VIS 를 이용한 I-Link Bus 중재 프로토콜의 정형검증 [1)]

엄현선*, 최진영*, 한우종**, 기안도**
고려대학교 컴퓨터학과*, 한국전자통신연구원 컴퓨터시스템연구소**
e-mail : hum@formal.korea.ac.kr

Formal Verification of I-Link Bus arbiter Protocol Using VIS

Hyun-Sun Um*°, Jin-Young Choi*, Woo-Jong Han**, An-Do Ki**
Department of Computer Science and Engineering, Korea University*,
Computer System Department, ETRI-CSTL**

요 약

시스템이 복잡해짐에 따라 현재 사용되고 있는 무작위적 테스트나 시뮬레이션은 프로토콜의 정확성을 확인하기에 충분하지 못하므로 보다 효율적이고 믿을 만한 검증 방법이 필요하다. 본 논문은 ETRI 에서 개발한 디렉토리 기반 CC-NUMA 시스템의 CCA(Cache Coherent Agent)보드 내부 버스인 I-Link(Inside Link) 버스의 중재 프로토콜을 정형 검증에 쓰이는 도구 중의 하나인 VIS(Verification Interacting with Synthesis)를 이용하여 검증한다. VIS 는 Verilog 입력을 받는 도구이므로 개발 단계에서 만들어진 소스를 그대로 이용하여 검증하는 기법을 사용하였고 이를 통해 보다 정확한 명세와 검증을 할 수 있었다.

1. 서론

하드웨어나 소프트웨어 시스템은 불가피하게 점점 더 규모, 기능면에서 대형화되고 복잡해지고 있다. 이로 인해 시스템에서 신뢰성이 차지하는 비중이 높아짐에 따라 설계 시 정확한 명세와 그에 따른 검증을 필요로 하게 되었다. 설계의 검증이 완전하지 않은 상태에서 시스템을 구현하게 되면 시스템의 오류로 인하여 문제가 발생할 가능성이 크기 때문이다. 따라서 비용, 인력, 시간면에서 엄청난 손실 또한 가져오게 되었다. 예를 들어 인텔의 펜티엄 프로세스의 FDIV명령어 오류와 아리안 로켓의 폭발등에서 알 수 있듯이 이런 에러들은 엄청난 손실을 가져왔다.[2] 에러를 발견하고 나서 수정하는 것이 어렵고 비용 또한 많이 들기 때문에 시스템을 구현하기 전에 어떤 설계의 정확성을 확인하는 것은 중요하다. 현재 설계 단계에서 에러를 찾기 위해 사용하는 방법인 시뮬레이션은 단순한 개념이지만 불완전하다는 문제를 가지고 있다. 이와는 달리 정형기법[1]은 쉽게 찾아내기 힘든 불일치성, 애매모호함, 불완전성을 나타내 보임으로써 보다 완벽한 시스템을 구축 할 수 있게 한다. 이런 문제들이 정형기법의 중요성을 강조 하고 있고 점차 기존에 사용되던 시뮬레이션 방법을 대체할 방법으로 정형검증에 대한 관심을 갖게 되었다. 특히 근래에 자동적으로 검증 할 수 있는 도구들이 개발되고 있어 학계에서 뿐만 아니라 산업계에서 설계하는데 이용할 수 있기 때문에 최근 들어 상당한 관심을 받아 오고 있다. 정형기법은 정형 명세와 정형 검증으로 나뉘고 정형 검증에서 최근 많이 사용되고 있는 모델 체킹은 시스템을 명세언어로 명세하고 명세한 시스템 모델에 대해 궁극적으로 검증하려는 특성이 만족하는지를 CTL[4]과 같은 논리식으로 표현하여 검증하는 기법이다. VIS[9]는 CTL모델 체킹[4]과 language emptiness검사 등을 할 수 있는 정형 검증 도구이다. VIS를 이용하여 검증하기 위해서는 우선 formal model을 만들어야 하며 이를 위해 검증할 대상에 대한 문서를 정확하게 이해하고 그 문서대로 명확하게 명세하는 것이 필요하다. 지금까지는 VIS를 이용할 때 검증할 대상에 대해 추상화 시킨 후 finite state machine을 그리고 그에 대해 VIS의 입력 언어인 Verilog[5]를 이용하여 명세하여 검증을 하였다. 그러나 이러한 방법을 이용할 경우 명세자의 오류가 있을 경우 정확한 검증을 하였다고 확신 할 수 없다는 문제가 발생할 수 있다. 따라서 이 논문에서는 실제 구현에 사용 될 Verilog 소스를 VIS의 입력으로 사용하고자 하였다. 이 방법을 이용할 경우 이전 방법을 이용할 때의 단점을 극복 할 수 있고 더 정확한 검증을 할 수 있을 것이다.

본 논문에서는 VIS 를 이러한 방법으로 이용하여 ETRI 에서 개발된 디렉토리 기반의 CC-NUMA 시스템의 CCA(Cache Coherent Agent) 보드 내부 버스인 I-Link(Inside Link) 버스에서 중재역할을 하는 중재기를 검증하고자 한다. 중재기의 몇 가지 특성들을 검증한 사례를 제시한다. 2 장에서는 VIS 에 대한 소개와 CTL 과 모델 체킹에 대해 설명하고 3 장에서는 I-Link 버스[13]와 중재기에 대한 대략적인 소개를 하고 4 장에서 VIS 를 이용하여 중재기를 검증한 결과에 대한 설명을 하고 5 장에서 검증 결과에 대한 결론을 맺고 모델 체킹 검증 기법의

[1)] 본 논문은 1999 년 한국전자통신연구원 위탁과제 99379 의 지원으로 연구 되었음

장점과 향후 과제를 제시한다.

2. VIS

2.1 VIS 에 대한 소개

VIS(Verification Interacting with Synthesis)는 verification, simulation, 유한 상태 하드웨어 시스템의 synthesis 를 종합한 도구(tool)이다. 이 도구는 입력 언어로 Verilog 를 사용하고 fair CTL 모델 체킹, language emptiness 검사, combinational and sequential equivalence equivalence checking)조합 순차 동등성 검사, 원형 기반 시뮬레이션(cycle-based simulation)과 계층적 체계화(hierarchical synthesis)를 지원한다. VIS 는 Berkeley 대학과 콜로라도 대학(Boulder) 과 협력하여 개발 되고 있으며 1 세대 도구인 HSIS 나 SMV 에 비교하여 보다 개선된 프로그래밍 환경을 제공하며, 새로운 호환성을 제공하며 어떤 경우에는 성능 또한 향상 시켜준다.

VIS 는 BLIF-MV[10]라는 중간 형태에서 작동하며 BLIF-MV 파일은 VL2MV 라 불리는 컴파일러에 의해 생성된다. VIS 에서 사용하는 Verilog 가 일반적인 Verilog 와 다른 점은 비결정성(nondeterminism)과 기호적 변수(symbolic variables)를 사용한다는 것이다. 비결정성 구성자인 $ND 는 wire 변수에서의 비결정성을 나타내기 위해서 Verilog 에 추가 되었다. 이것이 VIS 에서 비결정성을 자연스럽게 사용할 수 있도록 하였다. 또한 변수의 값을 기호적으로 명세하고 참조하는 것이 직접적으로 표현하는 것보다 바람직한 경우가 있다. VL2MV 가 Verilog 에서 C 에서 사용하는 것과 유사한 열거형을 사용하여 기호적 변수를 표현할 수 있도록 하였다.

BLIF-MV 묘사가 VIS 로 읽혀질 때 계층적 트리 형태로 저장된다. 이는 차례로 부모듈(sub module)을 구성하고 모듈의 기능은 arbitrary functionality 와 latch 를 가진 게이트의 네트워크에 의해 나타내어진다. 이 계층에 대한 순회(traversal)는 UNIX 의 디렉토리에 대한 순회와 유사하고 시뮬레이션과 검증 작업은 어느 계층에서나 가능하다.

2.2 VIS 를 이용한 검증

VIS 를 이용하여 검증하기 위해서는 먼저 간단하게 검증 할 수 있는 형태로 바꿔야 한다. 상태와 상태 사이의 전이는 유한 상태 기계(finite state machine)로 구성 되고 완전한 시스템은 각 구성 요소와 연관된 유한 상태 기계를 구성함으로써 얻어지는 유한 상태 기계이다. 그러므로 검증할 때 첫번째 단계는 시스템을 유한 상태 기계로 나타내는 것이다. 유한 상태 기계로 나타낸 시스템을 VIS 의 입력 언어인 Verilog 로 설계 한 후 VIS 를 이용하여 현재 자동 정형 검증에서 가장 많이 쓰이는 두 가지 방법인 language containment 와 모델 체킹(model checking) 방법으로 검증한다.

2.3 CTL (Computational Tree Logic) 을 이용한 명세와 모델 체킹

모델 체킹은 시스템에 대한 유한 모델을 만들고 이 모델 상에서 만족해야 하는 특성을 검사하는 것으로 주로 하드웨어와 프로토콜 검증에 이용되고 있다. 따라서 어떤 시스템을 정형 검증하기 위해서는 먼저 시스템이 검증 가능한 간단한 형태로 변환 되어야 한다. 이를 위해 시스템을 하나의 큰 유한 상태 기계로 구성하고 이 속에는 각각의 상태를 갖고 있는 작은 유한 상태 기계가 있어 이들 간의 상호 관계로 시스템이 작동한다. 이런 성질은 우리가 실세계에서 접하는 시스템을 모델링 하기 좋은 데 그 이유는 계층을 쉽게 표현 할 수 있기 때문이다. 모델 체킹 방법은 유한 상태기계로 표현된 모델에 대해서 검사하므로 종료가 보장되고 이로써 상태 공간의 모든 경우에 대해서 검사를 하게 된다. 시스템에 대해 검사하려는 특성은 시제 논리(temporal logic)로 표현할 수 있다.

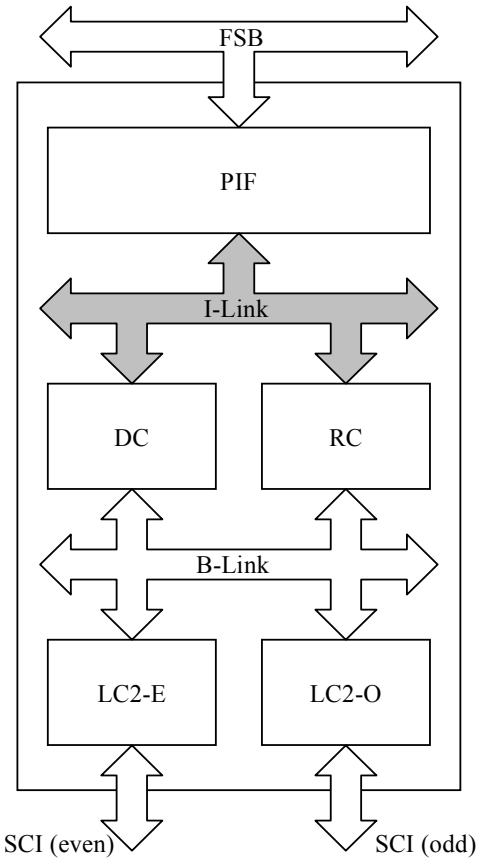
시제 논리는 시간을 표현할 수 있는 연산자(operator)를 사용하여 특성을 시간에 따른 사건들의 순서로 표현할 수 있다. 시제 논리에는 크게 LTL(Linear Temporal Logic)과 CTL(Computational Tree Logic)이 있다. LTL 은 하나의 시간흐름에 대해서만 고려하는 방법이고 CTL 은 트리 형태로 여러 갈래로 나뉘어지는 시간 흐름에 대한 방법이다. VIS 에서 사용하는 방법은 CTL 이다. 계산 트리(computation tree)는 상태전이 그래프로부터 유도된다. 상태 그래프는 초기 상태로부터 무한한 경우를 나타내는 트리로 표현된다. 이 트리에서 나타나는 모든 경로(path)는 모델링 되는 시스템의 모든 가능한 계산을 표현한다. CTL 은 이러한 트리와 같은 분기(branch) 구조를 묘사하는 오퍼레이터(operator)를 가지고 있기 때문에 분기 시간(branching time)논리로 분류된다. CTL 로 표현되는 식은 단위 명제(atomic proposition)와 명제논리의 부울 합성자(boolean connectives), 시제 오퍼레이터로 구성된다. 각 시제 오퍼레이터는 두 부분으로 구성되어지는데 경로 한정자(path quantifier A,E)와 시제 구성자(temporal modality F, F, X, U)이다. 각 formula 가 q 라는 상태에서 적용된다고 가정하면 경로 한정자에서 A 는 "q 에서 시작하는 모든 경로"라는 의미이고, E 는 "q 에서 시작하는 어떤 한 경로"를 나타낸다. 시제 구성자에서 F 는 " 해당 경로의 어떤 한 상태", X 는 "다음에", G 는 "해당 경로의 모든 상태"를 의미하며, U 는 "~일 때 까지"를 나타낸다. 그래서, AG safe 가 q 에서 만족되려면, q 에서 시작하는 모든 경로(A)의 모든 상태(G)들이 safe 라는 formula 를 만족해야 한다. 또한, AF safe 가 q 에서 만족되려면, q 에서 시작하는 모든 경로(A)에 대해, 각 경로에서 적어도 하나의 상태(F)가 safe 를 만족해야 하며, EG safe 가 q 에서 만족되려면, q 에서 시작하는 경로 중 적어도 하나의 경로(E)의 모든 상태(G)가 safe 를 만족해야 한다. 마지막으로, EF safe 가 q 에서 만족되려면, q 에서 시작하는 경로 중 적어도 하나의 경로(E)에서 적어도 하나의 상태(F)가 safe 를 만족해야 한다.

그러나 시스템이 여러 사용자 사이에 공유된 자원을 할당한다면 사용자가 보유하지 않은 자원 사이에 있는 경로들도 고려되어야 한다. CTL 로만은 fair path 사이의 정확성에 관해 나타낼 수 없다. Fair CTL 은 fairness 를 다루는 CTL 의 변형이다. Fair path 는 무한하게 자주 만족되는 각각의 fairness 조건 사이에 있는 경로이다. VIS 는 fair CTL 을 지원한다. 또한 CTL 에서 묘사 될 수 없는 실제적인 특성이 있다. 예를 들어 "거의 항상 (almost always)" 이라는 특성이다. 현재 VIS 는 language containment 의 제한된 형을 지원한다. language containment 검사를 수행하기 위해서는 원하는 특성의 반대를 나타낸 모델로 주어진 시스템을 나타내고 language emptiness 를 위해 검사 한다. VIS 내부에서는 language emptiness 는 CTL 에서의 EG true 를 검사 한다.

3. I-link 버스

3.1 I-link 버스의 개요

Cache Coherent Agent(CCA)보드[12]는 그림[1]에서와 같이 Intel SHV 내부 버스인 Front Side Bus(FSB)와 Scalable Coherent Interface(SCI)를 연동시키는 기능을 담당한다. 이를 위해 보드 내부에는 FSB 와의 인터페이스를 담당하는 PIF(Processor Interface)와 RACE 프로토콜을 위한 DC(Directory Controller)와 RC(Remote Access Cache Controller), SCI 전송계층 프로토콜을 담당하는 LC2 로 구성된다. 특히 이중 SCI 네트워크를 위해 LC2 는 LC2-E(LC2-Even)와 LC2-O(LC2 Odd) 두개로 구성된다. 이들 내부 모듈들(PIF, DC, RC, LC2-E, LC2-O)사이는 버스로 연결되는데 PIF 와 DC/RC 사이는 I-Link(Inside Link), DC/RC 와 LC2-E/O 사이는 B-Link 라 한다. I-Link 버스는 동기형 버스이며 패킷 전송을 이용한 분리형 프로토콜을 사용하며 우선순위를 이용하는 중앙 집중형 중재기법을 사용한다.



[그림 1] CC-Agent Board Block Diagram

3.2 중재 프로토콜

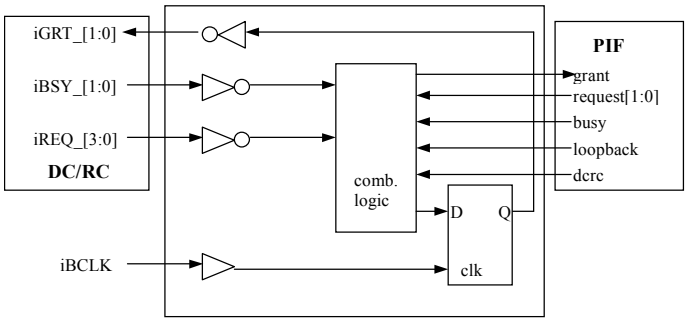
I-Link 버스에서 사용하는 중재방법은 우선순위를 이용한 중앙 집중형 중재기법을 사용한다. 중재가 필요한 모듈은 자신에게 할당된 중재 요청 신호 iREQ_[x]를 구동하여 중재 허가 신호 iGRT_[y]를 기다린다. 중재 허가 신호를 받은 모듈은 이어지는 바로 다음 버스 주기부터 원하는 만큼 버스를 사용하며 이 때 중재 요청 신호를 계속 구동함으로써 버스를 계속 사용함을 알린다. 따라서 중재 요청 신호와 중재 허가 신호가 구동 되어 있는 동안은 비록 우선 순위가 높은 모듈이 중재를 요청해도 중재 결과는 변하지 않는다. I-Link 버스를 효율적으로 사용하려면, 마지막 데이터 전송 주기 한 사이클 이전까지 만 iREQ_[x] 신호를 구동해야 된다. 이런 경우 한 개 보호 사이클(guard cycle) 이후 곧바로 데이터 전송 버스를 다른 모듈이 사용할 수 있게 된다.

3.3. 중재기 구현

중재기[14]를 구현하는 방법은 크게 중앙집중방식과 분산방식으로 구분할 수 있다. 중앙집중방식은 특별한 하드웨어가 있어 중재를 필요로하는 모듈로부터 요청신호를 받아 우선순위에 따라 허락신호를 우선순위가 높은 모듈에 주는 방식이다. 이때 각 모듈은 중재동작에 최소한 두개의 신호선을 사용해야 되며 시간적으로 볼 때 요청 신호 구동과 허락신호 구동 및 허락신호 확인이라는 일련의 동작이 필요하다.

반면 분산방식은 동일한 구성을 갖는 중재기가 중재에 참가할 각 모듈에 있고 중재가 필요한 경우 중재 요청 신호를 구동함과 동시에 다른 모듈에서 구동하는 중재요청 신호들을 보고 독립적으로 우선순위가 가장 높은 모듈이 중재에서 이겼다는 것을 알 수 있게 하는 방식이다. 따라서 중재 요청 신호선만 있으면 되고 허락 신호선은 필요가 없다.

그림[2]는 I-Link 버스에서 사용하는 중앙 집중형 중재기의 개념적 블록도이다. 중재기는 PIF, DC, RC 로부터 6 개의 요청 신호(iREQ_[5:0])와 3 개의 busy 신호를 받아 중재 한다. 특히 PIF 의 경우 요청이 DC/RC/PIF 에 관련된 것인지를 구별하기 위해 dcrc 와 loopback 신호를 이용한다. loopback 신호가 0 일때 dcrc 가 의미를 갖는다.



[그림 2] 중재기의 개략적 블록도

I-Link 버스에서 사용하는 중재기는 중앙 집중형으로 우선순위를 이용한 중재기법을 사용한다. 중재기는 그림과 같은 구성을 갖는데 PIF 와 DC 그리고 RC 로부터 6 개의 요청 신호와 3 개의 busy 신호를 받아 중재한다. 특히 PIF 의 경우 요청이 DC/RC/PIF 에 관련된 것인지를 구별하기 위해 dcrc 와 loopback 신호를 이용한다. 즉 dcrc 가 '1'이면 해당 요청이 RC 로 갈 것임을 표시하는 것이고 loopback 신호가 '1'일 때는 PIF 가 PIF 로 요청을 전송할 것임을 나타내므로 loopback 신호가 '0'일 때 dcrc 가 의미를 갖는다.

4. VIS를 이용한 I-Link 버스의 검증

VIS를 이용하여 검증하기 위해서는 우선 formal model을 만들어야 한다. 그러기 위해서는 검증할 대상에 대한 문서를 정확하게 이해하고 그 문서대로 명확하게 명세 하는 것이 필요하다. 지금까지는 VIS를 이용할 때 검증할 대상에 대해 추상화 시킨 후 finite state machine을 그리고 그에 대해 VIS의 입

력 언어인 Verilog를 이용하여 명세하여 검증을 하였다. 그러나 이러한 방법을 이용할 경우 명세자의 오류가 있을 경우 정확한 검증을 하였다고 확신 할 수 없다는 문제가 발생할 수 있다. 따라서 이 논문에서는 실제 구현에 사용 될 Verilog 소스를 VIS의 입력으로 사용하고자 하였다. 이 방법을 이용할 경우 이전 방법을 이용할 때의 단점을 극복 할 수 있고 더 정확한 검증을 할 수 있을 것이라 생각 된다.

VIS를 이용한 검증의 첫 단계로 formal model을 구현하여야 하는데 실제 구현에 사용할 Verilog 소스를 이용하기로 결정하고 VL2MV로 컴파일 하는 과정에서 여러 가지 오류가 발생함을 발견하였다. 실제 구현에 사용할 소스는 Synopsis tool에서 이용하기 위한 입력 언어인데 반해 VIS를 이용하여 검증 할 때 사용할 소스는 VL2MV를 이용하여 컴파일하여 BLIF-MV 화일을 생성하여 VIS의 입력을 받아야 하기 때문에 실제 구현에 사용할 소스에 약간의 수정이 필요하였다. 앞에서 언급했듯이 verilog를 VL2MV의 입력으로 받을 때 문법상의 차이 또한 있기 때문에 원래 의미상의 차이가 전혀 없이 VL2MV 입력 언어로 바꾸는 작업이 중요하다고 하겠다. 이 작업 중에 어려운 점 중에 큰 부분은 VL2MV에서는 inout port와 tri-state port를 이용 할 수 없었다. 따라서 이 부분에 대한 변환이 필요 하였다. 또한 비결정성에 관한 문제 또한 발생하였다.

다음 단계는 vl2mv를 이용하여 컴파일하면 생기는 BLIF-MV 파일을 VIS의 입력으로 받아 들여 검증의 초기화를 해야 한다. 이 작업에서도 또한 여러 가지 문제가 발생하였는데 l-link bus 전체를 컴파일한 blif-mv 화일을 입력 받을 때 blif-mv 화일의 특징인 non-determinism과 table 생성시의 문제로 인하여 에러가 발생하였다. 이 문제는 아직 연구 중이며 해결해야 할 문제이다. 따라서 모듈별로 검증해 보기로 하였다. 우선 I-link 버스에서 중요한 부분인 중재기에 대한 검증부터 시작하였다. 검증할 특성은

1. 우선 순위에 따라 중재 하는가.
2. 바쁨 신호(busy signal)에 대한 처리를 올바로 하는가.

이다. 이러한 내용을 바탕으로 VIS에서 검증 할 때 이용하는 logic인 CTL을 이용하여 property를 명세하였다.

4.2 I-link 버스의 구현

I-Link 버스를 VIS로 검증하기 위해서는 우선 시스템을 검증 도구의 입력언어 형태로 구현 해야 한다. 이번 연구의 목적은 VIS의 입력언어가 Verilog이며 이 언어를 사용하여 I-link 버스가 구현되고 시뮬레이션 된 상태이기 때문에 그에 사용된 소스를 그대로 이용하여 검증하는 데 있다. 따라서 검증 할 대상인 I-link 버스에 관련된 문서를 바탕으로 I-link 버스의 동작과 프로토콜에 관한 이해를 먼저 한 후 소스 분석에 들어갔다. 소스를 분석하는 단계에서 구현에 사용된 Verilog와 VIS입력으로 사용될 Verilog의 여러 차이점이 발견하였다. 따라서 다음에 열거한 내용을 바탕으로 소스를 수정하고 또한 VIS를 이용하여 검증하기 위해 비결정성을 가질 수 있도록 하였으며 또한 wire 변수는 CTL문장의 입력으로 사용 할 수 없으므로 그에 대한 변환을 하였다.

4.2.1 VIS 를 위한 Verilog 특징

1) always block 내부에서는 reg 변수에 blocking assignment 만 허용된다. 그러므로 always block 내에 중재변수(즉 정의해 의해 wire 가 되는)를 쓰지 말고 non-blocking assignment(<=) 도 사용하지 말아야 한다.
2) 만약 변수가 서로 의존하여 할당되어야 한다면 변수들을 분리하여 always block 에 할당해야 한다. 그렇게 하지 않으면 behavior 가 실행 순서에 따르게 된다.
3) always block 에서 blocking assignment (=) 는 문장의 순서에 민감하다. always block 에서 non-blocking assignment (<=)를 허용하지 않기 때문에 요구되는 행동을 하게 하기 위해서는 순서대로 실행하는지를 분석해야만 한다.
4) assignment 의 block 을 갖는 것은 허용되지 않고 initial 문장을 위한 assignment 의 block 은 허용된다.
5) always block 에서 다음 clock 에서 reg 변수가 명시적으로 할당 되지 않는다면 이전 값을 지킨다.
6) wire 에 $ND assignment 를 사용하여 비결정성을 나타낸다.
7) VL2MV 는 초기상태를 명세하지 않은 Verilog 프로그램은 받아들이지 않는다. 사용자가 비결정적인 초기 상태를 원한다면 이것은 명시적으로 $ND 구조를 사용하는 것을 명세해야 한다.
8) for 문장은 VL2MV 에 의해 지원 된다. for loop 은 always block 내부에서만 사용될 수 있다. 디폴트로 VL2MV 는 그것을 프로세싱 하기 전에 verilog code 를 매크로 확장(unrolls) 한다. 이 때 unrolling 을 막기 위한 –u 옵션을 사용해야 한다.
9) wire 는 배열이 아니라 벡터이다. 그러나 reg 는 배열이 될 수 있다
10) VL2MV 는 –z 옵션이 사용될 때 만들어진 $ND 구조를 위해 나머지 버퍼를 둔다. 원래는 없다. 다시 말하면 VL2MV 는 원래 왼쪽 변수를 $ND 를 위한 비결정성 table 에 직접 연결한다. 그러나 –z 가 사용될 때는 $ND 의 올바른 사용법을 알아야 한다. assign <var> = $ND(…); assign 문장이 continuous assignment 이다. 생성된 비결정성 table 은 출력 변수로써 <var>를 사용한다. 만약 –z 옵션을 사용했다면 표현에서 $ND 정의를 사용 할 수 있다. assign a = $ND(0, 1) + b 나 assign a = (sel) ? $ND (0, 1) : b. 이런 경우에 중재하는 변수는 $ND 구조를 위해 생성된다. 따라서 디폴트 값을 사용하고 비결정성 변수를 명시적으로 명명하는 것이 좋다. 왜냐하면 그렇게 하는 것이 VIS 에 의사 입력이 될 것이고 사용자에 의해 주어진 이름을 가질 것이기 때문이다.

4.2.2 중재기 구현

구현에 이용될 소스와 문서를 분석한 후 VIS를 이용하여 검증 할 수 있는 형식으로 변환하였다. 우선 전체적인 시스템에 대한 구현을 완벽하게 변환하지 않았기 때문에 중재 프로토콜만을 검증하기 위하여 중재 프로토콜을 제외한 다른 부분은 모두 환경적인 요소로 처리를 하였다. 따라서 중재기의 입력으로 들어 오게 되는 중재 요청 신호, busy 신호, dcrc 신호, lopb신호는 환경적인 요소이기 때문에 비결정적으로 다음과 같이 발생하게 하였다.

```
assign dcrc = $ND(0,1);
assign lopb = $ND(0,1);
assign bsy[1] = $ND(0,1);
assign bsy[0] = $ND(0,1);
assign req[1] = $ND(0,1);
….
```

그리고 원래 사용된 소스는 CTL문장을 사용하여 모델 체킹을 하기 어렵기 때문에 다음과 같이 VIS에서 지원하는 기호적

변수(symbolic vqriable)를 이용하여 변환하였다. 이를 이용하여 좀 더 용이하게 검증을 할 수 있었다.

```
   Typedef enum {ILINK_ARB_IDLE,   ILINK_ARB_GRT5,
ILINK_ARB_GRT4, ILINK_ARB_GRT3, ILINK_ARB_GRT2,
ILINK_ARB_GRT1, ILINK_ARB_GRT0} state;
   typedef enum {PIF, RC, DC, NULL} grant;
   typedef enum {ON, OFF} loop;
   typedef enum {ON, OFF} rcdc;
   typedef enum {ON, OFF} request0;
   ….
```

또한 이러한 신호들은 wire를 통해 입력 되기 때문에 CTL문장에서 사용할 수 없으므로 그에 대한 FSM을 다음과 같이 구현하여 검증에 사용하였다.

```
always @ (posedge clk) begin
case (busy_RC)
            OFF : if (bsy[0] == 1)
                Begin
                busy_RC = ON;
                      end
                     else
                begin
                busy_RC = OFF;
                      end
 ….
```

다른 상세한 부분은 원래 소스를 그대로 이용하려 하였으며 문법적인 차이에 의한 변환에 의해서만 수정하였다. 따라서 이 전의 방법인 추상화 시켜 구현하여 검증한 것 보다는 실제 구현에 사용할 소스를 그대로 이용하였다는 점에서 검증의 정확성이 더 크다고 할 수 있다. 중재 프로토콜 이외에 sender, receiver등의 구현은 현재 진행 중이며 각각의 검증을 행한 후 전체적인 기능 검증 또한 필요할 것이다.

4.3 검증 결과

VIS 입력 형식으로 중재 프로토콜을 구현하였으므로 구현한 소스를 vl2mv로 컴파일하여 BLIF-MV화일로 변환한 후 이를 이용하여 시뮬레이션과 검증 작업을 한다. VIS에서는 테스트 벡터를 무작위적으로 발생하여 시뮬레이션 한 결과를 다음과 같이 보여주게 된다.

```
# Network: ILINK_ARB
# Simulation vectors have been randomly generated
  .inputs   bsy<0> bsy<1> bsy<2> dcrc lopb req<0> req<1> req<2> req<3>
req<4> req<5>
.latches arbit_state busy_DC busy_PIF busy_RC grt lopb_state rcdc_state
req_state0 req_state1 req_state2 req_state3 req_state4 req_state5
.outputs grt
.initial ILINK_ARB_IDLE OFF OFF OFF NULL OFF OFF OFF OFF OFF
OFF OFF OFF
  .start_vectors
  # bsy<0> bsy<1> bsy<2> dcrc lopb req<0> req<1> req<2> req<3> req<4>
req<5> ; arbit_state busy_DC busy_PIF busy_RC grt lopb_state rcdc_state
req_state0 req_state
1 req_state2 req_state3 req_state4 req_state5 ; grt
  0 1 0 0 1 0 0 0 1 0 1 ; ILINK_ARB_IDLE OFF OFF OFF NULL OFF OFF
OFF OFF OFF OFF OFF OFF ; NULL
1 1 1 0 1 1 0 1 1 1 0 ; ILINK_ARB_GRT5 ON   OFF OFF RC    ON   OFF
OFF OFF OFF ON OFF ON    ; RC
….
```

또한 검증하고자 하는 특성들을 CTL을 이용하여 명세 한 후 모델 체킹 방법을 이용하여 검증을 한다. 검증할 특성은

1. 우선 순위에 따라 중재 하는가.
2. 바쁨 신호(busy signal)에 대한 처리를 올바로 하는가.

이다.

다음 검증 결과 중 앞의 6개 특성은 1번 특성인 우선순위에 따라 중재를 하는가에 대한 명세이다. 예를 들어 첫번째 조건은 가장 우선 순위가 높은 req[3]에 신호가 들어 오면 우선 busy신호는 모두 무시했을 경우 req[3]의 의미가 RC에서 PIF로의 응답을 하기 위한 요청이므로 중재기의 중재를 받은 후 RC가 grant 신호를 받아야 한다는 의미이다. 두 번째 조건은 가장 우선 순위 높은 req[3]이 요청하지 않을 때 그 다음 우선 순위를 갖는 req[2]가 요청을 한다면 DC가 PIF로 응답을 하기 위한 요청이므로 DC가 grant 신호를 받아야 한다는 의미이다. 뒤의 4개의 property 또한 그러한 의미이고 모두 만족한다는 것을 알 수 있었다. 따라서 1번 특성인 우선 순위에 따라 중재한다는 것을 검증 할 수 있었다. 다음으로 2번 특성인 바쁨 신호(busy signal)에 대한 처리를 올바로 하는가에 관한 검증의 예를 보면 마지막 property는 req[5]와 req[1]이 동시에 요청 했을 때 만약 DC나 RC가 busy라면 req[5]가 우선순위가 높다 하더라도 DC나 RC가 신호를 처리 할 수 없기 때문에 req[1]에게 grant를 주게 되어 RC가 버스를 사용할 수 있도록 중재 된다. 다른 특성도 여러 가지 검증해본 결과 busy신호에 관련하여서도 이 중재 프로토콜이 제대로 작동함을 알 수 있었다.

```
vis> mc arbiter.ctl
# MC: formula passed --- AG((((((req_state3=ON * busy_PIF=OFF) *
busy_RC=OFF) * busy_DC=OFF) * ((arbit_state=ILINK_ARB_IDLE +
arbit_state=ILINK_ARB_GRT5) + arbit_state=ILINK_ARB_GRT1)) ->
AF(grt=RC)))
# MC: formula passed --- AG(((((((req_state3=OFF * req_state2=ON) *
busy_PIF=OFF) * busy_RC=OFF) * busy_DC=OFF) *
((arbit_state=ILINK_ARB_IDLE + arbit_state=ILINK_ARB_GRT4) +
arbit_state=ILINK_ARB_GRT0)) -> AF(grt=DC)))
# MC: formula passed --- AG(((((((((req_state3=OFF * req_state2=OFF) *
req_state5=ON) * busy_PIF=OFF) * busy_RC=OFF) * busy_DC=OFF) *
((arbit_state=ILINK_ARB_IDLE + arbit_state=ILINK_ARB_GRT3) +
arbit_state=ILINK_ARB_GRT2)) -> AF(grt=PIF)))
# MC: formula passed --- AG(((((((((req_state3=OFF * req_state2=OFF) *
req_state5=OFF) * req_state4=ON) * busy_PIF=OFF) * busy_RC=OFF) *
busy_DC=OFF) * ((arbit_state=ILINK_ARB_IDLE +
arbit_state=ILINK_ARB_GRT3) + arbit_state=ILINK_ARB_GRT2)) ->
AF(grt=PIF)))
# MC: formula passed --- AG((((((((((req_state3=OFF * req_state2=OFF)
* req_state5=OFF) * req_state4=OFF) * req_state1=ON) *
busy_PIF=OFF) * busy_RC=OFF) * busy_DC=OFF) *
((arbit_state=ILINK_ARB_IDLE + arbit_state=ILINK_ARB_GRT1) +
arbit_state=ILINK_ARB_GRT5)) -> AF(grt=RC)))
# MC: formula passed --- AG((((((((((req_state3=OFF * req_state2=OFF)
* req_state5=OFF) * req_state4=OFF) * req_state1=OFF) *
req_state0=ON) * busy_PIF=OFF) * busy_RC=OFF) * busy_DC=OFF) *
((arbit_state=ILINK_ARB_IDLE + arbit_state=ILINK_ARB_GRT4) +
arbit_state=ILINK_ARB_GRT0)) -> AF(grt=DC)))

# MC: formula passed --- AG(((((((req_state3=OFF * req_state2=ON) *
busy_PIF=OFF) * busy_RC=OFF) * busy_DC=OFF) *
arbit_state=ILINK_ARB_GRT4) -> AF(grt=DC)))
# MC: formula passed --- AG((((req_state3=ON * busy_PIF=OFF) *
arbit_state=ILINK_ARB_IDLE) -> AF(grt=RC)))
# MC: formula passed --- AG((((req_state3=ON * busy_PIF=OFF) *
arbit_state=ILINK_ARB_GRT5) -> AF(grt=RC)))
# MC: formula passed --- AG(((((req_state3=ON * busy_PIF=OFF) *
arbit_state=ILINK_ARB_GRT4) * !((req_state2=ON * busy_PIF=OFF))) -
> AF(grt=RC))) # MC: formula passed --- AG(((((req_state3=ON *
busy_PIF=OFF) * arbit_state=ILINK_ARB_GRT3) * !((req_state5=ON *
((rcdc_state=ON * busy_DC=OFF) + (rcdc_state=OFF * busy_RC=OFF))
```

```
+ (lopb_state=ON * busy_PIF=OFF))))) -> AF(grt=RC)))
# MC: formula passed --- AG(((((req_state3=ON * busy_PIF=OFF) *
arbit_state=ILINK_ARB_GRT2) * !((req_state4=ON * (((rcdc_state=ON *
busy_DC=OFF) + (rcdc_state=OFF * busy_RC=OFF)) + (lopb_state=ON *
busy_PIF=OFF))))) -> AF(grt=RC)))
# MC: formula passed --- AG(((((req_state3=ON * busy_PIF=OFF) *
arbit_state=ILINK_ARB_GRT1) * !((req_state1=ON * busy_PIF=OFF))) -
> AF(grt=RC)))
# MC: formula passed --- AG(((((req_state3=ON * busy_PIF=OFF) *
arbit_state=ILINK_ARB_GRT0) * !((req_state0=ON * busy_PIF=OFF))) -
> AF(grt=RC)))
# MC: formula passed --- AG((((((((((req_state4=ON * req_state0=ON) *
busy_DC=ON) * busy_RC=ON) * req_state3=OFF) * req_state1=OFF) *
req_state2=OFF) * req_state5=OFF) * busy_PIF=OFF) *
arbit_state=ILINK_ARB_GRT5) -> AF(grt=DC)))
# MC: formula passed --- AG(((((((req_state5=ON * req_state1=ON) *
(busy_DC=ON + busy_RC=ON)) * req_state2=OFF) * req_state0=OFF) *
arbit_state=ILINK_ARB_GRT4
```

다음은 invariant 조건으로 이 중재 프로토콜에서 절대 일어나면 안 되는 경우를 명세하여 검증한 결과이다. PIF, RC, DC가 동시에 버스에 대한 사용권을 가지면 안 된다는 의미의 명세이며 모두 만족함을 알 수 있었다.

```
Vis> check_invariant arbiter.invar
# INV: formula passed --- !((grt=PIF * grt=DC))
# INV: formula passed --- !((grt=DC * grt=RC))
# INV: formula passed --- !((grt=RC * grt=PIF))
# INV: formula passed --- !(((grt=RC * grt=PIF) * grt=D
```

이 검증을 통하여 I-Link 버스가 제대로 중재를 하여 정확하게 작동함을 알 수 있었고 실제 Verilog 를 이용한 구현이 정확하게 구현되었음을 확인 할 수 있었다.

5. 결론 및 향후 과제

정형 기법은 정형 논리와 수학에 기반을 둔 방법으로 자연어가 내포하는 애매모호함과 불확실성을 배제할 수 있으며, 시스템이 어떤 특성을 만족하는지 검증하기 때문에 최소한 검증된 특성에 대해서는 완전히 믿을 수 있게 한다. 또한 시스템이 구현되기 전에 정확히 동작하는지를 검증 할 수 있으므로 시스템 개발 비용과 개발 시간의 측면에서 매우 효과적이다. 본 논문은 모델 체킹 기법을 사용하는 VIS를 이용하여 ETRI에서 개발한 디렉토리 기반의 CC-NUMA시스템의 CCA(Cache Coherent Agent) 보드 내부 버스인 I-Link(Inside Link) 버스에서 중재역할을 하는 중재기를 검증함으로써 정형 기법의 적용 가능성을 제시하였다. 특히 이번 검증에서는 실제 구현에 사용한 소스를 그대로 입력 언어로 사용하여 검증하는 방법을 시도하였다. 물론 실제 구현에 이용하는 소스가 크기가 너무 크고 VIS입력으로 받기 위해서는 여러 가지 변환이 필요하므로 많은 제약점이 있었다. 따라서 현재까지는 중재기의 중재 프로토콜만을 검증한 상태이다. 그러나 실제 구현에 이용한 소스를 그대로 이용하여 검증하였으므로 추상화하여 모델링 하여 검증한 것보다 신뢰성이 높다고 할 수 있다. 이 방법을 더 연구하여 다른 모듈에까지 확장하여 I-link 버스의 전체 기능을 모두 검증한다면 좋은 결과를 얻을 수 있을 것이다. 향후 과제는 이번 검증을 바탕으로 다양한 분야에 모델 체킹 기법을 적용하는 것이다.

참고 문헌

[1] Edmund M. Clarke and jeannette M. Wing, "Formal Methods : State of the Art and Future Directions" ACM Computing Surveys, pages 626-643, December 1996.

[2] David L. Dill, John Rushby, Acceptance of Formal Methods : Lessons from Hardware Design, IEEE Computer, April 1996, Vol. 29, No. 4, pp. 16-30.

[3]] Hossein Saiedien, Jonathan P. Bowen, Ricky W. Butler, David L. Dill, et al., An Invitation to Formal Methods, IEEE computer, pages 16-30, April 1996.

[4] Kenneth L. McMillan, "SYMBOLIC MODEL CHECKING:, Kluwer Academic Publisher 1993. Tiziano Villa, Gitanjali Swamy, and Thomas shiple. VIS User's Manual.

[5] Samir Palnitkar, Verilog HDL, Prentice Hall.

[6] D. E. thomas P.R. Moorby. The verilog Hardware Description Language. Kluwer Academic Publishers, Nowell, Massachusetts, 1991.

[7] R.K. Brayton et al.HSIS: A BDD based system for formal verification. Proc. Of Design Automation Conference,1994.

[8] R. E. Bryant, Graph-Based Algorithms for Boolean Function Manipulation, IEEE Transaction Computers, vol. 35, no. 6, pp.677-691, Aug. 1986.

[9] Tiziano Villa, Gitanjali Swamy, and Thomas shiple. VIS User's Manual.

[10] S. -T. Cheng. "Compiling Verilog into automata" Tech. Rep. UCB/ERL M94/37, May 1994

[11] William Chan et al., "Model Checking large software Specification", IEEE Transactions of Software Engineering, Vol.24, NO. 7, July 1998.

[12] 컴퓨터 시스템 연구부. CCA 보드에서 사용하는 전송 형태와 패킷 구성. Technical Report TM-3100-1999-036, 한국전자통신연구원 컴퓨터소프트웨어기술연구소, 1999.

[13] 컴퓨터 시스템 연구부. CCA 보드를 위한 I-Link 버스. Technical Report TM-3100-1990-013, 한국전자통신연구원 컴퓨터소프트웨어기술연구소, 1999.

[14] 컴퓨터 시스템 연구부. I-Link 버스 인터페이스 구현. Technical Report TM-3100-1999-071, 한국전자통신연구원 컴퓨터소프트웨어기술연구소, 1999.