 오류 상태에 도달할 수 있는 도달 가능성 분석과 같은 방법을 이용하여 그 안정성을 testing 해 보는 방법을 사용하는 것이다. 그러나, 그 과정이 완전히 자동화 되어 있기 때문에 인간의 개입에 따른 오류를 막을 수 있고, 수행에 걸리는 시간적인 낭비를 줄일 수 있기 때문에 매우 좋은 연구로 평가받고 있다.

Microsoft에서는 windows XP를 발표하면서 windows XP 내부에서 동작하는 몇 가지 device driver들을 SLAM toolkit을 이용하여 안정성을 증명하였다. 앞으로도 많은 windows 내부의 프로그램들을 대상으로 SLAM toolkit 을 이용하여 안정성을 확보하기 위해 연구를 진행중이며 곧 SLAM toolkit을 공개할 계획으로 있다.

4.1.3 Bandera 프로젝트

Bandera project[7, 8]는 미국의 Kansas 주립대학의 SAnTOs 연구실을 중심으로 수행되는 것으로 객체지향형 언어인 Java로 구현된 소프트웨어의 정확성을 검증하는 것을 그 목표로 하고 있다. 이 도구는 자바 소스코드를 분석하고 검증언어로 변환하며 모델검증을 하기 위한 통합환경을 제공하고 있다. 입력언어로는 자바 소스코드를 그대로 입력하며, 적용하고자 하는 검증 방법에 맞는 명세 언어를 자동으로 출력해 준다. 이 도구에서는 현재 SPIN, dSPIN[10] SMV 그리고 JPF II(Java Path Finder II)[11]의 입력언어로의 변환이 가능하다. 이 도구는 소스 코드의 변환과 검증이 자동으로 이뤄지기 때문에 사용자의 추가적인 개입이 없다. 결국 코드의 변환과정에서 생길 수 있는 오류를 배제할 수 있다. 또한, 오류의 발생시 역추적을 명세 단계가 아닌 자바 소스 코드의 레벨에서 보여주기 때문에 실제 코드의 수정을 통한 오류 수정이 가능하다. JPF II는 Bandera에서 변환한 검증할 자바 코드와 역시 자바 코드로 작성된 검증할 속성을 받아들여 프로그램이 속성을 만족하는지 여부를 알려주고, 만족하지 않을 경우에는 반례를 보여준다.

JPF는 상태공간을 줄이기 위해 슬라이싱(Slicing), 추상화(abstraction)등의 기법을 이용한다. Bandera toolset은 자바로 구현되어 있기 때문에 PC 및 UNIX/Linux의 어떤 환경에서도 동작이 가능하다.

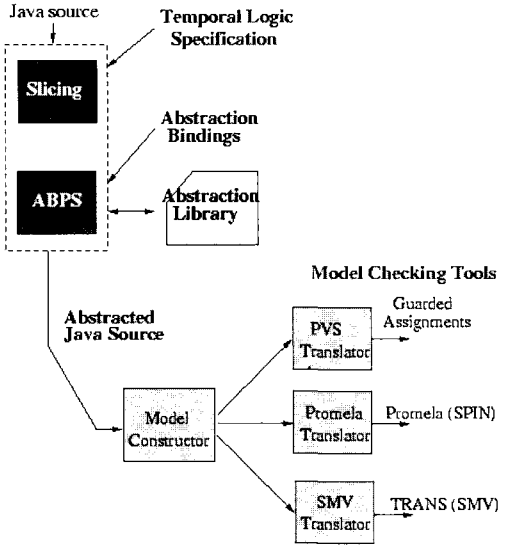


그림 4 Bandera toolset의 구조

4.2 하드웨어의 설계

위에서는 범용 프로그래밍 언어를 이용한 정확한 소프트웨어의 설계를 위한 연구에 대해 소개하였다. 현재 소프트웨어의 구현을 안전하게 하기 위한 연구가 매우 활발하게 이뤄지고 있다. 그러나 현실적으로 소프트웨어는 그 자체의 특성 때문에 정확한 설계를 위해 주로 설계자의 경험과 능력에 많이 의존하고 있는 편이다. 이에 비해 하드웨어는 그 동작이 명확하고, 결과를 예측하기에 보다 용이하다. 따라서, 설계시에 이미 안정성이 증명된 시스템을 구현하기가 보다 용이하고, 이를 목적으로 하는 설계 도구가 많이 나와 있다. 또한 정형 기법을 적용하여 시스템의 설계시에 오류를 발견하기 위한 연구가 오래 전부터 진행되어 훌륭한 도구가 많이 개발되어 있다. 이번 장에서는 그 중 대표적인 하드웨어 설계 언어인 Veirlog-HDL[13]과 이를 입력으로 받아 시스템의 안정성을 정형 검증하는 도구인 VIS[21]에 대해 알아보고, 소프트웨어와 마찬가지로 범용 프로그래밍 언어인 C를 확장하여 차세대 하드웨어 설계에 적용하고자 개발된 SystemC[14]를 이용한 안전한 시스템 설계 방법에 대해 설명하고자 한다.

4.2.1 Verilog-HDL과 VIS를 이용한 정형 검증

4.2.1.1 Verilog-HDL

Verilog-HDL[13]은 논리회로 설계에서 널리 사용되는 언어이다. Verilog-HDL은 하드웨어의 동적인 행위 뿐 아니라 정적인 구조와 관련된 하드웨어의 혼합수준 명세도 허용한다. 동적인 행위의 표현을 용이하게 하기 위해 Verilog-HDL은 조건문, 반복-제어문, 프로세스 생성과 같은 고수준 구조를 갖는다. Verilog-HDL은 또한 시간 표현을 용이하게 하는 식별자도 갖는다. 이는 statement, gate, module가 연관된 delay를 명세하기 쉽게 한다.

BLIF-MV[22]는 HSIS(integrated interactive Hierarchical verification/Synthesis System)에 의해 사용되는 입력 형식이다. BLIF-MV에 있는 기본적인 구조는 모듈 선언/인수화, 비결정성의 표현을 허용하는 테이블과 symbolic latch로 구성된다. 각 클럭 주기에서 각 테이블은 fixed point가 도달할 때까지 그것이 보고 있는 입력에 따라 현재 상태 입력에 따라 현재 상태 입력을 수정한다. 그런 다음 테이블은 그들의 출력을 수정한다. Vl2mv는 Verilog-HDL로 된 디자인과 VIS에 있는 강력한 합성 / 검증 알고리즘(synthesis / verification algorithm) 사이의 중간 역할을 한다. vl2mv는 시뮬레이션 결과에 관련되어 정의된 Verolg-HDL 프로그램 소스의 행위를 보존하는 유한상태 기계의 집합을 추출한다. 하드웨어 게이트를 Verilog-HDL에 있는 operator에 할당하는 것은 자원의 제한이 없다는 가정에 기반 한다. Resource pool은 BLIF-MV에 표현할 수 있는 모든 가능한 게이트를 구성한다. 이 때 추출된 유한상태 기계는 최적화 되었다고 할 수 없다. 최적화된 구현을 얻기를 원한다면 SIS같은 합성 시스템을 사용할 수 있다. BLIF-MV는 요구하는 특성이 만족하는지 검증에도 사용할 수 있다. BLIF-MV는 요구하는 특성이 만족하는지 검증에도 사용 할 수 있다. 이것이 정형적으로 디자인이 명세를 만족한다는 것을 보일 수 있다면 최적화하기 위해 합성 시스템에 입력으로 사용될 수 있다. 또한 Verilog-HDL를 비결정적 전이를 묘사할 수 있게 확장한다.

VIS(Verification Interfacting with Synthesis)[21]는 유한 상태 기계로 표현된 하드웨어 시스템을 정형

검증하고 시뮬레이션 할 수 있는 도구로써 Verilog-HDL을 사용하여 계산형 트리(CTL) 모델 검증, 계층적 분석(hierarchical synthesis), 동등성 검사(equivalence checking)를 지원한다. VIS는 Verilog-HDL를 입력받아 MDD로 변환 가능한 파일 형태인 BLIF-MV(Berkeley Logic Interchange Format-Multi valued Variable)로 변환한 후 수행한다. BLIF-MV 표현을 이용하여 정형 검증의 핵심인 CTL 모델검증을 행할 수 있고 SIS와 연계하여 Synthesis도 할 수 있다. VIS의 특징을 살펴보면 다음과 같다. VIS는 BLIF-MV라는 중간 형태에서 작동하는데 이는 SIS에서 사용되는 BLIF의 확장된 형태이다. VIS는 Verolg-HDL을 BLIF-MV로 변환해 주는 vl2mv컴파일러를 갖고 있다. Vl2mv는 원래의 Verilog-HDL 프로그램과 같은 행위(behavior)를 보존하고 상호 작용(interactive)하는 유한상태 기계를 추출해 낸다. VIS에서 사용되는 Verilog-HDL이 일반적으로 사용되는 Verilog-HDL과 다른 점은 비결정성과 기호적 변수이다.

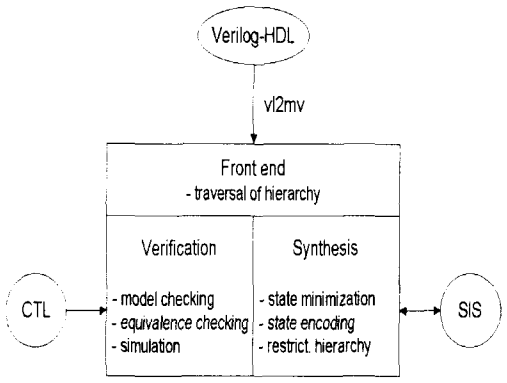


그림 5 VIS의 구조

비결정적 구성자인 $ND는 wire변수에서의 비결정성을 나타내기 위해서 Verilog-HDL에 추가 되었다. 이것이 VIS에서 비결정성을 자연스럽게 사용할 수 있게 하였다. 종종 변수의 값을 기호적으로 명세하고 참조하는 것이 직접적으로 표현하는 것보다 바람직한 경우가 있다. Vl2mv가 Verilog-HDL에서 C에서 사용하는 것과 유사한 열거형(enumerated type)으로 사용하여 기호적 변수를 표현 할수 있도록 한다.

BLIF-MV 표현이 VIS에서 읽혀질 때 모듈의 트리로써 계층적으로 저장된다. 이 계층에 대한 순회는 UNIX의 디렉토리에 대한 순회와 유사하고 시뮬레이션과 검증 작업은 어느 계층에서나 가능하다

VIS를 이용한 검증 결과는 소프트웨어와 마찬가지로 오류의 발생을 나타내는 동작 경로를 모두 보여주게 된다. 이 경로에 따라 Verilog 입력 모델을 분석하면 어떤 상태에서 오류를 발생하는지를 찾아낼 수 있고, 그 결과에 따라 하드웨어 시스템의 설계 오류를 수정할 수 있다.

4.2.2 SystemC를 이용한 하드웨어 시스템 설계

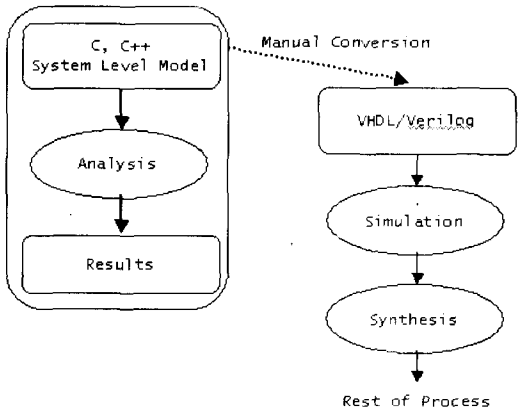


그림 6 일반적인 하드웨어 시스템 설계 방법

VIS를 이용해서 검증을 할 경우 입력 언어로는 Verilog를 사용하게 된다. 이 언어는 하드웨어 설계용으로 만들어졌기 때문에 전문 하드웨어 설계자들은 대부분이 이 언어를 이해하고 있다고 볼 수 있다. 그러나, 실제 시스템 개발 단계에서는 일반적인 설계자들의 경우에는 하드웨어 설계 언어에 익숙하지 않다. 따라서, 일반 설계자들이 C 언어와 같은 범용 프로그래밍 언어를 이용하여 시스템의 동작을 설계하면 하드웨어 전문 설계자들에 의해 verilog로 변환하여 검증하는 과정을 거치게 된다. 또한 근래에는 하드웨어 시스템 역히 급속도로 발전하고 있어 단순한 논리회로 수준의 시스템이 아닌 SoC와 같이 복잡한 구조를 갖게 되었다. 이러한 추세에서 기존의 하드웨어 설계 방식에 따라 설계 및 검증을 수행하게 되면 실제 시스템과 검증 모델의 변환 과정에서 오류가 발생할 수 있고, 시스템 설계자와 하드웨어 검증을 수행하는 사람사이의 의사 소통에도 문제가 발생할 수 있다. 이러한 이유로 일반 시스템 설계자들이 사용하

는 C 언어와 같은 범용 프로그래밍 언어를 확장하여 하드웨어의 설계를 직접 수행할 수 있는 설계 방법론이 대두되었는데 이 때 사용하고자 하는 프로그래밍 언어가 SystemC[14]이다.

SystemC는 C++ 라이브러리이고 소프트웨어 알고리즘, 하드웨어 아키텍쳐, 시스템 레벨 디자인, SoC를 효과적으로 생성하는데 사용할 수 있는 방법론이다. SystemC와 표준 C++개발 도구를 시스템 레벨 디자인이나 유효성을 증명하기 위한 빠른 시뮬레이션, 디자인 최적화, 다양한 알고리즘의 검증, 소프트웨어-하드웨어 개발 팀에 시스템의 실행 가능한 사양을 제공하기 위해서 사용할 수 있다. 실행 가능한 사양은 실행되었을 때 시스템과 같은 동작을 표현하는 본질적인 C++ 프로그램이다.

C, C++은 효과적인 시스템 기술, 제어와 컴팩트한 시스템을 개발하는데 필요한 데이타 추상을 제공해 주기 때문에 소프트웨어 알고리즘을 위한 선택이고 인터페이스 사양이다. 대부분의 설계자들은 이러한 언어들에 익숙해져 있고 많은 수의 개발 장비들이 이와 연관되어 있다.

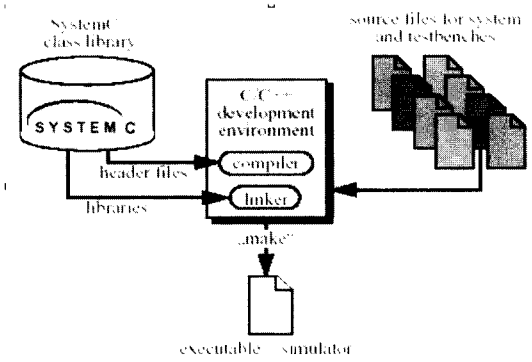


그림 7 SystemC 소개

SystemC 클래스 라이브러리는 표준 C++에서는 찾아볼 수 없는 하드웨어 타이밍, 병렬 동작, 반응 동작(reaction behavior) 등을 포함하는 시스템 아키텍쳐를 모델링하기 위한 필요한 구조를 제공해준다. 이러한 구조를 C에 포함시키는 것은 언어에 대한 독점적인 확장을 요구한다. C++ 객체 지향 프로그래밍 언어는 새로운 문법적 구조를 추가하지 않고도 언어를 클래스들을 통해서 확장할 수 있는 능력을 제공하고 있다. SystemC는 C++에 필요한 클래스들을 제공하고 설계자들에게 친숙한 C++언어와 개발 장비를 계속 사용할 수 있도록 해 만약 C++프로그래밍 언어에 친숙해 있다면 추가적인 문법을 배우지 않고도 SystemC 클래스에서 소개되는 추가적인 의미를 이해함으로써 SystemC로 프로그램하는 방법을 배울 수 있다.

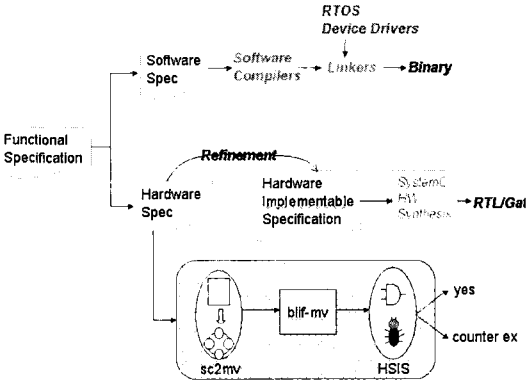


그림 8 SystemC 프로그램의 정형 검증 방법

SystemC가 C 언어의 특성을 가지고 있기 때문에 구현은 용이하지만 현재 SystemC로 설계된 하드웨어에 대한 정확성 검증을 수행하는 도구는 만들어져 있지 않다. 이에 필자의 연구실에서는 SystemC를 입력으로 받아 VIS를 이용한 모델 검증을 수행하는 방법을 연구하였고, 그 초기 버전으로 sc2mv라는 도구를 개발하였다. Sc2mv는 SystemC를 VIS의 입력 중간 언어인 Blif_MV로 변환시켜주는 도구이다.

앞에서 말한 바와 같이 VIS는 Verilog로부터 Blif_MV로 변환된 언어를 입력으로 동작하게 된다. 이 방법과 마찬가지로 SystemC를 Blif_MV로 변환할 수 있다면 이것 역시 VIS를 이용한 모델 검증에 사용할 수 있다. 연구 결과 약간의 제약은 존재하지만 SystemC로부터 Blif_MV로 성공적인 변환을 수행할 수 있는 도구를 개발할 수 있었으며, VIS를 통한 정형 검증을 수행했을 때 verilog를 이용한 정형 검증과 같은 효과를 가져올 수 있었다.

이 방법에 따르면 시스템의 설계 단계가 다음과 같이 간략화 된다.

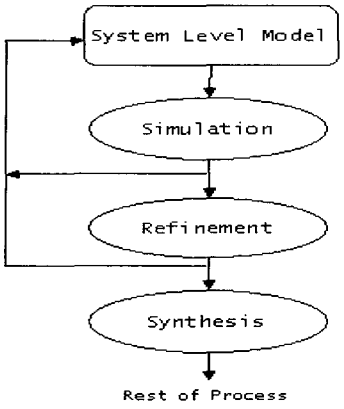


그림 9 SystemC를 이용한 개발 방법

5. 결 론

정보 산업이 발전함에 따라 컴퓨터 하드웨어 및 소프트웨어 시스템의 개발 주기가 매우 짧아지고 있다. 또한, 시스템의 동작 역시 예전의 중앙집중형 구조에서 분산형으로, 그리고 객체지향형 프로그래밍 언어에 의한 개발로 발전하고 있다. 그리고, 내장형 시스템과 같이 크기는 매우 작지만 하드웨어 및 소프트웨어가 유기적으로 연결되어 기능을 발휘하는 시스템이 활발히 연구되고 있다. 이러한 발전에 따라 시스템의 동작상의 안정성을 보장하기 위한 노력이 꾸준히 이뤄지고 있다. 전통적으로 안전한 시스템을 설계하기 위해 테스팅과 같은 방법이 주로 사용되어 왔다. 그러나, 이 방법은 일단 시스템을 설계한 후 수행되고, 테스팅에 사용할 시나리오에 따라 그 신뢰도가 결정된다. 따라서 최근에는 시스템을 설계하는 단계에서 안정성을 증명하는 정형 기법에 대한 연구 및 정형 검증 도구가 많이 개발되고 있다. 그러나, 정형 검증에 필요한 지식이 매우 이론적이고, 일반 설계자들에게는 어렵게 느껴지는 것이 사실이다. 결국, 정형 검증을 수행하기 위한 시간적/경제적인 소비 때문에 정형 검증에 기반한 안전한 컴퓨터 시스템의 개발이 어렵게 느껴지는 것이 사실이다.

이러한 어려움을 극복하기 위해 시스템 설계자에 의해 설계된 하드웨어 및 소프트웨어 코드를 자동으로 정형 검증 도구로 변환하여 안정성 증명을 수행하는 방법론이 활발히 연구되고 있으며 몇 가지 좋은 도구들이 개발되어 있다. 국내에서도 여러 대학을 중심으로 정형 기법에 대한 연구가 활발히 이뤄지고 있으며 해외에서 개발중인 도구와 마찬가지로 좋은 결과를 산출하고 있다.

현재 우리 나라의 컴퓨터 산업 수준은 세계 그 어느 나라와 비교해도 손색없는 수준에 올라와 있다고 말할 수 있다. 그러나, 산업체에서 개발한 시스템에 대한 안정성을 보장하는 기술 및 안정성이 검증된 개발품의 생산에 관한 관심은 다소 소홀한 것이 사실이다. 국내에서도 이러한 문제에 조금 더 민감하게 반응하고, 보다 정확하고 안전한 시스템의 개발을 위한 연구에 박차를 가한다면, 보다 진보되고 국제적인 경쟁력을 갖추게 될 것으로 보인다.

참고문헌

[1] A. Harry, Formal Methods-FACT FILE, VDM and Z, Wiley, 1996.

[2] G. J Myers. The Art of Software Testing. Wiley, 1979.

[3] D. H. Craigen, S. L. Gerhart and T. J. Ralston, "An International Survey of Industrial Applications of Formal Methods", NRL/FR /5546 ?93-9581, Vol. 1, 1993.

[4] E. M. Clarke and J. M. Wing, "Formal Methods: State of the Art and Future Directions", ACM Computing Surveys, vol. 28, No. 4, pp. 626-643, 1996.

[5] K. L.McMillan, Symbolic Model Checking, Kluwer Acadecmic Publisher, 1993.

[6] Z. Manna and A. Pnueli, The Temporal Logic of Reactive and Concurrent Systems - Specification, Springer-Verlag, 1996.

[7] J. Hatcliff and M. Dwyer, "Using the Bandera Tool Set to Model-check Properties of Concurrent Java Software", Proceedings of CONCUR 2001 (LNCS 2154), June, 2001.

[8] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, H. Zheng, "Bandera : Extracting Finite-state Models from Java Source Code", Proceedings of the 22th ICSE, 2000.

[9] T. Ball, S. K. Rajamani, "The SLAM Project: Debugging System Software via

Static Analysis", Proceedings of POPL 2002, January, 2002.

[10] C. Demartini, R. Iosif and R. Sisto, "dSPIN: A dynamic extension of SPIN. In Theoretical and Applied Aspects of SPIN Model Checkin g", Proceedings of the 6th International SPIN Workshop LNCS 1680, 1999.

[11] K. Havelund and T. Pressburger. "Model Checking Java Programs Using Java PathFinder", International Journal on Soft ware Tools for Technology Transfer, 1998.

[12] G. J. Holzmann and M. Smith, Software Model Checking, Invited Paper, ICSE 99, 1999.

[13] P. Samir, Verilog HDL, Prentice Hall, 1996.

[14] SystemC Version2.0 User's Guide, SystemC.org, 2002.

[15] G. J. Holzmann, Design and Validation of Computer Protocols, Prentice Hall, 1991.

[16] G. J. Holzmann, "The Model Checker? SPI N", IEEE Transactions on Software Engineering, 1997.

[17] T. Ball and S. K. Rajamani, "The SLAM Toolkit", proceeding of CAV 2001, 2001.

[18] T. Ball and S. K. Rajamani, :Boolean Programs: A Model and Process for Software Analysis:, MSR Technical Report 2000-14.

[19] T. Ball and S. K. Rajamani, "SLIC: A Specification Language for Interface Checking(of C)", MSR-TR-2001-21.

[20] T. Ball and S. K. Rajamani, "Bebop: A Symbolic Model Checker for Boolean Programs", Proceedings of 6th International SPIN Workshop, LNCS 1885, 2000.

[21] T. Villa, G. Swamy, T. Shiple, VIS User's Manual, University of California, Berkeley, 1996.

[22] Y. Kukimoto, BLIF-MV, The VIS Group, University California, Berkely, May, 1996.

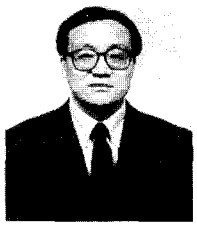
방 기 석

1997 고려대학교 정보공학과 졸업
2000 고려대학교 대학원 컴퓨터학과 석사 졸업
2000 현재 고려대학교 대학원 컴퓨터학과 박사 과정
관심분야 : 정형기법, 실시간 시스템, 운영체제, 소프트웨어 공학, 네트워크 프로토콜
E mail : kbang@formal.korea.ac.kr

유 희 준

1997 고려대학교 컴퓨터학과 학사
1999 고려대학교 대학원 컴퓨터학과 석사
1999~현재 고려대학교 대학원 컴퓨터학과 박사과정
관심분야 : 컴퓨터이론, 정형기법(정형명세, 정형검증), 소프트웨어 공학, 암호 프로토콜
E mail : hyoo@formal.korea.ac.kr

최 진 영



1982 서울대학교 컴퓨터공학과 졸업
1986 Drexel University Dept. of Mathematics and Computer Science 석사
1993 University of Pennsylvania Dept. of Computer and Information Science 박사
1993~1996년 Research Associate, University of Pennsylvania
1994~1995 Computer Scientist, Computer Command and Control Company(Part time)
1996~1999 고려대학교 컴퓨터학과 조교수
1999~현재 고려대학교 컴퓨터학과 부교수
관심분야 : 컴퓨터이론, 정형기법(정형 명세, Formal verification), 실시간 시스템, 분산 프로그래밍 언어, 소프트웨어 공학
E mail : choi@formal.korea.ac.kr