■ 2008년도 학생논문 경진대회 수상작

하드웨어/소프트웨어 동시검증을 위한 합성 가능한 인터페이스 검증 기법
(Synthesizable Interface Verification for Hardware/Software Co-verification)

이 재 호 † 한 태 숙 †† 윤 정 한 †††
(Jaeho Lee) (Tai Sook Han) (Jeong-Han Yun)

요 약 임베디드 시스템은 오늘날 우리 일상에서 널리 사용되고 있고 그 중요성은 더욱 증대되고 있다. 이에 비례하여 임베디드 시스템의 복잡도와 이를 개발하려는 노력 또한 더욱 더 증가하고 있다. 하드웨어와 소프트웨어로 구성되어 있는 임베디드 시스템의 이질적인 특성은 시스템 개발 및 통합 시에 에러를 야기하는 주원인이 된다. 그 중에서도, 하드웨어와 소프트웨어 간의 인터페이스에서 발생하는 에러가 시스템 에러의 13%를 차지하고 있으며 이 비율은 더욱 증가하는 추세이다.

우리는 하드웨어와 소프트웨어 동시설계를 위한 실제적인 인터페이스 동시 검증 기법을 제안하고 이를 지원하는 도구를 구현하였다. 먼저, 이 논문은 하드웨어와 소프트웨어간의 상호작용을 기술할 수 있는 인터페이스 명세를 정의한다. 이 명세 방법은 하드웨어와 소프트웨어 서로간의 특성을 잘 표현할 수 있고, 소프트웨어 명세로부터 하드웨어 명세로의 변환이 가능하여 전체 시스템이 소프트웨어의 입장에서 기술될 수 있도록 한다. 둘째, 작성된 하드웨어 설계와 소프트웨어 설계에 대해 명시된 인터페이스의 의미대로 동작하는지를 검증하는 기법을 제시한다. 주어진 명세로부터 소프트웨어의 동작을 가정하고 이를 하드웨어 설계로 모델링하여 하드웨어 인터페이스에 대한 모델검증을 수행하고, 그 후 소프트웨어의 동작에 대해 검증을 수행하는 가정-보증 추론(assume-guarantee reasoning) 방식의 검증을 수행한다. 마지막으로 기존의 검증 연구들이 저수준의 인터페이스를 추상화하여 현실적 적용이 힘들었던 반면 우리는 디바이스 API, 디바이스 드라이버, 디바이스 컨트롤러 등의 저수준의 인터페이스 코드들을 자동으로 생성하여 검증된 하드웨어와 소프트웨어 코드가 바로 통합되어 시스템을 구축할 수 있는 실제적인 해결책을 제시한다.

키워드 : 하드웨어/소프트웨어 동시검증, 인터페이스 검증, 인터페이스 자동 생성

Abstract The complexity of embedded systems and the effort to develop them has been rising in proportion with their importance. Also, the heterogeneity of the hardware and software parts in embedded systems makes it more challenging to develop. Errors caused by hardware/software interfaces, especially, account for up to 13 percent of failures with an increasing trend. Therefore, verifying the interface between hardware and software in embedded system is one of the most important research areas. However, current approaches such as co-simulation method and model checking have explicit limitations.

· 본 연구는 지식경제부 및 정보통신산업진흥원의 대학 IT연구센터 지원사업(NIPA-2010-C1090-1031-0004) 및 2008년 정부(교육과학기술부)의 재원으로 한국학술진흥재단(KRF-2008-313-D00968)의 지원으로 수행되었음

† 정 회 원 : 삼성전자 DMC연구소 SE Lab
namjelee@gmail.com
†† 종신회원 : KAIST 전산학과 교수
han@cs.kaist.ac.kr
††† 학생회원 : KAIST 전산학과
jeonghan.yun@gmail.com

논문접수 : 2008년 4월 22일
심사완료 : 2009년 9월 25일

Copyright©2010 한국정보과학회 : 개인 목적이나 교육 목적인 경우, 이 저작물의 전체 또는 일부에 대한 복사본 혹은 디지털 사본의 제작을 허가합니다. 이 때, 사본은 상업적 수단으로 사용할 수 없으며 첫 페이지에 본 문구와 출처를 반드시 명시해야 합니다. 이 외의 목적으로 복제, 배포, 출판, 전송 등 모든 유형의 사용행위를 하는 경우에 대하여는 사전에 허가를 얻고 비용을 지불해야 합니다.

정보과학회논문지: 소프트웨어 및 응용 제37권 제4호(2010.4)

In this paper, we propose the synthesizable interface co-verification framework for hardware/software co-design. Firstly, we introduce the separate interface specifications for the heterogeneous components to describe hardware design and software design. Our specifications are expressive enough to describe both. We also provide the transformation rules from the software specification to the hardware specification so that the whole system can be described from the software view. Secondly, we address the solution of verifying the interface of the software and hardware design by adopting and extending existing verification-techniques and extending them. In hardware interface verification, we exploit the model checking technique and provide more efficient verification by closing the hardware design from the assumption of the software behavior which is ensured by software verification step. Lastly, we generate the interface codes such as device APIs, device driver, and device controller from the specification so that verified hardware and software codes can be synthesized without extra efforts.

Key words : Hardware/Software Co-verification, Interface verification, Interface generation

1. 서 론

일반 가전기기부터 안정성이 요구되는 공공설비의 제어 시스템까지 임베디드 시스템은 오늘날 우리 일상에서 널리 사용되고 있고 그 중요성은 더욱 증대되고 있다[1]. 반면에 이런 중요성과 하드웨어 기술 발전 속도에 비례하여, 신뢰성 있는 임베디드 시스템의 개발은 더욱 많은 노력과 시간을 요구하고 있다. 하드웨어와 소프트웨어의 이질적인 두 요소로 구성되어 있다는 임베디드 시스템의 이질성이 시스템 개발 및 통합을 어렵게 하며, 하드웨어/소프트웨어 인터페이스 설계를 위해 하드웨어와 소프트웨어 설계를 모두 이해하고 있는 새로운 유형의 개발자를 필요로 한다[2]. 최근 연구[3]에 의하면, 하드웨어/소프트웨어 인터페이스에서 발생하는 에러가 실제 시스템 에러의 13%를 차지하고 있으며 이 비율은 점차 증가하는 추세이다.

이를 해결하기 위해, 임베디드 시스템 개발에 대한 생산성과 신뢰성을 높이기 위한 많은 연구들이 계속 활발하게 진행되고 있지만 각각 한계를 지니고 있다. 먼저 동시-시뮬레이션(co-simulation)과 동시-에뮬레이션(co-emulation)을 이용한 방법들[4-10]은 테스트 기반의 검증이기 때문에 완전히 에러가 없음을 보장해 주지 못하고, 속도 면에서 현실적이지 못한 경우가 많다. 반면 모델검증[11-13]을 이용한 방법의 경우, 모델기반의 검증이기 때문에 실제 구현과 검증 모델 간에 차이가 발생하게 되며, 하드웨어 분야에 대해서는 잘 연구되어 있으나, 소프트웨어 분야에 대해서는 한계가 존재하며, 소프트웨어와 하드웨어가 결합된 시스템의 경우에는 해결책이 존재하지 않는다[3].

이 논문은 임베디드 시스템의 소프트웨어와 하드웨어 컴포넌트들이 어떻게 사용되어야 하는지를 기술할 수 있는 오토마타 기반의 인터페이스 명세를 정의하고 기술된 명세대로 시스템이 구현되었는지를 검증하는 인터페이스 기반의 검증 프레임워크를 제안한다. 본 논문에서 의미하는 인터페이스는 구조적인 측면이 아닌 행위적인 측면에서의 인터페이스로써, 컴포넌트가 외부환경과 상호작용하면서 어떻게 사용되는지를 표현하고, 외부환경에 대한 제약사항을 함축한다[14].

주어진 명세에 대한 검증은 기존에 잘 연구된 기법을 병행하여 적용한다. 소프트웨어 설계(SW design)의 검증은 기존에 우리가 연구한 정적 자원 검사 도구인 Pruv-C[15]를 이용하였고, 하드웨어의 검증은 모델검증 기술을 사용하여 검증한다. 모델검증에 있어서, 상태 폭발 문제와 외부환경에 대한 모델링을 해주어야 한다는 제한점이 존재하는데[12], 우리는 소프트웨어 인터페이스 명세로부터 가정할 수 있는 소프트웨어의 동작 정보를 모델검증기(model checker)에 추가로 제공해 줌으로써, 닫힌 시스템을 만들어 하드웨어 설계에 대한 검증의 효과를 높이도록 노력하였다. 또한 검증에서 추상화되고 배재되었던 하드웨어와 소프트웨어 사이의 저수준의 인터페이스 코드들(디바이스 API, 디바이스 드라이버, 디바이스 컨트롤러)에 대해 자동 생성을 함으로써 검증된 코드가 바로 통합되어 시스템을 구축할 수 있는 실제적인 동시검증 프레임워크를 제공한다.

1.1 문제 정의

우리의 관심은 임베디드 시스템에 대한 하드웨어와 소프트웨어에 대한 코드가 그림 1과 같이 주어졌을 때 이 두 컴포넌트가 서로 유기적으로 잘 결합되어 동작하는 지를 검증하는 것이다. 그림 1(a)는 Esterel[17] 언어로 구현된 하드웨어 설계이고, 그림 1(b)는 C언어로 구현된 소프트웨어 설계이다. 이 시스템은 문을 열고 닫는 간단한 도어 컨트롤 시스템으로써, 소프트웨어 설계는 사용자의 요구를 받아 현재 상태에 대해 적법한 요청이 들어오면 그것을 하드웨어에게 지시하고, 하드웨어 설계는 소프트웨어의 명령에 따라 모터를 조작하여 문을 열고 닫는 동작을 수행하도록 구현되었다.

```
module DoorController:
input open, close, opened, closed;
output moveUp, moveDown,
output open_compl, close_compl;
loop
   await open;
   abort
      sustain moveUp;
   when opened;
   emit open_compl;
end loop;
||
loop
   await close;
   abort
      sustain moveDown;
   when closed;
   emit close_compl;
end loop;
end module
```

(a) 하드웨어 설계

```
...
state = CLOSED;
while(1){
  userReq = getReq();
  switch(userReq){
    case OPEN_DOOR:
      if(state == CLOSED){
        openDoor();
        state = OPENED;
      }
      break;
    case CLOSE_DOOR:
      if(state == OPENED){
        closeDoor();
        state = CLOSED;
      }
      break;
  }
}
...
```

(b) 소프트웨어 설계

그림 1 간단한 임베디드 시스템에 대한 하드웨어/소프트웨어 설계 예제

이러한 간단한 코드에서 조차, 다양한 문제가 발생할 수 있다. 소프트웨어의 openDoor()나 closeDoor() 명령에 대해 하드웨어 설계가 잘못되어 올바르게 응답하지 않는다면 시스템이 교착상태에 빠질 수 있고, 반대로 소프트웨어 설계가 잘못되어 문이 열린 상태에서 계속 openDoor()함수를 호출하여 여는 동작을 반복시키면, 여기서는 시스템의 수명 단축으로만 끝날 수 있지만, 좀 더 복잡한 시스템에서는 대형 인명사고가 야기될 수 있다. 또한 추상화되었지만 두 코드 사이에는 통신을 위해 버스를 컨트롤하는 모듈과 디바이스를 조작하기 위한 디바이스 드라이버 모듈 등의 인터페이스 모듈이 필요하고, 이 모듈들의 구현 또한 정확해야 한다.

문제는 두 컴포넌트가 통합된 상태에서의 검증은 하드웨어와 소프트웨어의 이질적인 특성으로 인해 힘들다는 데 있다. 그림 1의 예제의 경우, 예제의 간단함과 저수준 인터페이스 모듈들의 추상화에도 불구하고 쉽게 이 두 코드가 맞아서 돌아가는 지를 확인하기 위해서는 두 언어에 대한 지식과 각각에 대한 모듈단위 테스트가 필요하게 되고, 테스트를 통과한 이후에도 시스템을 통합했을 경우에 대하여 안정성을 완벽히 보장할 수 없다. 이러한 동시 검증 문제를 해결하기 위해서는 아래와 같은 두 문제가 해결되어야 된다.

- *하드웨어와 소프트웨어를 위한 명세 제공*:
  그림 1에서 보는 것과 같이 소프트웨어와 하드웨어의 설계는 이질적이다. 소프트웨어의 경우 하드웨어와의 통신을 디바이스 API만으로 수행한다. 반면, 하드웨어는 소프트웨어와의 상호작용을 입력 신호에 따른 출력 신호의 방출로 여긴다. 따라서 이 이질성을 잘 표현할 수 있는 명세가 필요하다.
- *명세에 대한 동시 검증 제공*:
  기존의 검증 연구는 대부분 하드웨어나 소프트웨어 중 한 분야만을 타깃으로 하고 있고, 하드웨어와 소프트웨어가 통합된 시스템의 경우에 대해서는 해결책을 제공하지 못하고 있다. 또한 주로 구현이 아닌 모델에 대한 검증만을 수행하거나 저수준의 인터페이스들을 요약하기 때문에 검증방법이 실제 적용하기 힘들다. 따라서 실제적인 동시 검증 방법에 대한 해결책이 필요하다.

1.2 제안하는 해결책

이 논문에서는 위에서 정의한 문제에 대해서 다음과 같이 해결하고자 하였다.

- 하드웨어와 소프트웨어의 이질적인 특성을 살릴 수 있도록 두 개의 인터페이스 명세를 제공한다. 기존의 연구들[4]은 하나의 명세만을 제공하여 둘 간의 근본적인 차이를 잘 표사할 수가 없었다. 두 개의 명세를 제공하면 이를 해결할 수 있지만, 두 명세 간에 호환성 문제가 대두된다. 따라서 두 개의 명세의 제공과 더불어 소프트웨어 인터페이스 명세에서 하드웨어 인터페이스 명세를 추출할 수 있도록 변환규칙을 정의하였다.
- 둘째로, 하드웨어 검증과 소프트웨어 검증에 대해 각각 적합한 다른 검증 방법을 적용하되 서로간의 검증 정보를 활용한 가정-보증 추론[19]을 수행한다.
- 하드웨어와 소프트웨어 검증으로부터 추상화되고 배제되었던 인터페이스 모듈들에 대해 자동 생성을 한다. 버스 관리나 통신 채널에 대한 구현은 잘 작동한다는 가정 하에 검증을 수행하기 때문에 검증이 완전하려면 검증된 코드들 사이의 인터페이스를 지원하는 모듈이 필요하다. 우리 연구에서는 이러한 모듈에 대해 *인터페이스 코드*라고 명명하고 이를 자동 생성한다.

논문은 구성은 다음과 같다. 2장에서는 논문과 관련된 기존의 연구에 대해 설명한다. 3장에서는 논문에서 제시하는 시스템의 전체 프레임워크에 대한 개관하고, 4장에서는 우리가 정의한 하드웨어와 소프트웨어 인터페이스 명세에 대해 설명하고 이 둘 간의 관계에 대해서 설명한다. 자세한 동시검증 알고리즘과 인터페이스 코드 생성에 대한 알고리즘은 5장과 6장에 각각 설명한다. 7장에서는 구현과 사례 연구를 소개한다. 8장에서는 본 논문의 결론과 향후 개선점에 대해 기술한다.

2. 관련 연구

2.1 동시 시뮬레이션 기반 검증

임베디드 시스템 개발의 어려움을 해결하기 위해, 다양한 동시 개발 방법론과 동시 개발 도구들이 연구되고 개발되어 오고 있다. 동시 시뮬레이션 기반 검증 연구들은 시뮬레이션 엔진과 명세 언어에 따라 크게 동종(homogeneous)[4,5], 이종(heterogeneous)[6-8], 혼종(semi-homogeneous)[9] 동시 검증 환경으로 구분할 수 있다. 하지만 이러한 동시 시뮬레이션 기반 검증 방법들의 근본적인 문제는 시뮬레이션에 의존한 방법이기 때문에 완전한 동시 검증을 제공하는 데 있어서 한계가 있고 검증을 대상으로 하는 임베디드 시스템에 대하여 어떤 속성을 만족한다고 확실히 보장하지 못한다.

2.2 인터페이스 오토마타 기반 검증

시스템을 블록의 조합으로 보고, 블록 단위로 시스템을 개발하고 조합할 수 있도록 인터페이스를 정의하고 알고리즘을 제안하는 연구가 활발하게 진행되고 있다 [14,19]. 이 연구들에서는 기존의 전통적인 타입 시스템의 도메인과 값을 인터페이스 오토마타로 확장했다고 볼 수 있다. 두 개의 컴포넌트가 존재할 때 이들에 대한 인터페이스가 주어지면 이 컴포넌트가 서로 합성이 가능한 지를 검증해 준다. 따라서 개발자들은 자신이 사용할 모듈을 고른 후 이 모듈들이 합성 가능한 지 검사할 수 있다.

하지만 이 연구들에서는 입출력만을 고려한 오토마타 기반의 추상화를 제공하고, 오직 모델에 대해서만 합성 알고리즘을 제시하고 있다. 따라서 이 연구들은 하드웨어, 소프트웨어의 이질적인 케이스에 적용될 수 있는 해결책이 아니고, 실제 구현 코드와 오토마타 사이에 서로 동치의 관계가 있음을 보장해 주는 일이 필요하다.

2.3 외부환경 자동 생성

모델검증 분야에서도 기존의 문제를 해결하기 위한 여러 가지 노력이 있고, 우리와 같이 외부환경을 모델링하여 모델검증의 효과를 높이려는 노력도 있다. [20]에서는 사용자의 가정으로부터 외부환경에 대한 모델링을 하여 Java 언어에 대한 모델검증의 성능을 증가시키려고 하였다.

명세로부터 외부환경에 대한 모델링을 한다는 점에서 우리의 연구와 비슷하지만 이 논문에서 대상으로 하는 것은 소프트웨어의 한 언어만을 대상으로 하였고, 실제로 이질적인 두 요소 사이에는 적용이 불가능하기 때문에 임베디드 시스템의 검증을 위해서는 적합하지 않은 해결책이다.

3. 시스템 개요

그림 2에 우리가 제안하는 검증 프레임워크를 볼 수 있다. 옅은 음영의 박스는 기존에 존재하는 도입된 검증 도구이고 짙은 음영 박스는 추가로 구현된 부분이다. 우리 프레임워크는 하드웨어 설계(HW design)와 소프트웨어 설계(SW design)와 둘 간의 인터페이스 명세를

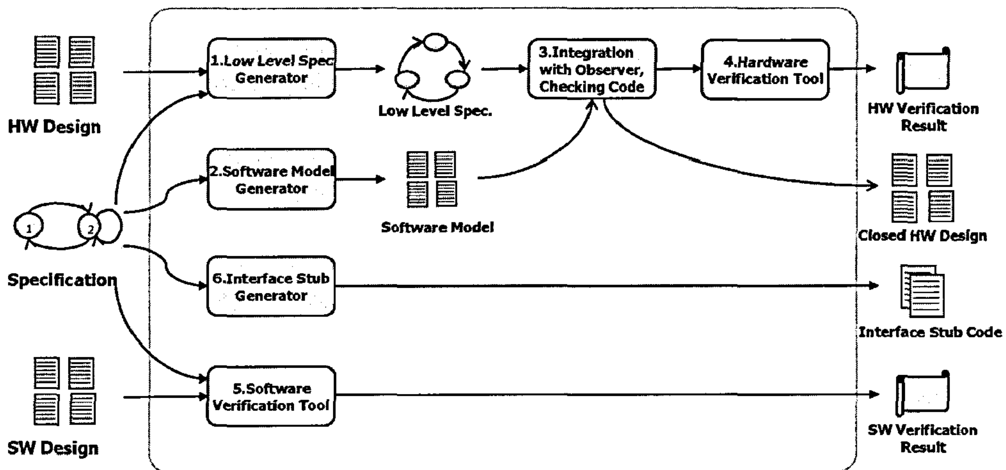


그림 2 인터페이스 동시 검증 프레임워크

입력으로 받는다. 우리가 이 논문에서 대상으로 하고 있는 임베디드 시스템은 특별히 더 높은 안정성이 요구되는 반응형 시스템(reactive system)이다. 따라서 반응형 시스템의 기술에 적합한 Esterel을 하드웨어 설계 언어로 선택하였고, 소프트웨어 설계 언어는 C 언어를 대상으로 한다. Esterel과 C의 구현코드와 명세로부터, 두 코드에 대한 검증 결과와 이 두 검증된 코드를 연결 시켜주는 인터페이스 코드, 그리고 사용자가 수동적으로 다른 속성을 검증할 수 있도록 소프트웨어의 동작이 통합된 하드웨어 설계를 출력한다. 각각의 요소를 간단히 설명하면 아래와 같다.

1. Low Level Spec. Generator: 사용자가 입력한 상위 레벨의 소프트웨어 인터페이스 명세로부터 하드웨어의 설계를 위한 인터페이스 명세를 추출한다. 이러한 추출은 소프트웨어와 하드웨어 사이에 정의한 가정에 따라 이루어지며, 정의한 가정에 따라 나중에 인터페이스 코드들이 생성된다.
2. Software Model Generator: 모델검증기로 모델을 보다 정확히 검증하기 위해서는 검증 대상의 외부환경에 대한 모델링이 필요하다. 우리 프레임워크에서는 소프트웨어가 주어진 인터페이스 명세를 만족한다고 가정을 할 수 있기 때문에, 소프트웨어 명세로부터 소프트웨어가 하드웨어와 상호작용하는 모델을 추출해 낼 수 있다. 즉, 소프트웨어 명세로부터 Esterel 코드 M'이 생성된다.
3. Integration with Checking Code: 프로그램 수행 중에 시그널(signal)의 발생 여부만을 알려주는 Esterel용 모델검증기를 통해 오토마타 형태의 인터페이스를 검증하기 위해서는 추가적인 정보가 제공되어야 한다. 먼저 하드웨어 명세를 인코딩하여 Esterel 코드 A'를 생성하고, 명세로부터 검증할 속성들을 추출하여 Esterel 코드 P'를 생성한다. 생성된 A'와 P'와 2번에서 생성한 코드 M'을 하드웨어 구현 코드와 병렬로 통합한다.
   $(Code \| M' \| A' \| P')$
4. Hardware Verification Tool: 3번에서 통합된 하드웨어 설계를 입력으로 받아 실제 모델검증을 수행하여 하드웨어 인터페이스 검증을 수행한다.
5. Software Verification Tool: 소프트웨어 인터페이스 명세와 소프트웨어 구현 코드를 입력으로 받아 검증을 수행한다. Pruv-C를 이용하여 C 코드에 대한 정적 검증을 수행한다.
6. Interface Stub Generator: 검증을 통과한 소프트웨어 설계와 하드웨어 설계는 인터페이스 코드와 함께 합쳐져 시스템의 프로토타입을 이루게 된다. Interface Stub Generator는 인터페이스 명세로부터 인터페이스 코드들인 디바이스 API, 디바이스 드라이버 함수와 디바이스 컨트롤러를 생성한다.

4. 인터페이스 명세

본 연구는 추상적이고 표현력이 충분한 인터페이스 명세를 제공하는 것으로부터 시작된다. 소프트웨어 측면에서 보면, 기본적인 드라이버와 I/O 기능에서부터 미들웨어와 복잡한 운영체제에 이르기까지의 계층을 숨길 수 있는 추상화가 필요하고 하드웨어 측면에서는 CPU 인터페이스라고 불리는 HAL(Hardware Adaptation Layer)을 통해 CPU 버스의 자세함을 숨기는 추상화가 요구된다[2]. 이러한 추상화의 요구와 더불어 인터페이스 명세는 하드웨어와 소프트웨어의 특성을 기술할 수 있도록 충분한 표현력을 제공해야 한다. 추상화와 표현력에 대한 요구는 상반된 요구이기 때문에 인터페이스 명세를 정의하는 것은 매우 힘든 일이다.

두 가지 고려사항을 모두 만족시키기 위해, 우리는 하드웨어와 소프트웨어 설계를 위해 두 개의 인터페이스 명세를 도입했다. 일반적으로 소프트웨어 개발자는 하드웨어 모듈과의 상호작용을 연속된 디바이스 API 호출이라고 생각한다. 우리는 이러한 디바이스 API 호출의 관계를 소프트웨어가 사용할 수 있도록 인터페이스를 제공하고, 이것을 *소프트웨어 인터페이스 명세* 또는 *고수준의 명세*라고 부른다. 반대로 하드웨어 개발자는 소프트웨어와의 통신을 입력 신호와 이에 따른 적절한 출력 신호의 방출로 생각할 수 있다. 우리는 하드웨어 설계를 위해 이러한 입출력 신호의 발생 순서를 표현할 수 있는 하드웨어 명세를 제공하고, 이를 하드웨어 인터페이스 명세 또는 저수준의 명세라고 부른다.

이 장에서는 두 인터페이스 명세에 대해서 각각 설명하고, 소프트웨어 인터페이스 명세로부터 하드웨어 인터페이스 명세를 추출하는 규칙에 대해 설명하도록 한다.

4.1 하드웨어 인터페이스 명세

본 연구의 하드웨어 인터페이스 명세는 기존의 연구[14,19]에서 정의된 I/O 오토마타를 우리의 하드웨어 설계에 맞게 확장한 오토마타 명세이다. 기존의 I/O 오토마타가 입력, 출력, 내부 액션으로 나누어지는 반면에, 우리의 확장에서는 외부환경과의 상호작용을 묘사하는데 불필요한 내부 액션을 제외하였다. 또한 출력 액션에 대해 일반적인 *output* 액션과 *sustain*이라는 특수한 액션을 추가하여 하드웨어와 소프트웨어 설계 간의 타이밍 차이를 표현할 수 있도록 확장하였다.

정의 1: 하드웨어 인터페이스 오토마타

하드웨어 오토마타 $hwF = <Q, q_0, op, A^I, A^O, \delta>$는 다음의 여섯 가지 요소로 구성되어 있다.

- 유한 상태 집합 Q;
- 시작상태 $q_0 \in Q$;
- 액션의 행위를 나타내는 op: input, output, sustain (각각 ?, !, # 으로 표시);
- 공통원소를 갖지 않는 입력신호와 출력신호에 대한 두 집합 A^I와 A^O;
- $\delta \subseteq Q \times (A \times op) \times Q$ 의 전이함수 (여기서 $A = A^I \cup A^O$);

Q는 하드웨어가 가질 수 있는 상태들이고, q_0는 시작 상태로 하드웨어 설계상 시스템이 제일 처음 가지는 상태이다. A는 인터페이스 신호들의 집합으로 소프트웨어 설계와의 통신을 할 때 사용되는 신호들의 집합이다. 소프트웨어로부터 들어오는 신호에 대해서 A^I, 소프트웨어로 나가는 신호에 대해서 A^O가 된다. 사상함수의 형태는

$$Q \times (A \times op) \rightarrow 2^Q,\ \text{즉}\ \delta(q, <a, op>) = p$$

이고,(q, a, op, p)로 간결하게 나타낼 수 있다. 여기서 q,p는 상태이고, a와 op는 함께 합쳐서 액션을 의미한다. 오토마타 F는 결정적(deterministic) 속성을 가진다. 즉, 상태 $q,q',q'' \in Q$와 액션 $a \in A^I$와 이에 대응하는 op가 존재할 때, 만약 $(q,a,op,q') \in \delta$이고 $(q,a,op,q'') \in \delta$이라면 $q' = q''$가 된다는 것을 의미한다.

특별히 우리의 오토마타에서는 *sustain*이라는 행위를 나타내는 op가 존재한다. 이 op는 출력 시그널에 대한 발생과 이 시그널이 발생한 후 언제까지 유지되어야 한다는 상태 전이 이후의 제약사항을 포함한다. 예를 들어 $(q,a,\#,q')$라는 δ가 존재한다면, 이것은 q상태에서 a시그널이 발생한다면 시스템은 q'상태에 도달하게 되고, q' 상태에서 어떤 전이가 발생하기 전에는 계속 a시그널을 발생하고 있어야 함을 의미한다.

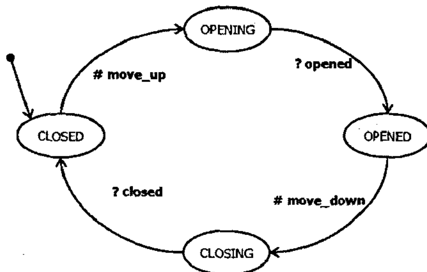
그림 3에서 우리가 정의한 하드웨어 인터페이스 오토마타의 적용 예를 보여주고 있다. 그림 3(a)는 1장에서 보았던 그림 1의 코드에 대한 시스템의 블록도이다. 그림 3(b)는 하드웨어 오토마타를 통해 하드웨어 설계와 외부환경과의 상호작용의 명세를 기술한 예이다. 그림 3(c)는 하드웨어 설계와 소프트웨어 설계 간의 인터페이스 명세를 표현한 것이다.

그림 3(b)를 보면 하드웨어와 외부환경과의 동작을 다음과 같이 묘사한다. 시스템은 문이 닫힌 상태로 시작하고, 이 상태에서 문을 여는 모터를 움직이는 `move_up` 신호가 발생되면 OPENNING 상태가 된다. 여기서 하드웨어 모듈은 문이 다 열렸다는 신호가 들어올 때까지 `move_up` 신호를 계속 방출시켜야 한다. 즉, 문이 완전히 열렸다는 신호인 `opened` 시그널이 들어오기 전까지 계속 방출되어야 하고, 이러한 제약을 *sustain op*를 통해 표현할 수 있다.

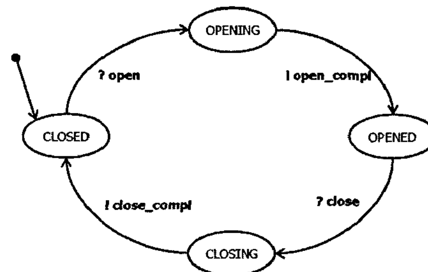
그림 3(c)는 하드웨어와 소프트웨어 모듈 사이의 상호작용을 묘사한다. 문이 닫힌 상태에서 시작하여 소프트웨어 모듈로부터 open 신호가 들어오면 문을 여는 동작을 수행하고 `open_compl` 신호를 소프트웨어에게 보냄으로써 문을 여는 동작이 완료되었음을 알린다. `close` 연산도 `open`과 대칭되게 기술되어 있다.

4.1.1 확장된 정규표현을 통한 하드웨어 인터페이스의 의미

(a) Component view of the simple door controller



(b) Hardware Interface Specification between HW and Env.



(c) Hardware Interface Specification between HW and SW

그림 3 하드웨어 인터페이스 명세의 예

기존의 오토마타는 정규표현(regular expression)으로 나타낼 수 있다. 하지만 우리가 정의한 오토마타는 *sustain*이 추가되어 상태가 제약을 가지게 되고, 하드웨어의 시간 개념이 묵시적으로 사용되고 있어 기존의 정규표현으로 나타낼 수가 없다. 따라서 좀 더 의미를 명확히 하기 위해서 확장된 정규표현을 정의하고 오토마타의 의미를 이를 이용해서 표현해 본다. 타이밍 개념이 적용된 확장된 정규표현이 그림 4에 정의되어 있다.

TICK은 논리상에서의 가상 시간 단위이다. 한 TICK의 구성요소는 한 단위 시간에 발생하는 시그널의 집합과 상태의 쌍의 값이다. 여기서 상태는 하드웨어 오토마타에서 의미하는 상태로 반드시 모든 TICK은 한 개의 상태만을 가져야 한다. 즉 한 TICK에서 두 개의 상태가 유지될 수 없다. 시그널들은 하드웨어 인터페이스 오토마타에서의 신호들의 집합이고, 아무 신호가 없을 수도 있다. 이 표현은 각 단위 시간에 대한 현재 상태와 발생하는 시그널을 의미하기 때문에 이 표현을 통해 하드웨어 오토마타의 의미를 명확히 할 수 있다.

그림 5에서 하드웨어 인터페이스 오토마타의 기본 전이에 대하여 확장된 정규표현의 형식으로 보여주고 있다. *input op*을 가지는 전이는 입력 신호의 발생에 따라 상태전이가 발생한다. *output*의 *op*를 가지는 전이는 현재 상태에서 출력신호가 한번 발생하면 다음 상태로 전이한다. 반면 *sustain*의 행위를 가지는 전이는 현재 상태에서 출력신호가 한번 발생하면 다음 상태로 전이하고, 다음 상태에서 또 다른 전이가 발생하기 전까지는 계속 출력신호가 유지되어야 한다.

하드웨어 인터페이스 오토마타가 주어졌을 때, 어떤 프로그램으로부터 발생하는 입력신호와 출력신호의 순서가 그 오토마타의 확장된 정규표현으로 표현될 수 있다면 프로그램이 오토마타를 만족한다고 정의할 수 있다. 예를 들어 그림 3(c)에 대해 정규표현으로 표현하여 축약하면 아래와 같다.

$$
\begin{aligned}
extendedRE &::= TICK^{*} \\
TICK &::= [Signals/State] \\
Signals &::= InputSignals \mid OutputSignals \mid \epsilon \\
State &\in Q \\
InputSignals &\subseteq A^{I} \\
OutputSignals &\subseteq A^{O}
\end{aligned}
$$

그림 4 확장된 정규표현

$$
\begin{aligned}
(q_1, a, ?, q_2) &::= [\epsilon/q_1]^{*}[?a/q_2][\epsilon/q_2]^{*} \\
(q_1, a, !, q_2) &::= [\epsilon/q_1]^{*}[!a/q_2][\epsilon/q_2]^{*} \\
(q_1, a, \#, q_2) &::= [\epsilon/q_1]^{*}[!a/q_2]^{+} \\
(q_1, a \wedge b, ?, q_2) &::= [\epsilon/q_1]^{*}[?a, ?b/q_2][\epsilon/q_2]^{*}
\end{aligned}
$$

그림 5 확장된 정규표현을 이용한 하드웨어 오토마타의 전이함수의 표현

따라서 프로그램 수행 중에 `open; open_compl; close; close_compl`의 연속된 시그널 발생했다면 프로그램은 이 명세를 만족하는 프로그램이지만 `open; open_compl; close; open_compl;`이 발생했다면 이 명세를 만족하지 못하는 프로그램이다.

4.2 소프트웨어 인터페이스 명세

소프트웨어 인터페이스 명세는 디바이스 API 호출에 대한 규칙이라고 볼 수 있다. 예를 들어 소프트웨어가 파일 디바이스와 연산을 수행할 때, 소프트웨어에서는 반드시 `open()`을 호출한 이후에 `read()`, `write()` 함수를 호출하고 최종적으로는 `close()`라는 디바이스 API를 호출해야 한다. 다시 말해, `read()`를 `open()`보다 먼저 불러서는 안 된다. 이러한 규칙을 묘사하는 데에도 오토마타를 사용한다. 하드웨어 인터페이스 오토마타를 변형하고 액션의 집합을 디바이스 API의 집합으로 바꾼다면 소프트웨어를 위한 명세 방법으로 사용할 수 있다. 이러한 명세의 정형적인 정의는 우리의 SW 자원 사용 검증 도구인 Pruv-C의 명세와 동일하다.

정의 2: 소프트웨어 인터페이스 오토마타

$swF = < Q, q_0, retVal, fName, \delta >$는 다음의 구성요소를 가진다.

- 유한 상태 집합 Q;
- 시작상태 $q_0 \in Q$;
- 디바이스 API의 리턴값 $retVal$: `true or false`;
- 디바이스 API들의 이름을 의미하는 $fName$;
- $\delta \subseteq Q \times (fName \times retVal) \times Q$의 전이함수;

Q와 q_0는 사용자로부터 의미가 부여된 소프트웨어가 가질 수 있는 상태집합과 시작상태를 의미한다. 오토마타에서의 액션은 하드웨어를 사용할 때 호출하는 디바이스 API 이름과 리턴값의 쌍이다. 그리고 $\delta(q, fName, retVal) = p$는 사상함수의 나타낸다. 여기서 q, p는 상태이고 이는 간단히 $(q, fName, retVal, p)$로 표시할 수 있다. 일단 본 논문에서는 함수의 리턴값은 `true`와 `false`의 부울 값을 갖도록 한정하였다.

그림 6은 소프트웨어 인터페이스 오토마타에 대한 예를 보여주고 있다. 1장에서 보았던 그림 1의 코드에 대한 소프트웨어 설계에 대한 명세이다. 연속으로 두 번 문을 여는 함수나 연속으로 두 번 문을 닫는 함수를 부르지 않도록 프로그램이 설계되어야 한다는 것을 의미하는 간단한 예제이다. 이러한 소프트웨어 인터페이스 오토마타는 정규표현으로 표현할 수 있다. 아래는 그림 6에 대해 정규표현으로 표현한 것이다.

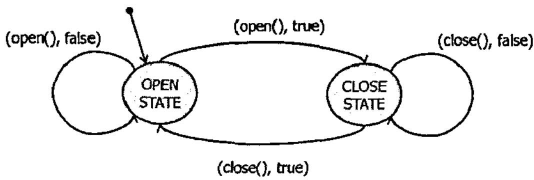


그림 6 소프트웨어 인터페이스 명세의 예

소프트웨어 인터페이스 오토마타가 주어졌을 때, 어떤 프로그램이 이 오토마타를 만족한다는 것은 프로그램으로부터 발생할 수 있는 디바이스 API와 함수의 리턴값의 시퀀스가 이 오토마타의 정규표현의 형태로 기술될 수 있음을 의미한다. 따라서 프로그램 수행 중에 `open()=false; open()=true; close()=true`의 연속된 시퀀스가 발생했다면 프로그램은 이 명세를 만족하는 프로그램이다. 하지만 `open()=true; open()=false;`의 호출 시퀀스가 존재한다면 이 명세를 만족하지 못하는 프로그램이다.

4.3 SW 인터페이스 명세로부터 HW 명세의 추출

하드웨어와 소프트웨어 명세 간에 의미의 차이를 채우기 위해, 우리는 소프트웨어 인터페이스 오토마타로부터 하드웨어 인터페이스 오토마타로의 변환 규칙을 제공한다. 이 규칙에 따라 소프트웨어 명세를 하드웨어 인터페이스 오토마타로 변환하고, 그 후 이 오토마타의 역을 취해 하드웨어 인터페이스 명세를 얻을 수 있다.

소프트웨어 오토마타에서 하드웨어 오토마타로의 변환규칙이 실제적으로 적용 가능하기 위해서 타이밍 이슈에 대해서 고려해야 한다. 그리고 유일한 변환 규칙을 정의하기 위해 우리는 한 가지 가정을 하였다. 고려사항과 가정은 아래와 같다.

- *고려사항: 하드웨어 모듈에서의 한 TICK이 소프트웨어의 함수의 호출보다 빠르다.*
  먼저 하드웨어와 소프트웨어 간의 가장 큰 차이라고 하면 속도이다. 하드웨어 모듈은 TICK에 의해 동기화되어 작동하고, 이 TICK은 소프트웨어의 읽기 연산보다 빠르다. 따라서 소프트웨어가 하드웨어 모듈로부터 데이터를 읽을 때는 하드웨어 모듈에서 소프트웨어가 데이터를 읽었다는 보장이 될 때까지 출력 데이터를 계속해서 유지해야 한다. 즉, 읽기 연산에 대해 3-way handshaking 방식의 통신이 필요하다.
- *가정: 소프트웨어가 하드웨어를 호출할 때 부르는 모든 디바이스 API는 블로킹(blocking) 모델을 따른다.*
  소프트웨어가 하드웨어 모듈에 대해 함수 호출을 한 후, 하드웨어로부터 응답이 있기 전에 바로 리턴 하는 비-블로킹(none-blocking) 모델의 디바이스 API도 있고 하드웨어로부터 응답이 있을 때까지 기다리는 블로킹 모델의 API 함수도 있다. 디바이스 API의 형태에 비례하여 변환규칙이 증가할 수 있다. 현재 연구에서는 디바이스 API는 모두 블로킹 모델이라는 가정 하에 유일한 형태의 변환 규칙을 제공하도록 한다. 이러한 가정은 인터페이스 코드 생성을 통해 실제 구현과 안전성이 보장된다.

```
( [/CLOSED]+ [open/OPENING] [/OPENING]+
[open_compl/OPENED] [/OPENED]+ [close/CLOSING]
[/CLOSING]* [close_compl/CLOSED] )*
```

가정과 고려사항을 적용하여 소프트웨어 오토마타로부터 하드웨어 오토마타를 생성하면, 아래와 같이 하나의 소프트웨어 함수의 전이규칙에 대하여 세 개의 하드웨어 인터페이스의 전이규칙이 생성된다.

$$(q_1, f(), true, q_2) ::= (q_1, f, !, q'_1); (q'_1, f_true, ?, q''_1); (q''_1, f_ack, !, q_2)$$

$$(q_1, f(), false, q_2) ::= (q_1, f, !, q'_1); (q'_1, f_false, ?, q''_1); (q''_1, f_ack, !, q_2)$$

먼저 f()라는 함수의 호출은 시그널 기반의 저수준의 입장에서 보면, f라는 시그널을 하드웨어로 보낸 것이고, 그 결과를 시그널로 받게 된다.(성공이 f_true, 실패면 f_false) 그 후 하드웨어의 모듈에게 받았다는 것을 알리기 위해 ack 신호를 주게 된다.

하드웨어와 소프트웨어가 맞물려 작동하므로, 소프트웨어 명세로부터 변환된 하드웨어 오토마타의 역을 취하면 하드웨어 설계가 지켜야 할 하드웨어 인터페이스 명세가 된다. 다만 앞에 언급한 하드웨어가 빠르다는 고려사항에 따라 소프트웨어의 *input op*의 액션은 하드웨어 인터페이스 명세의 *sustain op*의 액션으로 아래와 같이 변환된다.

지금까지 설명한 소프트웨어 인터페이스 명세로부터 하드웨어 인터페이스 명세로의 구체적인 변환 알고리즘이 그림 7에 나와 있다.

5. 하드웨어/소프트웨어 인터페이스 동시검증

우리는 가정-보증 추론 방식의 검증을 따른다. 일반적으로 가정-보증 추론은 ϕ이라는 가정 하에 속성 Ψ을 만족시키는 프로그램 M과 항상 ϕ을 만족시키는 프로그램 M'가 존재할 때, M과 M'의 병렬 수행은 항상 속성 Ψ을 만족시킨다는 아래와 같은 추론 규칙이다.

$$\frac{<\phi> M <\Psi> \quad <true> M' <\phi>}{<true> M \| M' <\Psi>}$$

우리의 검증 프레임워크에서 M은 하드웨어 프로그램

Algorithm 1. *소프트웨어 인터페이스 명세로부터 하드웨어 인터페이스 명세 추출*

INPUT: $swF = < Q, q_0, retVal, fName, \delta >$

OUTPUT: $hwF = < Q, q_0, op, A^I, A^O, \delta >$

1. 소프트웨어 인터페이스 오토마타를 동등한 의미의 하드웨어 인터페이스 오토마타로 변환한다.

① $hwF = < Q = \varnothing, q_0 = swF$의 q_0, $op = (!, \#, ?)$, $A^I = \varnothing, A^O = \varnothing, \delta = \varnothing >$로 초기화한다.

② swF에서 시작상태를 방문한다. 이 상태에서 한 번도 선택되지 않은 간선을 선택한다. 시작상태 q_0에서 선택된 간선을 $(q_0, f(), true, q_1)$라 하자.

③ q_0상태를 hwF의 상태집합 Q에 추가하고 $f()$와 동일한 이름의 출력신호 f를 A^O에 추가한다.

④ q_0'의 상태를 생성하여 hwF의 상태집합 Q에 추가하고, $(q_0, f, !, q_0')$의 전이규칙을 hwF의 δ에 추가한다.

⑤ 함수명+*true*(이 경우 *f_true*)를 입력신호 A^I에 추가한다. (리턴값이 *false*인 경우는 함수명+*false*)

⑥ q_0''을 생성하여 Q에 추가하고 $(q_0', f_true, ?, q_0'')$의 간선을 δ에 추가한다.

⑦ 함수명+*ack*(이 경우 *f_ack*)를 출력신호 A^O에 추가하고 $(q_0'', f_ack, !, q_1)$을 새 간선으로 추가한다.

⑧ 위의 과정을 BFS 탐색을 통해 방문되는 모든 상태의 모든 전이에 대해서 반복한다.

2. 위에서 얻어진 하드웨어 인터페이스 명세의 역을 취해서 실제로 소프트웨어와 대응하여 동작하는 하드웨어 설계에 대한 명세를 얻는다.

① hwF의 모든 A^O의 원소들과 모든 A^I의 원소들을 바꾼다.

② δ에 대해서 모든 !(*output*)을 ?(*input*)로 바꾸고, 모든 입력을 *sustain* 출력(#)으로 바꾼다.

그림 7 swF 로부터 hwF 추출 알고리즘

이 되고, M'는 소프트웨어 프로그램이다. ϕ과 Ψ는 각각 하드웨어와 소프트웨어가 만족시켜야하는 명세가 된다. 우리는 소프트웨어가 소프트웨어 명세 ϕ을 만족시킨다고 가정하고, ϕ의 상황 하에서만 하드웨어가 명세 Ψ을 만족하는지 검증을 수행한다.

이 장에서는 4장에 정의한 하드웨어/소프트웨어 인터페이스 명세로부터 하드웨어와 소프트웨어 구현에 대한 검증이 어떻게 이루어지는지에 대해서 자세히 설명한다.

5.1 하드웨어 인터페이스 검증

하드웨어 설계에 대한 인터페이스 명세를 검증하기 위해서 하드웨어 분야에 잘 연구가 되어 있는 모델 검증 기술을 적용한다. 우리의 검증 프레임워크에서는 하드웨어 설계를 위해 Esterel 언어를 사용하는데 이 언어를 위한 모델검증기로 Xeve[22]가 존재한다. Xeve는 기존의 모델검증기와는 달리 모델이 아니라 프로그램 코드 자체를 입력으로 받아들이고, 수행 경로를 추적하여 프로그램 실행 동안에 어떤 출력 시그널이 발생할 수 있는지 없는지를 체크해 주는 간단한 기능의 모델검증기이다. 따라서 Xeve를 이용해서 우리의 오토마타를 검증하기 위해서는 오토마타의 대한 명세와 검증할 속성에 대하여 Xeve의 입력형태(여기서는 Esterel 코드)로의 인코딩이 필요하다. 이에 부가적으로 앞에 언급한데로 모델 검증의 정확성을 높이기 위해서 소프트웨어 설계에 대한 모델링을 하여 열린 시스템을 닫힌 시스템으로 만들어 준다. 각각에 구체적인 내용에 대해 아래에 자세히 설명한다.

5.1.1 소프트웨어 모델링

가정되는 소프트웨어 동작을 모델링하여 하드웨어 설계에서 최대한 자유 변수(free variable)들을 제거한다.

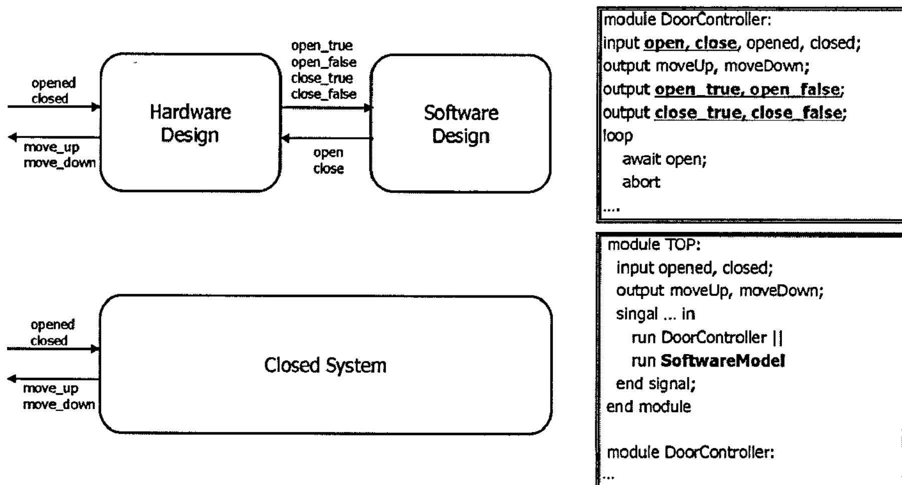


그림 8 소프트웨어 모델링의 예

자유변수는 매 상황마다 조건을 나누게 되어 더 많은 상태를 야기하여 모델검증기의 성능을 악화시키고, 모든 상황을 가정하기 때문에 오경보(false alarm)를 발생할 수 있다. 따라서 시스템을 닫힌 시스템으로 만들기 위해서 소프트웨어의 동작을 모델링 하는 것이 필요하다.

그림 8에서는 이 전의 도어 컨트롤 예제에 대하여 소프트웨어 모델링 후에 어떠한 코드 상의 변화가 있는지를 보여주고 있다. 그림 8의 상단의 그림과 코드는 모델링이 되기 전에 블록 다이어그램과 하드웨어 설계이다. 아래는 가정되는 소프트웨어 동작으로부터 소프트웨어를 모델링 한 후의 블록 다이어그램과 하드웨어 설계이다. 소프트웨어와의 인터페이스 시그널들(밑줄 그어진 시그널들)이 사라짐에 따라 좀 더 정확한 모델검증을 수행할 수 있다.

소프트웨어의 동작을 모델링 하는 것은 주어진 소프트웨어 인터페이스 명세로부터 시작한다. 4장에서 이미 소프트웨어의 명세로부터 동등한 의미의 하드웨어 명세로 변환하는 방법을 설명하였다. 이렇게 변환된 하드웨어 명세들의 각 상태들이 Esterel 코드의 분기문으로 모델링이 된다. 그림 9는 하드웨어 오토마타의 상태들로부터 Esterel 분기문으로 모델링된 코드를 보여준다. 상태가 가지는 전이규칙들의 시그널 발생 조합에 따라 다양한 경우가 나올 수 있는데 그림 9의 응용으로 해결이 되므로 대표적인 3가지 경우에 대해서만 그림 9에 기술하였다. 프로그램은 수행되면서 오토마타에 기술된 상태 중에 반드시 하나의 상태에만 머무른다. 각 상태에서 일어날 수 있는 동작은 출력 신호를 하드웨어 모듈로 발생시키며 다음 상태로 전이하거나, 하드웨어 모듈로부터 입력 신호를 받으며 다음 상태로 전이하는 것이다. 이러한 각 상태에서의 동작들은 그 상태에서 나가는 전이의 종류에 따라 결정된다. 그림 9(a)와 9(b)는 상태로부터 나가는 간선(전이규칙)을 하나만 가지는 상태에 대한 분기문의 모델링이고, 그림 9(c)는 두 개의 전이규칙을 가지는 상태에 대한 모델링 코드이다.

먼저 각각의 상태로의 돌입은 그 상태를 나타내는 시그널이 발생했을 때이다. 상태에 돌입한 이후, 상태가 입력 액션에 대한 하나의 전이 규칙만을 가지면 입력신호가 하드웨어로부터 발생할 때까지 기다리고, 반대로 출력액션에 대한 전이규칙만 있는 경우는 임의의 시간동안 기다린 후 해당 출력 신호를 발생시킨다. 임의의 시간을 표현하기 위해 *trigger*라는 시그널을 도입하였다. 두 개 이상의 전이 규칙을 가진 상태의 경우 둘 중의 먼저 일어난 액션에 따라 상태 전이를 수행한다. 상태가 출력 액션을 가지는 두 개 이상의 전이 규칙을 가지면 그 전이 규칙의 수만큼의 *trigger* 시그널을 도입하고, *trigger* 시그널이 발생하면 해당 출력액션을 발생

(a) $(q_1, a, ?, q_2)$의 전이를 가지는 상태 q_1

$\Rightarrow$

```
case q_1 do             % entry code
    await a;            % wait for presence of a
    emit q_2;           % q1->q2 transition
    emit pre_q_1;        % to remain the pre state
```

(b) $(q_1, a, !, q_2)$의 전이를 가지는 상태 q_1

$\Rightarrow$

```
case q_1 do             % entry code
    await trigger;      % wait for random time
    emit a;             % !a
    emit q_2;           % q1->q2 transition
    emit pre_q_1;       % to remain the pre state
```

(c) $(q_1, a, ?, q_2)$과 $(q_1, b, !, q_3)$ 전이를 가지는 상태 q_1

$\Rightarrow$

```
case q_1 do                 % entry code
    await [a or trigger];   % because of 2 edges
    present a then
        emit q_2;           % q1->q2 transition
    else
        emit b;             % make !b
        emit q_3;           % q1->q3 transition
    end present;
    emit pre_q_1;           % to remain the q1
```

그림 9 하드웨어 오토마타로부터 소프트웨어 모델링

시키며 전이를 한다. 따라서 *trigger* 시그널의 개수는 오토마타 상에서 한 상태에서 나타날 수 있는 최대 출력액션의 출력 간선 수와 동일하다.

5.1.2 하드웨어 명세 모델링: Automata Observer

검증을 위해서 하드웨어 명세에 대한 인코딩이 필요한데 우리는 이것을 Automata Observer로 명명한다. Automata Observer는 두 가지 역할을 수행한다. 첫째, 현재 수행중인 프로그램의 상태를 추적하고 기억한다. 즉, 프로그램이 시작하면 시작상태에 해당하는 시그널을 계속해서 발생시키고, 오토마타 상에서 전이를 일으키는 액션 시그널이 발생하면 명세에 따라 다음 상태에 해당하는 시그널을 계속해서 발생시킨다. 둘째, 하드웨어 명세 상 현재 *sustain* 되고 있어야 하는 시그널이 무엇인지 기억하고 이를 알려준다. 즉, 프로그램이 수행 중에 *sustain* 전이가 발생하면 해당 시그널이 *sustain* 되고 있어야 한다는 것을 의미하는 *sustain_SignalName* 형태의 시그널을 계속 발생시킨다. 이 신호는 다음 전이가 발생할 때까지 유지되어 어느 시그널이 *sustain*이 되어야 하는지를 검사할 수 있다. 그림 10은 하드웨어 명세의 각 상태로부터 생성된 Automata Observer에 대한 Esterel 코드를 보여준다. 상태가 가지는 전이규칙의 시그널 발생 조합에 따라 여러 경우가 존재하지만 그림 10

(a) $(q_1, a, ?, q_2)$ 또는 $(q_1, a, !, q_2)$ 전이를 가지는 상태 q_1
⇒
```
case immediate q_1 do      % entry code
    [
    abort
        pause;
        sustain q_1;        % sustain q_1
    when state_change;     %   for ¬state_change
    ||
    await a;               % ?a
    emit q_2;              % q1->q2
    emit state_change;     % broadcast
    emit pre_q_1;          % remain pre state
    ]
```

(b) $(q_1, a, \#, q_2)$ 의 전이를 가지는 상태 q_1
⇒
```
case immediate q_1 do      % entry code
    ...         ((a)번과 abort ~ when은 동일)
    ||
    await a;               % ?a
    emit sustain_a;        % remain sustain sig
    emit q_2;              % q1->q2
    emit state_change;     % broadcast
    emit pre_q_1;          % remain pre state
    ]
```

그림 10 명세로부터 Automata Observer 생성

의 응용으로 해결되므로 *sustain* 액션을 가지는 경우와 가지지 않는 경우로 나누어 기술하였다.

5.1.3 속성 코드의 생성

하드웨어 인터페이스 명세로부터 검증할 속성들을 자동 생성하여 Esterel 코드의 형태로 인코딩한다. Xeve 모델검증기는 프로그램이 수행되는 동안 어떤 출력 시그널이 발생할 수 있거나 절대로 발생할 수 없다는 정보만을 알려주므로, 속성이 만족하지 못할 경우 특정 시그널을 발생시키는 형태로 코드를 생성해 에러를 검사하도록 한다. 하드웨어 인터페이스 명세가 주어졌을 때, 앞으로 언급된 것과 같이, 네 가지 종류의 속성 코드를 생성한다. 아래에서 속성 명시에 사용된 기호로 *emit()*는 괄호안의 시그널이 현재 클록에서의 발생했음을 의미하고, *pre()*는 바로 직전 클록에서의 발생을, *post()*는 현재 클록이후의 임의의 시간에서의 언젠가 발생한다는 것을 의미한다.

- *Output Violation*:

  하드웨어 인터페이스 명세에 명시된 대로 하드웨어 모듈에서 출력신호를 내보내고 있는지를 검사한다. 출력신호는 하드웨어 명세 상에 명시된 곳에서만 발생하거나 sustain 되어야 할 상황에만 발생해야 한다. 아래와 같은 속성에 대해서 코드가 생성된다.

  $$\text{for } a \in A^O,\quad emit(a) \rightarrow \bigvee_{(q_a, a, !, q_b) \in \delta} (pre(q_a) \wedge emit(q_b)) \vee \bigvee_{(q_a, a \wedge b \in A^O, !, q_b) \in \delta} (pre(q_a) \wedge emit(q_b) \wedge emit(b)) \vee (emit(sustain_a))$$

- *Sustain Violation*:

  *sustain op*를 가진 상태전이가 발생할 때 *sustain* 제약조건이 지켜지는지를 확인하기 위해, 아래와 같은 속성에 대해서 검사 코드가 생성된다.

  $$\text{for } a \in A^O, (q_a, a, \#, q_b) \in \delta,\quad emit(sustain_a) \rightarrow emit(a)$$

- *Deadlock Detection*:

  소프트웨어의 디바이스 API 호출 요청이 발생하면 언젠가는 이 디바이스 API에 대한 응답 신호가 발생해야 한다. 그렇지 않다면 교착상태가 존재한다. 아래와 같은 속성에 대해 교착상태를 검사하는 코드를 생성한다.

  $$\text{for } (q_a, a(), retVal, q_b) \in \delta \text{ in } swF,\quad emit(a) \rightarrow post(a_true) \vee post(a_false)$$

- *Non-deterministic Transition Check*:

  비결정적 전이를 발생시키는 경우를 검사하는 코드가 삽입된다.

  $$\text{for } (q_a, (! \text{ or } \#), a, q_b), (q_a, (! \text{ or } \#), b, q_c) \in \delta,\quad (pre(q_a) \wedge emit(a) \rightarrow \neg emit(b)) \wedge (pre(q_a) \wedge emit(b) \rightarrow \neg emit(a))$$

5.1.4 모델 통합 및 검증

소프트웨어 모델링과 Automata Observer와 속성 코드의 생성이 끝나면, 생성된 이 세 가지 요소와 실제 하드웨어 설계를 통합하여 병렬로 수행한다. 이렇게 통합된 코드는 Xeve의 입력으로 들어가게 되고, Xeve의 실행결과를 해석하여 인터페이스 검증을 수행한다.

5.2 소프트웨어 인터페이스 검증

소프트웨어 설계에 대한 인터페이스 검증을 수행하여 소프트웨어가 하드웨어를 올바르게 사용하고 있음을 검증하고, 하드웨어 인터페이스를 검증할 때 가정했던 소프트웨어의 행위에 대한 보장을 할 수 있다. Pruv-C를 통해서 소프트웨어 설계에 대한 검증을 수행한다. 이 때 소프트웨어 구현 코드 이외에 디바이스 API 코드가 필요하다(디바이스 API 생성에 대해서는 6장에서 다룬다). 이 둘을 입력으로 주고 소프트웨어 명세에 대해 Pruv-C의 입력명세로 인코딩하여 제공하면 소프트웨어 인터페이스 검증을 수행하고 결과를 출력해준다.

6. 인터페이스 코드 생성

그림 11과 같이, 임베디드 시스템은 우리가 검증한 하드웨어 설계와 소프트웨어 설계 외에 디바이스 API, 디

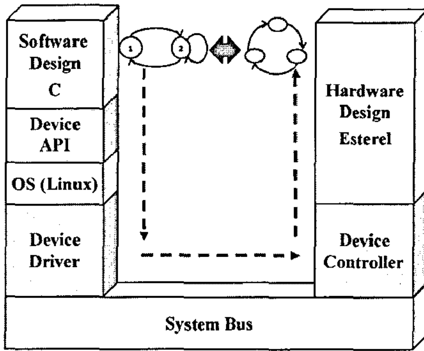


그림 11 인터페이스 코드가 포함된 시스템 구조도

바이스 드라이버, 운영체제, 디바이스 컨트롤러가 필요하다. 우리는 하드웨어 장치는 시스템 버스를 통해 마이크로프로세서와 연결되어 있고, 소프트웨어는 Linux 운영체제 상에서 작동하는 일반적인 임베디드 시스템 환경을 가정한다. 가정한 외부환경을 고려하여, 그림 11에서 음영으로 표시된 박스에 대한 인터페이스 코드를 생성한다. 인터페이스 코드의 생성은 하드웨어와 소프트웨어 명세의 변환에 사용되었던 고려사항들을 만족시킨다.

6.1 디바이스 API 생성

소프트웨어 인터페이스 명세에 기술된 모든 디바이스 API에 대해서 생성하여 검증된 소프트웨어가 바로 컴파일될 수 있게 한다. 보통 소프트웨어에서 하드웨어와 통신은 그 하드웨어를 위해 제작된 디바이스 드라이버의 함수 호출을 통해 이루어지는데 디바이스 API 수준에서의 명세를 제공하고 디바이스 API를 자동으로 생성해줌으로써 디바이스 드라이버 수준에서 개발을 피하게 한

```
∀ API f() ∈ fName in swF
  ⇒
/* 3way: !f  ->  ?f_true or ?f_false  ->  !f_ack  */
// !f
lseek(address of f);
write(f);
// ?f_true or ?f_false
while(1){
   lseek(address of f_true);
   retTrue := read();
   lseek(address of f_false);
   retFalse := read();
   if(retTrue == 1 || retFalse ==1) break;
}
// !f_ack
lseek(address of f_ack);
write(true);
```

그림 12 디바이스 API 생성에 대한 Pseudo 코드

다. 디바이스 API의 생성은 우리가 정의한 명세의 변환룰을 그대로 지원하도록 이루어져 변환의 안전성을 보장한다. 디바이스 API 내부는 일련의 입력액션과 출력액션의 집합으로, 디바이서 드라이버가 제공하는 `read()`, `write()`, `lseek()` 함수의 연속이라고 할 수 있다. 그림 12에 우리가 정의했던 변환 규칙을 만족하는 디바이스 API 생성에 대한 의사 코드가 나와 있다.

6.2 디바이스 드라이버 생성

하드웨어에 대한 주소가 주어지면 그 주소의 해당하는 IO 장치에 원하는 값을 읽고 쓸 수 있도록 범용적인 디바이스 드라이버를 생성한다. 우리가 생성하는 디바이스 드라이버는 Linux 운영체제를 위한 문자형 디바이스 드라이버로 우리가 검증한 하드웨어 프로그램과 소프트웨어 프로그램 사이의 중재자의 역할을 수행한다. 하드웨어 설계와의 통신을 위해, `open()`, `release()`, `lseek()`, `read()`, `write()`, `init_module()`, `cleanup_module()`의 디바이스 드라이버 함수들을 생성한다.

- open(): 소프트웨어의 open() 함수 호출시 하드웨어에 대한 사용권한을 얻을 때 수행되어야 할 코드가 들어간다.
- release(): 소프트웨어의 close() 함수를 호출할 때, 수행될 코드가 들어간다.
- read(): 현재의 오프셋 포지션의 위치로부터 데이터를 읽어서 읽은 데이터를 유저 영역 메모리로 전달하도록 구현되었다.
- write(): 현재의 오프셋 포지션의 위치에 유저 영역으로부터 전달된 데이터를 쓰도록 구현되었다.
- lseek(): 디바이스 드라이버의 오프셋 포지션을 변경한다.
- init_module(): Linux 운영체제에 디바이스 드라이버 모듈에 대해 등록할 때 수행되는 코드이다.
- cleanup_module(): 운영체제에 등록된 디바이스 드라이버 모듈에 대해 등록 해체 될 때 수행하는 일을 구현하였다.

6.3 디바이스 컨트롤러 생성

하드웨어 디바이스에 대한 주소가 주어지면, 이로부터 디바이스 컨트롤러를 생성한다. 그림 13과 같이 소프트웨어와의 통신을 위해 사용되었던 하드웨어 설계상의 모든 인터페이스 시그널들에 대해 순차적으로 주소를 할당한다. 디바이스 컨트롤러는 이 할당된 주소와 하드웨어 설계에서 사용된 시그널과의 매칭을 연결하는 디코더가 된다. 이 전에 언급한 디바이스 API들 또한 이 할당된 주소와 일치하여 하드웨어 모듈에게 신호를 주도록 생성되어 있다.

각각의 시그널에 대해 주소가 할당되었을 때, 그림 14와 같이, 디바이스 컨트롤러는 하드웨어 설계와 실제 시스템 버스 사이에 위치하여 하드웨어로부터의 신호를 소프트웨어가 읽을 수 있는 데이터로 변환하여 전달하

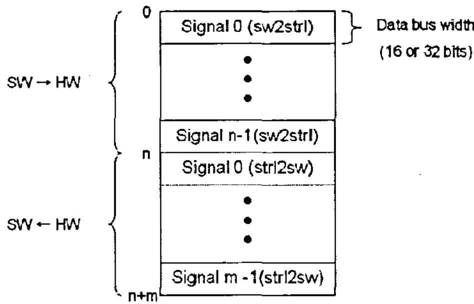


그림 13 하드웨어 시그널에 대한 주소로의 연결

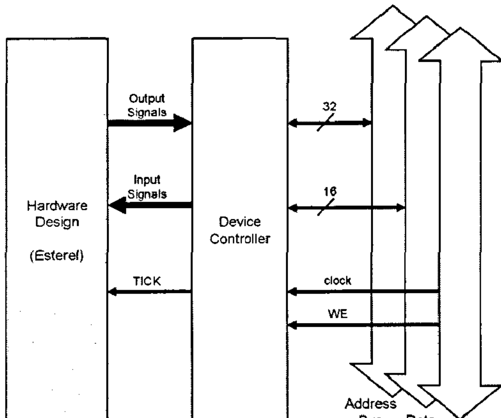


그림 14 디바이스 컨트롤러 구조도

고, 반대로 소프트웨어로부터의 데이터를 하드웨어가 읽을 수 있는 신호로 변환하여 전달한다.

그림 14의 시그널들을 살펴보면 다음과 같다. 먼저 *Output Signal*들은 Esterel 코드에서 소프트웨어와의 통신을 위해 사용되는 출력 신호들이고, *Input Signal*은 소프트웨어로부터의 입력 신호들이다. *TICK*은 Esterel 코드 수행 시 이론적 시간 단위이며, 디바이스 컨트롤러는 실제 시스템의 clock신호(타이머에 의한 실제 하드웨어 수행 클록)를 Esterel 코드의 *TICK*으로 전달하여 하드웨어 디바이스가 시스템과 동기화 되어 동작할 수 있도록 한다. 하드웨어 설계로부터 *Output Signal*이 발생하면, 디바이스 컨트롤러는 이 신호가 연결되어 있는 주소버스로 1을 보냄으로써 소프트웨어가 그 주소를 액세스하였을 때 해당 시그널이 발생하였음을 알게 한다. 반대로 WE 신호가 들어오면 소프트웨어로부터의 입력 신호가 발생하는 것이므로 그 주소 번지와 연결되어 있는 입력 시그널에 대한 신호를 발생시켜 하드웨어 모듈이 소프트웨어로부터의 신호를 전달받을 수 있도록 한다.

7. 구현 및 실험

7.1 동시검증 기법의 구현

이 논문에서 제안하는 알고리즘은 Java로 구현하였고, 우리가 기존에 연구한 개발도구[23]와 통합하여 Eclipse [24] Plug-in의 형태의 일체화된 동시 개발 환경을 제공한다. 그림 15는 구현된 개발 도구의 모습이다. 이 도

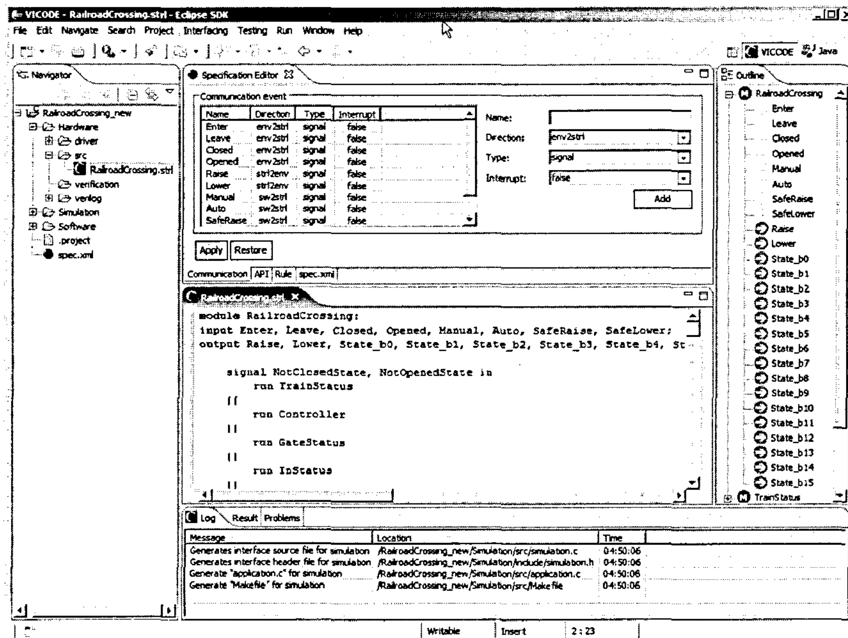


그림 15 구현된 동시 검증 도구

구는 이 논문의 알고리즘뿐만 아니라, 코드 개발을 위한 편집기, 파서, 시뮬레이션 기능들을 제공한다.

7.2 Case Study : 자동 주차 시스템 제어기

구현된 알고리즘의 평가를 위해, 우리 툴을 이용하여 자동 주차 시스템 제어기를 개발하고 검증하였다. 또한 주차 시스템에 대한 실제 기계장치들을 레고(Lego) 블록과 센서와 모터를 이용하여 제작하였다. 자동 주차 시스템은 사용자의 요구에 따라 자동차를 주차타워에 주차시키거나, 주차타워에서 출고시키는 자동화 시스템이다.

그림 16에 우리가 설계하려는 주차 시스템에 대한 구조도가 그려져 있다. 전체 설계가 하드웨어와 소프트웨어 파트로 나누어져 있기 때문에 둘 사이에 통신하는 신호들이 필요하다. 이 시스템에서 소프트웨어의 역할은 주차 시스템의 스케줄링 관리이다. 소프트웨어는 현재의 주차 시스템의 상태를 파악하고, 유저의 요청에 따라 실제 하드웨어에게 명령을 내린다. 소프트웨어 모듈은 잘못된 사용자의 명령은 필터링하여 하드웨어한테 지시되는 일이 없도록 해야 한다. 반대로 하드웨어의 입장에서는 소프트웨어로부터 명령이 들어오면 명령에 따라 하드웨어 장치들을 올바르게 움직이고, 그 결과를 소프트웨어에게 반환해야 한다. 예를 들어, 차의 출고 명령이 들어왔다면 먼저 문을 열고 차를 완전히 뺀 후에 차가 완전히 빠졌다는 것이 보장될 때 문을 닫아야 한다.

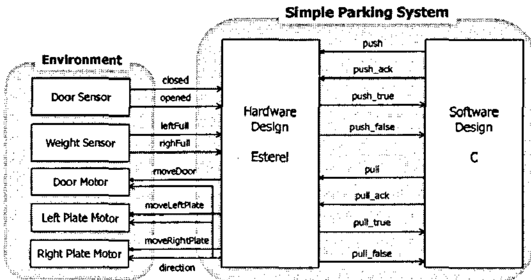


그림 16 주차 시스템 구조도

7.2.1 주차 시스템에 대한 인터페이스 명세

우리 프레임워크에서 시스템의 구조도가 결정된 후 가장 먼저 해야 할 일은 소프트웨어의 입장에서 상위수준의 명세를 하는 것이다. 이 주차 시스템의 경우는 간결하게 그림 17과 같이 기술될 수 있다. 소프트웨어는 스케줄링을 담당하므로, 차가 입고되어 있을 때만 pull 함수를 호출할 수 있고, 차가 주차공간에 없을 때만 push 함수를 호출할 수 있도록 제약을 주고 있다. 그림 17의 소프트웨어 인터페이스 명세로부터 그림 18과 같은 하드웨어에 대한 인터페이스 명세가 자동으로 생성된다.

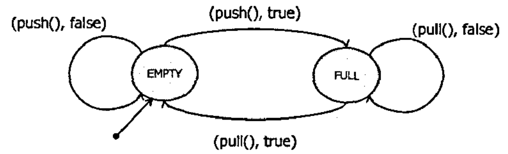


그림 17 주차 시스템에 대한 SW 인터페이스 명세

7.2.2 검증 및 인터페이스 코드 자동생성

소프트웨어 명세의 입력과 이로부터 하드웨어 명세가 생성이 되면, 이 명세들에 맞추어 하드웨어와 소프트웨어 설계에 대해 구현을 한다. 구현이 끝나면 구현 코드들에 대해 검증을 수행한다. 검증이 통과된다면 주차 시스템에 대한 인터페이스 코드들이 생성된다. 아래의 순서대로 코드가 검증되고 코드가 생성된다.

- 소프트웨어 모델링: 소프트웨어 명세로부터 Esterel 코드 M'이 생성된다.
- 하드웨어 명세 모델링: 하드웨어 명세를 인코딩하여 Esterel 코드 A'가 생성된다.
- 속성 코드 생성: 명세로부터 검증할 속성들이 추출되어 Esterel 코드 P'가 생성된다.
- 하드웨어 인터페이스 검증: 하드웨어 설계를 생성된 코드들과 병렬로 실행하여 검증을 수행된다. ($Code \| M' \| A' \| P' \models \Psi$)
- 소프트웨어 인터페이스 검증: 디바이스 API를 생성하

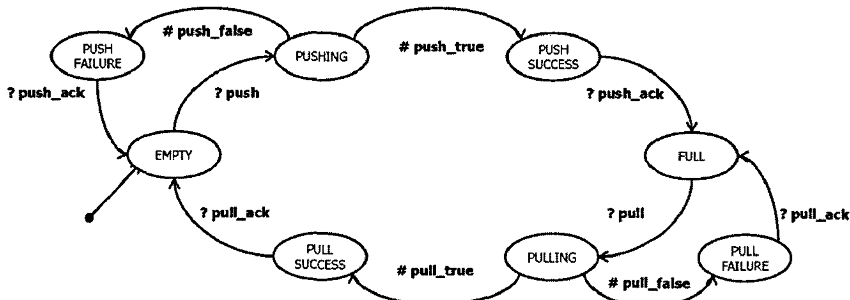
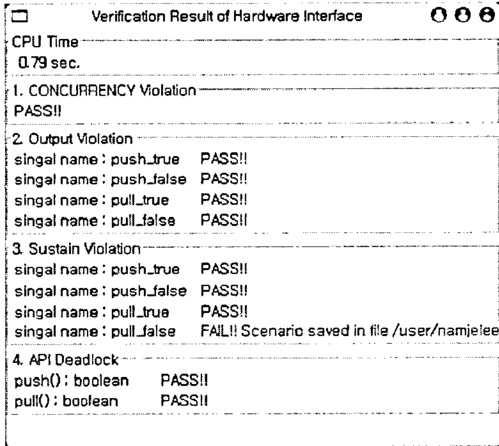
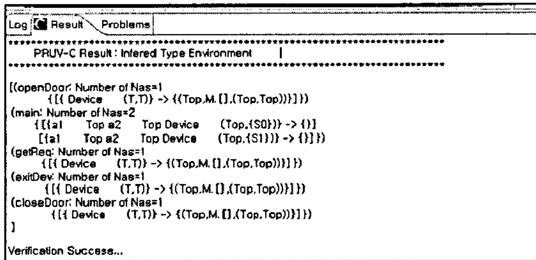


그림 18 주차시스템에 대한 HW 인터페이스 명세



(a) 하드웨어 인터페이스 검증 결과



(b) 소프트웨어 인터페이스 검증 결과

그림 19 하드웨어/소프트웨어 동시 검증 결과

고 소프트웨어 명세와 소프트웨어 설계를 입력으로 하여 검증이 수행된다.

- 통합을 위한 인터페이스 코드들을 생성한다.

그림 19는 주차 시스템에 대한 하드웨어와 소프트웨어 인터페이스 검증 결과이다.

7.2.3 시스템의 통합

구현된 코드와 생성된 인터페이스 코드로부터 아래 단계를 통해 시스템을 구축할 수 있다.

- 하드웨어 설계에 대하여 Esterel 컴파일러를 이용하여 Verliog로 컴파일 한다.
- 컴파일된 하드웨어 설계와 생성된 디바이스 컨트롤러 코드를 통합하여 FPGA와 같은 하드웨어 장치에 굽는다.
- 디바이스 드라이버를 컴파일 한 후 Linux 운영체제에 모듈로 적재한다.
- 소프트웨어 설계와 생성된 디바이스 API 코드를 함께 바이너리 파일로 컴파일 한다. 이제 소프트웨어를 실행시키면 소프트웨어와 하드웨어가 통합된 주차 시스템이 동작한다.

이 시스템은 실제로 EMPOS II[25](ARM기반의 마이

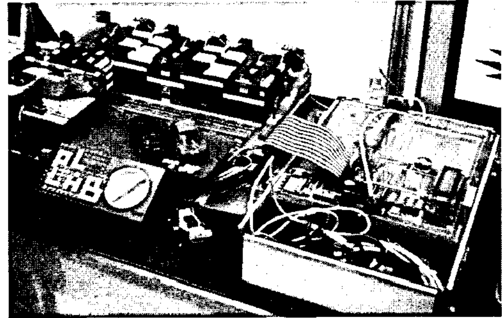


그림 20 구현된 자동 주차 시스템

크로프로세서를 포함하는 임베디드 시스템 개발보드), EMP2CYC[25](FPGA 보드), 레고 마인드스톰[26]의 블록, 모터, 센서들만을 이용하여 구현하였다(레고 마인드스톰에서 제공하는 모터/센서 컨트롤러는 사용하지 않았다). EMPOS II에는 FPGA가 확장포트를 통해 장착되어 있어 하드웨어 모듈을 적재할 수 있다. 그림 20은 주차 시스템에 대한 실제 개발한 프로토타입 화면이다.

7.2.4 실험 결과

주차 시스템 제어기가 작은 프로그램이었고 간단한 명세를 주었기 때문에 검증 소요시간은 하드웨어 검증에 0.79초와 소프트웨어 검증에 0.3초가 소요되었다.

표 1은 하드웨어 검증을 위해 생성된 정보에 대한 분석 테이블이다. 2개의 상태를 가진 소프트웨어 명세로부터 생성된 하드웨어 명세는 3way-handshaking이 적용되어 3배 정도 확장된 크기를 가지는 것을 볼 수 있다. 다음으로 소프트웨어 모델링의 경우를 보면, 하드웨어 설계에서 자유 변수가 반으로 줄어드는 효과를 보였다. 속성 생성을 살펴보면, 총 11개의 검사 속성을 생성하여 검증하였음을 볼 수 있다. 문제는 이러한 모델링의 결과로 최종 모델검증기의 입력 코드의 크기가 커진다는 것이다. 따라서 이 부분에 대한 개선이 향후 필요하다.

표 2는 개발자가 개발한 소프트웨어/하드웨어 설계와

표 1 HW 검증을 위한 생성 정보 분석

생성된 HW 명세		소프트웨어 모델링		명세 모델링	속성 생성	
상태수	간선수	FV변화	LOC	LOC	속성수	LOC
8	10	8->4	84	163	11	80

표 2 구현코드와 자동 생성된 인터페이스 코드 비교

작성코드 LOC	HW Code (Esterel)		SW Code (C)	Total
	92		43	135
생성코드 LOC	디바이스 컨트롤러	디바이스 드라이버	디바이스 API	Total
	96	112	133	341

검증 도구가 생성한 인터페이스 코드의 라인 수에 대해 비교한다. 이 예제의 경우는 간단한 시스템이기 때문에 실제 주차시스템에 대한 구현코드(135줄)보다 자동 생성된 하드웨어/소프트웨어 사이의 통신에 관련된 일을 수행하는 인터페이스 코드(341줄)가 2배 이상 크다. 인터페이스 코드가 실제 임베디드 시스템 개발 시 에러를 야기하기 쉬운 부분이므로 이에 대한 자동생성은 의미를 가진다고 볼 수 있다.

8. 결 론

본 연구에서는 하드웨어와 소프트웨어 사이의 상호작용을 기술할 수 있는 인터페이스 명세를 정의하고, 구현된 코드가 명세를 지키는지를 검증하는 하드웨어/소프트웨어 동시 검증 기법을 제안하고 구현하였다. 우리 논문의 기여도를 요약하면 다음과 같다.

첫째, 하드웨어와 소프트웨어 설계에 대한 인터페이스를 기술할 수 있는 추상화된 두 개의 인터페이스 명세를 제공하고 이를 검증할 수 있는 방법을 제공한다.

둘째, 오토마타 형식으로 정의된 인터페이스를 모델검증을 통해 검증할 수 있는 기법을 제안한다. 가정되는 소프트웨어의 동작으로부터 외부환경에 대한 모델을 추출하여 보다 정확한 모델검증을 수행할 수 있다.

마지막으로, 현실적인 합성 가능한 검증 기법을 제공한다. 기존의 검증 기법들이 검증을 한 모델과 실제 코드 사이에 차이가 존재하는 반면, 코드를 직접 검증하고, 검증된 코드가 바로 통합될 수 있도록 디바이스 API, 디바이스 드라이버, 디바이스 컨트롤러의 인터페이스 코드들을 생성해 주어 실제적인 해결책을 제시한다.

사례연구로써, 본 연구의 알고리즘을 구현한 도구를 이용하여 주차 시스템 제어기를 제작 및 검증하였다. 검증하려는 명세의 크기에 비례하여 검증 도구의 입력 데이터가 커진다는 단점이 있지만 앞으로 개선이 이루어질 것이고, 고도의 안전성을 요구하는 제어 관련 임베디드 시스템 개발에 효과적으로 사용될 수 있는 동시검증 방법이 될 수 있다.

참 고 문 헌

[1] E.A. Lee, "What's ahead for embedded software?" *IEEE Computer*, vol.33, Issue.9, pp.18-26, Sep. 2000.

[2] A.A. Jerraya and W.Wolf, "Hardware/software interface codesign for embedded systems," *IEEE Computer,* pp.63-69. Feb. 2005.

[3] Daniel GroBe, Ulrich K¨uhne, and Rolf Drechsler, "HW/SW co-verification of embedded systems using bounded model checking," *GLSVLSI '06: Proceedings of the 16th ACM Great Lakes symposium on VLSI*, pp.43-48, 2006.

[4] J. Buck, S. Ha, E. Lee, and D. Messerschmitt, "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems," *International Journal in Computer Simulation*, vol.4, no.2, pp.155-182. 1994.

[5] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A, Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara, Hardware-Software Co-Design of Embedded Systems: The Polis Approach. Kluwer Academic Press. 1997.

[6] Synopsys Inc., Eaglei, http://www.synopsys.com/products.

[7] Mentor Graphics Inc., Seamless CVE. http://www.mentor.com/seamless.

[8] L. Benini, D. Bertozzi, D. Bruni, N. Drago, F. Fummi, and M. Poncino, "SystemC Cosimulation and Emulation of Multi-Processor SoC Designs," *IEEE Computer*, vol.36, no.4, pp.53-59 2003.

[9] L. Formaggio, F. Fummi, and G. Pravadelli, "A Timing-Accurate HW/SW Co-simulation of an ISS with SystemC," *Proceedings of IEEE International Conference on Hardware/Software Codesign and System Synthesis*, pp.152-157. 2004.

[10] Kudlugi, M. Hassoun, S. Selvidge, C. Pryor, D., "A transaction-based unified simulation/emulation architecture for functional verification," In *Proceedings of Design Automation Conference*, 2001.

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

[12] 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.

[13] T. Ball, S. K. Rajamani, "The SLAM Project: Debugging System Software via Static Analysis," *Proceedings of Principles of Programming Languages*, 2002.

[14] L. de Alfaro and T.A. Henzinger, "Interface theories for component-based design, In *Proc. Embedded Software*, Lecture Notes in Computer Science 2211, pp.148-165. Springer-Verlag, 2001.

[15] Hyun-Goo Kang, Youil Kim, Taisook Han, and Hwansoo Han, "A Path Sensitive Type System for Resource Usage Verification of C like Languages," In *the 3rd Asian Symposium on Programming Languages and Systems*, November 2005.

[16] K.L. McMillan. Symbolic Model Checking. Kluwer Academic Publisher, 1993.

[17] G. Berry, M. Kishinevsky and S. Singh, "System Level Design and Verification Using a Synchronous Language," *Proceedings of the 2003 IEEE/ACM International Conference on Computer-Aided Design*, p.433, November 09-13, 2003.

[18] Z. Manna and A. Pnueli, The Temporal Logic of Reactive and Concurrent Systems: Specification, Springer-Verlag, 1996.
[19] L. de Alfaro and T.A. Henzinger, "Interface-based design," In Engineering Theories of Software-intensive Systems (M. Broy, J. Gruenbauer, D. Harel, and C.A.R Hoare, eds.), NATO Science Series: Mathematics, Physics, and Chemistry, vol.195, Springer, pp.83-104, 2005.
[20] O. Tkachuk, M. Dwyer, C. Pasareanu, "Automated environment generation for software model checking," In Proceedings of the 18th IEEE International Conference on Automated Software Engineering, pp.116-127, 2003.
[22] A. Bouali, "Xeve: an Esterel verification environment," Technical report, INRIA, Dec. 2000.
[23] VICODE(Verification Integrated Codesign Environment), http://plus.kaist.ac.kr/~vicode, 2008.
[24] Eclipse, http://www.eclipse.org/
[25] Hanback Electronics, http://www.hanback.co.kr/
[26] Lego Mindstorm, http://mindstorms.lego.com/

이 재 호

1998년~2006년 한성대학교 컴퓨터공학과 학사. 2006년~2008년 한국과학기술원 전산학과 석사. 2008년~2009년 삼성전자 기술총괄 소프트웨어 연구소. 2009년~현재 삼성전자 DMC 연구소 SE Lab. 관심분야는 프로그램 분석, 임베디드 시스템

한 태 숙

1976년 서울대학교 전자공학과 학사. 1978년 KAIST 전산학과 석사. 1990년 Univ. of North Carolina at Chapel Hill 박사 1991년~현재 KAIST 전산학전공 교수 관심분야는 프로그래밍 언어, 함수형 언어, 임베디드 시스템

윤 정 한

2001년 KAIST 전산학전공 학사. 2003년 KAIST 전산학전공 석사. 2003년~현재 KAIST 전산학과 박사과정. 관심분야는 임베디드 시스템, 프로그램 분석