論文2002-39CI-3-1

SMV를 이용한 RACE 프로토콜의 정형 검증 및 테스팅

(Formal Verification and Testing of RACE Protocol Using SMV)

南 元 洪 *, 崔 振 榮 **, 韓 宇 宗 **

(Won-Hong Nam, Jin-Young Choi, and Woo-Jong Han)

요 약

본 논문은 심볼릭 모델 체커 SMV(Symbolic Model Verifier)를 이용하여, 한국전자통신연구원(Electronics and Communications Research Institute)에서 개발한 캐쉬 일관성 프로토콜인 RACE(Remote Access Cache coherency Enforcement) 프로토콜의 몇 가지 특성(property)들을 검증함으로써, RACE 프로토콜이 중요 요구사항(requirement)들을 만족함을 보인다. 본 검증에서는 RACE 프로토콜의 모델을 SMV 입력 언어로 명세하며, 검증할 특성들을 CTL(Computational Tree Logic)을 이용하여 나타낸다. 본 검증을 통해서 RACE 프로토콜은 4개의 노드로 구성된 시스템에서 비정상적인 state/input 조합이 발생하지 않으며, liveness와 safety를 만족한다는 것을 검증하였다. 또한, 프로토콜 개발자들이 예상하지 못한 명세서 상의 모호성(ambiguity) 및 기아현상(starvation)을 발견하였으며, 본 검증 사례를 통하여 모델 체킹 기법이 하드웨어 프로토콜 검증에 효과적으로 이용될 수 있다는 것을 제안한다. 그리고, 검증시에 구현된 모델을 이용하여 시뮬레이션 및 테스팅에 유용하게 사용될 수 있는 테스트 케이스를 자동적으로 생성할 수 있는 새로운 방법을 제안한다.

Abstract

In this paper, we present our experiences in using symbolic model checker(SMV) to analyze a number of properties of RACE cache coherence protocol designed by ETRI(Electronics and Communications Research Institute) and to verify that RACE protocol satisfies important requirements. To investigate this, we specified the model of the RACE protocol as the input language of SMV and specified properties as a formula in temporal logic CTL. We successfully used the symbolic model checker to analyze a number of properties of RACE protocol. We verified that abnormal state/input combinations was not occurred and every possible request of processors was executed correctly We verified that RACE protocol satisfies liveness, safety and the property that any abnormal state/input combination was never occurred. Besides, We found some ambiguities of the specification and a case of starvation that the protocol designers could not expect before. By this verification experience, we show advantages of model checking method. And, we propose a new method to generate automatically test cases which are used in simulation and testing.

Key Words : Cache coherence protocol, formal verification, model checking, Test case

* 正會員, 美國 펜실베니아 大學校 컴퓨터 情報科學科
(Department of Computer and Information Science, University of Pennsylvania*, U.S.A.)

** 正會員, 高麗大學校 컴퓨터學科
(Department of Computer Science and Engineering, Korea University)

*** 正會員, Bethel Info U.S.A.
(Bethel Info U.S.A.)

※ 본 연구는 한국전자통신연구원 위탁과제 98241, 99379의 지원으로 연구되었음.

接受日字:1999年8月28日, 수정완료일:2002年4月30日

Ⅰ. 서 론

오늘날 하드웨어 시스템이나 소프트웨어 시스템은 기능면, 규모면에서 점점 커지고 복잡해지고 있으며, 이러한 복잡성의 증가로 시스템에서의 에러 존재 가능성은 더욱 높아지고 있다. Ariane-5 로켓의 폭발, 인텔 펜티엄 프로세서의 FDIV 명령어 오류,[2] 덴버 공항 지하 화물 처리 시스템의 실패 등에서 알 수 있듯이, 이러한 에러들은 비용, 시간, 인력면에서 엄청난 대가를 치러야 했으며, 심지어 인간의 생명에도 위협을 주었다. 특히, 인텔 펜티엄 프로세서의 FDIV 오류는 시스템 디자인에 있어서 가장 심각한 오류이기 때문이 아니라, 시스템이 대형화됨에 따라 이러한 종류의 오류는 언제든지 발생할 수 있고, 그 손실 역시 매우 크다는 이유로 정형 검증(formal verification)의 중요성을 강조 시켰다.[1] 따라서 하드웨어, 소프트웨어 설계자들은 점차로 기존에 사용하고 있는 시뮬레이션과 테스팅 기법[3]을 보완할 방법으로 정형 검증에 대해 관심을 갖게 되었다.

캐쉬 일관성 프로토콜은 여러 개의 데이터 복사본 사이의 일관성(consistency)을 유지하기 위해서, 캐쉬, 메모리 등의 통신 개체들을 조정하는 일종의 규칙이라고 할 수 있다. 현재까지 많은 프로토콜들이 제안되고 구현되었지만, 정형적으로 검증된 사례는 극히 드물다고 하겠다. 그 주요 이유는 대부분의 프로토콜들이 비교적 간단한 스누핑(snooping) 프로토콜이기 때문이다. 이러한 프로토콜들의 완전성은 시뮬레이션이나 테스팅과 같은 간단한 방법을 통해서도 증명될 수 있다. 그러나, 고성능이며 확장성이 뛰어난 시스템들이 개발됨에 따라, 캐쉬 일관성 프로토콜의 복잡성은 증가하게 되었으며, 이러한 복잡성의 증가로 무작위 시뮬레이션 기법이나 수작업을 통한 캐쉬 일관성 프로토콜의 검증은 불가능하게 되었다. 몇 가지 연구[4,12]들에서 시뮬레이션은 실제적으로 상당히 신뢰성이 떨어지며, 이러한 이유로 복잡한 프로토콜을 검증할 수 있는 보다 효율적이며 신뢰할 만한 방법론의 필요성이 제기되었다.

정형 기법(formal methods)[15]은 수학과 논리학에 기반을 둔 방법으로 하드웨어 시스템이나 소프트웨어 시스템을 명세하거나 검증하는 방법론들이며, 이러한 정형 기법에는 크게 정형 명세(formal specification)와 정형 검증(formal verification)이 있다. 정형 명세는 정형 논리(formal logic) 또는 수리 논리(mathematical logic) 등을 이용하여 시스템이 동작할 환경, 시스템이 만족해야 할 요구사항, 요구사항을 수행할 시스템 설계 등을 기술하는 것이다. 정형 검증은 정형 논리 또는 수리 논리 등에서 제공하는 증명 방법을 이용, 정형 명세를 분석하여 시스템의 무모순성 및 완전성을 검증하거나, 설계(implementation)가 주어진 가정에서 요구사항을 만족하는지를 검증하는 기법이다.

정형 검증에는 크게 정리 증명(theorem proving)과 모델 체킹(model checking)이 있다. 정리 증명 방법에서는 시스템과 증명하고자 하는 특성(property)을 수학적 논리를 이용한 논리식(formula)으로 표현하고, 시스템의 공리(axiom)로부터 inference rule들을 이용하여 원하는 특성의 증명을 찾아내는 방법론이다. 이 방법은 매우 강력하고 유연한 방법이지만, 수많은 논리적 증명을 해야 하므로 매우 어렵고 전문가가 아니면 이용하기가 어렵다. 반면, 모델 체킹은 비록 적용분야는 유한 시스템(finite system)으로 제한되지만, 직접 증명할 필요가 없이 검증 도구를 이용하여 손쉽고 빠르게 검증할 수 있으며 완전히 자동화 될 수 있다는 장점이 있다.

모델 체킹은 상태 탐색(state exploration)을 기반으로 하는 정형 검증 기법으로써, 상태 전이 시스템(state transition system)과 특성이 주어지면, 모델 체킹 알고리즘은 주어진 시스템이 검증하고자 하는 특성을 만족하는지를 알아보기 위해서 전체 상태 공간(state space)을 검사한다. 특히, 심볼릭 모델 체킹(symbolic model checking)[10]은 OBDDs(Ordered Binary Decision Diagrams)[11]을 이용함으로써, 엄청난 수(약 10120 states 이상[14,19])의 상태를 가지는 시스템도 증명할 수 있게 되었다.

본 논문은 심볼릭 모델 체킹 검증 도구의 일종인 SMV(Symbolic Model Verifier)를 이용해서 한국전자통신연구원(Electronics and Telecommunications Research Institute)에서 개발한 디렉토리 기반의 캐쉬 일관성 프로토콜인 RACE(Remote Access Cache coherency Enforcement) 프로토콜을 검증한 사례를 제시함으로써, 하드웨어 프로토콜 검증에 모델 체킹 기법이 효과적으로 적용될 수 있음을 보인다. 또한, 모델 체커를 이용해서 시뮬레이션과 테스팅을 위한 새로운 테스트 케이스 생성 방법을 제안한다. 본 논문의 구성은 다음과 같다. 2장에서는 본 연구와 관련된 연구들에 대해서 알아본다. 그리고, 3장에서는 RACE 프로토콜을,

4장에서는 CTL 모델 체킹과 SMV에 대한 설명을 본 논문과 관련해서 간략하게 한다. 5장에서는 RACE 프로토콜의 명세에 대해서 설명하며, 6장에서는 RACE 프로토콜의 검증, 7장에서는 테스트 케이스 생성 방법에 대해서 설명한다. 끝으로 8장은 결론에 대해 설명한다.

Ⅱ. 관련 연구

1. 정형 검증

그 동안 많이 사용되어져 왔던 시뮬레이션과 테스팅은 개념적으로 간단하다는 장점은 있지만, 검사한 입력 값에 대해서만 시스템의 완전성을 보장하며, 대형 시스템의 경우 검사해야할 입력 값이 테스팅이 불가능할 정도로 많다는 단점 때문에 불완전성 문제를 내포하고 있다. 이와는 달리 정형 검증(formal verification)은 시뮬레이션이나 테스팅으로는 쉽게 찾아내기 힘든 불일치성, 애매모호함, 불완전성을 검증함으로써 보다 완벽한 시스템을 구축할 수 있게 한다. 특히, 자동적으로 검증할 수 있는 검증 도구의 개발로 인해 학계에서뿐만 아니라 산업계에서도 최근 들어 상당한 관심을 받아오고 있다.

CMU(Carnegie Mellon University)의 K. L. McMillan에 의해 개발된 SMV(Symbolic Model Verifier)[10]는 동기적인 시스템에서부터 비동기적인 시스템에 이르기까지 SMV 입력언어를 이용하여 시스템을 명세하고, 검증하려는 특성을 CTL(Computational Tree Logic)[9]로 표현하며, 심볼릭 모델 체킹 알고리즘을 사용하여 시스템이 검증하려고 하는 특성을 만족하는지를 검사한다. CTL 논리식은 safety, liveness, fairness, deadlock등과 같은 시간과 관련된 특성들을 쉽게 표현할 수 있는 장점이 있다. 1992년에 CMU의 E. Clarke은 SMV를 이용해서 IEEE Futurebus+ Standard 프로토콜을 검증함으로써 프로토콜 설계시 발견하지 못했던 에러들과 잠재적인 에러들을 발견하였다.[16]

Murφ[17]는 Stanford 하드웨어 검증 그룹의 David L. Dill에 의해 개발된 정형 검증 도구이다. Dill은 1992년 Murφ를 이용해서 SCI(Scalable Coherent Interface, IEEE Std. 1596-1992)프로토콜을 검증했다.[18] Dill은 검증을 위해 먼저 SCI 프로토콜을 Murφ 입력 언어로 표현한 후 캐쉬 일관성에 대한 명세를 작성하였다. 전체 상태의 수가 매우 커지므로 시스템의 아주 작은 부분만 정의하였지만, 변수 초기화에서부터 미묘한 논리적 에러에까지 다양한 프로토콜의 에러를 발견하였다.

VIS(Verification Interacting with Synthesis)[6]는 검증, 시뮬레이션, 하드웨어 시스템 합성(synthesis)등을 위한 도구이다. Texas-97 Verification Benchmarks에서의 VIS의 사용은 정형 검증에 대해 많은 가능성과 문제점을 제시해 주었다. Cache Coherence Protocol, PCI local Bus, PI-Bus Protocol, MESI Cache Coherence Protocol, MPEG System Decoder등을 검증하였다.

2. 테스트 케이스 생성 방법론

이 번 절에서는 기존의 3가지 테스트 케이스 생성 방법론들[22]에 대해서 설명한다. 이 방법론들은 프로토콜의 전체 명세가 밀리 머신(Mealy machine)으로 모델링되어 있는 것을 가정한다.

(1) T-Method

T-method[23]는 나머지 2개의 방법들에 비해 상대적으로 간단하다. 이 방법론은 시스템이 minimal, strongly connected, completely specified된 밀리 머신 모델이다라고 가정한다. 테스트 케이스는 machine의 모든 전이(transition)를 적어도 한번씩은 순회할 때까지 무작위(random) 입력을 fault-free machine에 적용함으로써 얻을 수 있다. 그러나, 생성된 케이스는 loop을 만드는 불필요한 입력을 포함하기도 한다. 이러한 불필요한 입력은 reduction 과정을 통해서 제거할 수 있다. T-method에 의해 생성된 테스트 케이스는 단지 전이의 존재만을 검사할 뿐 전이의 tail state는 검사하지 못한다.

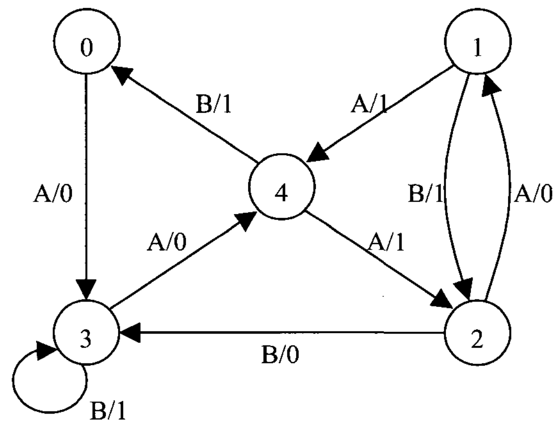


그림 1. Machine M에 대한 전이도
Fig. 1. The transition diagram of machine M.

표 1. Machine M에 대한 전이표
Table 1. The transition table of machine M.

	output		next-state	
input state	A	B	A	B
0	0	λ	3	0
1	1	1	4	2
2	0	0	1	3
3	0	1	4	3
4	1	1	2	0

예 : [그림 1]의 machine M을 위한 transition-tour 케이스는 아래와 같다.(아래 줄은 해당 상태의 시퀀스이다.)

B A B A B A A A A A A A A B B
0 0 3 3 4 0 3 4 2 1 4 2 1 2

(2) U-Method

U-method[24]는 *minimal, strongly connected, completely specified*된 밀리 머신 모델로 가정한다. 이 방법은 각 상태의 UIO(Unique Input/Output) 시퀀스[22]를 생성하는 것을 필요로 한다. 각 상태에 대한 UIO 시퀀스는 다른 상태에 의해서는 보여지지 않는 I/O behavior이다.

β-시퀀스[22]는 각 전이의 테스트 시퀀스를 연결함으로써 얻어 진다. Machine M의 임의의 전이 (s_i, s_j)에 대해 테스트 시퀀스를 다음과 같은 방법으로 구할 수 있다.

1) M을 초기 상태로 리셋하기 위해서 리셋 입력 r을 적용한다.

2) 상태 0에서 상태 s_i로의 가장 짧은 path SP(s_i)를 구한다.

3) M이 s_j 상태로 전이하도록 입력 심벌을 적용한다.

4) 상태 s_j의 UIO를 적용한다.

표 2. M에 대한 UIO 시퀀스
Table 2. The UIO sequence of M.

state	UIO
0	B/λ
1	A/1 A/1
2	B/0
3	B/1 B/1
4	A/1 A/0

예 : [표 2]는 [그림 1]의 M의 상태들의 UIO 시퀀스의 집합을 나타낸다. 모든 상태들은 UIO의 입력 스트링을 적용함으로써 얻어지는 결과 스트링에 의해 각각 구별된다. 즉, 입력 스트링이 AA이고 결과 스트링이 11이면, 입력 스트링을 적용하기 전에 상태 1에 있었다는 것을 알 수 있다.

그림 1의 M에 대한 β-시퀀스는 다음과 같다.

β-시퀀스 :

r A B B
r B B
r A A A A A A A
r A A A A B B
r A A A A A A
r A A A B B B
r A A A A
r A B B B
r A A A B
r A A B B

위의 서브시퀀스로부터 최적화된 테스트 케이스는 다음과 같다.

rAAAAAAArAAAABBrAAABBBrAABBrABBBrBB

(3) D-Method

D-Method[21]는 *minimal, strongly connected, completely specified*된 밀리 머신 모델로 가정한다. 또한, DS(Distinguishing Sequence)를 가져야 한다. 각 상태에서 어떤 입력 스트링 x가 주어졌을 때, 각각의 결과 스트링이 모두 다르다면, 이 입력 스트링 x를 distinguishing sequence라고 한다. D-method의 주요 개념은 machine M의 DS를 구하는 것이다.

D-method에서 β-시퀀스를 만드는 방법은 U-method에서 각 상태에 대한 UIO 시퀀스 대신 DS로 대체하는 것 이외에는 U-method와 같다.

예 : 표 1에서 보면 BB가 그림 1의 machine M의 shortest DS라는 것을 쉽게 알 수 있다. 표 3은 이러한 DS를 M의 각 상태에 적용해서 얻은 결과 스트링을 보여 준다. 즉, 이 DS를 적용했을 때, 결과 스트링이 10이라면 DS를 적용하기 전에 상태가 1이었다는 것을 알 수 있다.

표 3. M에 DS에 대한 결과 스트링
Table 3. The result string of M's DS.

state	M/s<DS>
0	λ λ
1	1 0
2	0 1
3	1 1
4	1 λ

그림 1의 machine M에 D-method를 적용해서 얻은 β-시퀀스는 아래와 같다.

β-시퀀스 :

r A B B
r B B B
r A A A A A B B
r A A A A B B B
r A A A A B B
r A A A B B B
r A A B B
r A B B B
r A A A B B
r A A B B B

위의 서브시퀀스로부터 최적화된 테스트 케이스는 다음과 같다.

rAAAAABBrAAAABBBrAAABBBrAABBBrABBBrBBB

Ⅲ. RACE 프로토콜

1. PDLSystem

CC-NUMA의 일종인 PDLSystem(Physically-Distributed but Logically-Shared memory system)은 그림 2와 같이 노드간 네크워크(inter-node network)에 연결되어 있는 여러 개의 노드들로 구성되어 있다. 각각의 노드들은 캐쉬를 가지는 프로세서들과 공유 메모리의 일부분인 메모리 모듈, I/O, 그리고 외부 접근 캐쉬(RAC, Remote Access Cache)와 디렉토리로 구성된 CCA(Cache Coherent Agent) 보드로 구성된다.[7]

노드 안의 프로세서 캐쉬들은 스누피(snoopy) 캐쉬 일관성 프로토콜에 의해 일관성을 유지한다. 프로세서는 프로세서 캐쉬에 없는 데이터 접근 시, 해당 캐쉬 라인을 가져오기 위해서 프로세서 버스에 요청을 보낸다. 만약, 그 주소가 노드 안의 로컬 메모리에 해당한다면 로컬 메모리가 해당 데이터를 제공한다. 그렇지 않은 경우는 외부 접근 캐쉬가 그 요청을 처리하게 된다. 즉, 외부 접근 캐쉬가 그 데이터를 유효한 상태로 가지고 있다면 바로 데이터를 제공하게 되고, 그렇지 않으면 노드간 네트워크로 연결된 외부 노드(remote node)의 메모리로부터 데이터를 가져오게 된다. 각 노드들은 디렉토리를 이용하여 외부 노드들의 접근에 관한 공유 정보들을 관리한다.

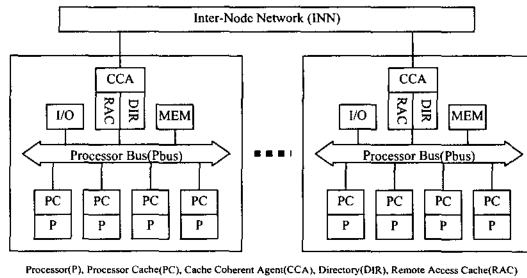


그림 2. PDLSystem
Fig. 2. PDLSystem.

2. 메모리 계층

(1) 프로세서 캐쉬(Processor cache)

프로세싱 노드 안의 프로세서 캐쉬들의 일관성은 MESI[20] 프로토콜에 의해 유지된다고 가정한다. 각각의 프로세서 캐쉬는 write-back 캐쉬이다.

(2) 외부 접근 캐쉬(Remote Access Cache)

프로세서 관점에서 외부 접근 캐쉬는 외부 데이터(remote data)를 프로세서에게 제공해 주는 메모리와 같다. 외부 주소(remote address)에 관해서는 같은 노드 안에 있는 외부 접근 캐쉬와 프로세서 캐쉬 사이에 space inclusion[13]이 유지된다. 각각의 캐쉬 블록이 가질 수 있는 상태는 다음과 같다.

- I(INVALID, RACI) : 이 블록은 유효하지 않은 상태이다. 이러한 블록에 대한 프로세서 버스의 접근은 miss되며, 외부 접근 캐쉬가 네트워크를 통해 해당 블록에 대한 요청을 하게 된다.
- S(SHARED, RAC_S) : 이 블록은 읽기 전용 상태이며, 홈 메모리가 같은 값을 가지고 있다. 0, 1, 또는 그 이상의 외부 접근 캐쉬가 읽기 전용 상태로 있을

수 있다.

- M(MODIFIED, RAC_M) : 이 블록은 쓰기 가능 상태이며, 외부 접근 캐쉬가 홈 메모리보다 최근의 데이터 값을 가지고 있다.
- L(LOCKED, RAC_L) : 이 블록은 interlock된 값을 가지며, 다른 모든 외부 접근 캐쉬들은 INVALIDED 상태이다. Locked read와 locked write를 제외한 모든 프로세서 버스 접근은 허용되지 않는다. 그리고, locked write만이 이 블록의 상태를 MODIFIED (RAC_M)로 변화시킬 수 있다.

위의 상태 이외에도 pending 상태들이 존재한다. 이러한 pending 상태들은 외부 접근 캐쉬의 상태가 전이 중이라는 것을 나타내며, 외부 접근 캐쉬의 상태가 atomic하게 변경되지 않는다는 것을 의미한다.

(3) 디렉토리(Directory)

디렉토리는 각 블록들에 해당하는 presence 비트들과 state 비트들로 구성된다. presence 비트는 각 블록의 공유 노드들에 대한 정보를 가지고 있다. 디렉토리는 외부 접근 캐쉬들과의 공유 상태를 나타내기 위해서 다음과 같은 상태를 가질 수 있다.

- U(UNCACHED, DIR_U) : 메모리 블록의 값이 유효하며, 어떠한 외부 접근 캐쉬도 복사본을 가지고 있지 않다.
- S(SHARED, DIR_S) : 메모리 블록의 값이 유효하며, 0, 1, 또는 그 이상의 외부 접근 캐쉬가 읽기 전용 복사본을 가질 수 있다.
- M(MODIFIED, DIR_M) : 메모리 블록의 값이 유효하지 않으며, 다른 노드의 외부 접근 캐쉬가 쓰기 가능 권한을 가지고 있다.

위의 상태 이외에도 pending 상태들이 존재한다. 이러한 pending 상태들은 디렉토리의 상태가 전이 중이라는 것을 나타내며, 디렉토리의 상태 역시 외부 접근 캐쉬처럼 atomic하게 변경되지 않는다는 것을 의미한다.

3. RACE 프로토콜

이번 절에서는 한국전자통신연구원(Electronics and Telecommunications Research Institute)에서 개발한 RACE(Remote Access Cache coherency Enforcement) 프로토콜[7]에 대해서 간략히 소개한다.

(3) 가정

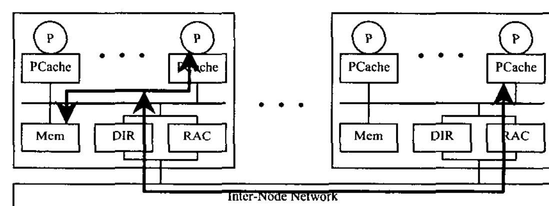
본 논문에서 검증할 RACE 프로토콜은 다음과 같은 가정을 따른다.

- 외부 접근 캐쉬에서 일어나는 모든 외부 접근(remote access) miss는 우선 홈 노드로 전해진다.
- Reply-forwarding이 사용된다.
- 노드간 네트워크(inter-node network)를 통한 데이터 전송 프로토콜은 한 노드에서 다른 노드로 보내는 메시지들의 순서를 보장한다.
- Ghost-sharing이 허용된다.

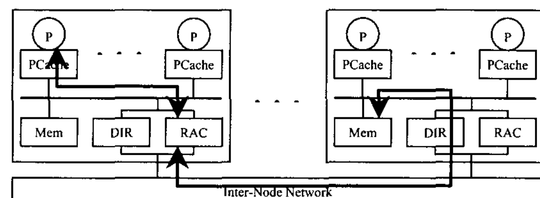
(2) RACE 프로토콜 개요

프로세서가 캐쉬 miss를 발생시키면, 해당 요청에 대한 홈 노드가 로컬인지 리모트인지를 구별하기 위해서, 해당 캐쉬 라인에 대한 메모리 블록의 주소를 확인한다. 만약 local인 경우, 그림 3(a)에서처럼 로컬 메모리가 해당 데이터를 프로세서 캐쉬에 제공할 것이며, 리모트인 경우는 그림 3(b)에서처럼 외부 접근 캐쉬(Remote Access Cache)가 해당 데이터를 프로세서 캐쉬에 제공할 것이다.

요청된 캐쉬 라인이 로컬 메모리에 해당하는 경우, 즉 로컬 요청이라도 데이터 일관성을 위해서 디렉토리를 참조해야 한다. 왜냐하면, 외부 접근 캐쉬가 복사본을 가지고 있을 수 있기 때문이다. 외부 요청(remote request)을 처리하기 위해서 외부 접근 캐쉬는 적절한 상태로 데이터를 가지고 있어야 한다. 그렇지 않으면, 요청 노드의 외부 접근 캐쉬는 노드간 요청(inter-node request)을 홈 노드로 발생시킨다.



(a) Local access



(b) Remote access

그림 3. Local and remote accesses
Fig. 3. Local and remote accesses.

그림 4와 그림 5는 RACE 프로토콜의 중요 부분인 외부 접근 캐쉬와 디렉토리의 상태 전이도(state transition diagram)이다.

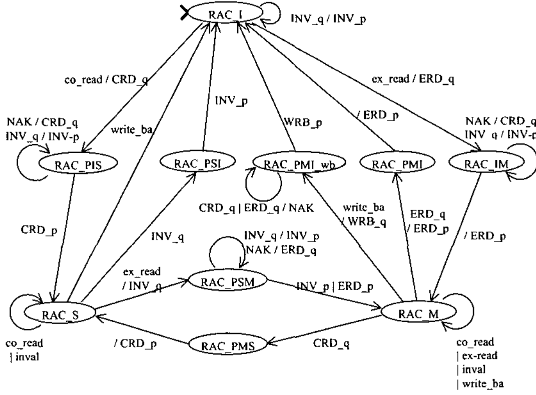


그림 4. 외부 접근 캐쉬의 상태 전이도

Fig. 4. The state transition diagram of RAC.

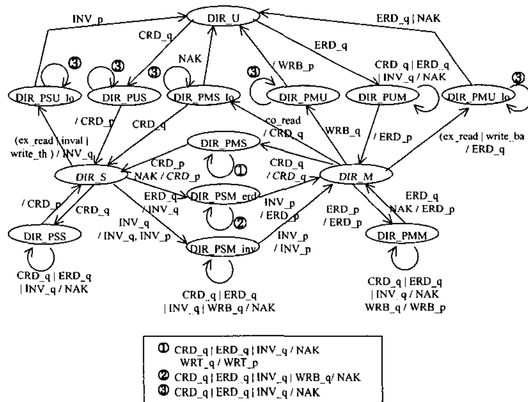


그림 5. 디렉토리의 상태 전이도

Fig. 5. The state transition diagram of DIR.

Ⅳ. CTL(Computational Tree Logic) 모델 체킹

1. Temporal Logic CTL(Computation Tree Logic)

이번 절에서는 Clarke와 Emerson에 의해서 제안된 모델 시제 논리(model tense logic)의 일부분인 모델 이론, 즉 CTL(Computation Tree Logic)[9]에 대해서 알아본다. CTL의 연산자들은 주어진 유한 모델(finite model)의 임의의 한 상태(state)가 주어진 논리식(formula)을 만족하는지를 효율적으로 알아 볼 수 있는 fixed point 특성을 갖는다.

CTL에서 시제 연산자는 A또는 E 다음의 F, G, U, X의 쌍으로 이루어진다. 또한, 과거에 대한 연산자는 제공되지 않으며, 시제 연산자는 명제 연산자(propositional connective)와 바로 연결되지 못한다. 보다 정확히 나타내면 다음과 같다.

- 모든 atomic proposition은 CTL formula이다.
- f 와 g가 CTL formula이면, 다음의 formula들도 CTL formula이다.

$$f, (f \wedge g), AXf, EXf, A(fUg), E(fUg)$$

나머지 연산자들은 다음의 규칙에 의해 위의 규칙으로부터 유도될 수 있다.

$$\begin{aligned} f \vee g &= \neg(\neg f \wedge \neg g) \\ AFg &= A(true \mathrm{U} g) \\ EFg &= E(true \mathrm{U} g) \\ AGf &= E(true \mathrm{U} \neg f) \\ EGf &= A(true \mathrm{U} \neg f) \end{aligned}$$

모든 연산자들은 A 또는 E로 시작하기 때문에 논리식의 참, 거짓은 특정 branch에 해당하는 것이 아니라 특정 상태에 해당하게 된다.

CTL의 semantics는 Kripke structure와 관련하여 정의한다. Kripke structure M은 $\langle S, R, L \rangle$로 정의한다. 여기서 S는 상태들의 집합이며, $R \subseteq S \times S$은 total인 전이 관계(transition relation), $L: S \rightarrow 2^{AP}$는 labeling 함수이다. 그리고, AP는 모든 atomic proposition의 집합이다. M에서 path는 상태들의 무한한 sequence $(s_0, s_1, s_2, \cdots)$이다. Sequence는 모든 $i \geq 0$에 대해서, $(s_i, s_{i+1}) \in R$이 성립한다.

$s \models p$ iff $p \in L(s)$, where p is an atomic proposition

$s \models \neg f$ iff $s \not\models f$

$s \models f \wedge g$ iff $s \models f$ and $s \models g$

$s_0 \models AX\ f$ iff for all paths $(s_0, s_1, s_2, \cdots)$, $s_1 \models f$

$s_0 \models EX\ f$ iff for some path $(s_0, s_1, s_2, \cdots)$, $s_1 \models f$

$s_0 \models A(f\ \mathrm{U} g)$ iff for all paths $(s_0, s_1, s_2, \cdots)$, for some i, $s_i \models g$ and for all $j < i$, $s_j \models f$

$s_0 \models E(f\ \mathrm{U} g)$ iff for some path $(s_0, s_1, s_2, \cdots)$,

for some i, $s_i \models g$ and for all $j < i$, $s_j \models f$

2. SMV

SMV 시스템[10]은 유한 상태 시스템(finite state system)이 CTL로 표현된 요구 명세를 만족하는지를 자동적으로 검증하는 정형 검증 도구이다. SMV의 입력 언어는 유한 상태 시스템을 명세하기 위해 만들어졌으며, 이 입력 언어를 이용해서 시스템을 동기적인 밀리 머신(Mealy machine)이나 비동기적인 네트워크로 손쉽게 명세할 수 있다. SMV 언어가 유한 상태 시스템을 위해 만들어진 것이기 때문에 언어가 제공하는 자료 구조도 유한한 형태(Boolean, scalar, fixed array 등)만을 제공한다. CTL은 safety, liveness, fairness, dead lock freedom등을 포함한 다양한 종류의 시간적 특성(temporal property)들을 간단한 문법을 이용하여 표현하는 것이 가능하다. SMV 시스템은 SMV 입력 언어로 나타내어진 모델이 CTL로 표현된 요구 명세를 만족하는지의 여부를 효율적으로 검사하기 위해서, OBDD(Ordered Binary Decision Diagram)[11]를 기반으로 하는 심볼릭 모델 체킹 알고리즘을 사용한다.

계속해서 본 논문과 관련된 SMV 언어의 특징들을 설명하겠다. SMV에서는 1이 *true*를 의미하고, 0이 false를 의미하며, 논리 연산자 *and, or, not*은 &, |, !으로 표현된다.

SMV 프로그램은 유한 상태 시스템의 명세와 CTL formula의 리스트로 구성된다. 상태 전이 시스템(state transition system)은 상태 공간(state space), 전이 관계(transition relation), 초기 상태(initial state) 집합의 세 가지로 구성된다. 이 중 상태 공간은 *VAR*라는 예약어 뒤에 나오는 상태 변수 선언으로써 결정된다. 예를 들면 다음과 같다.

```
VAR
   b : boolean;
   from : 0..5;
   switch : {on, off};
```

위의 예는 부울 변수 b와 0과 5사이의 정수를 갖는 *from*, 집합 {*on, off*}의 원소를 값으로 가지는 변수 *switch*를 선언하는 것이다.

전이 관계와 초기 상태는 simultaneous assignment들로 명세될 수 있다. Assignment들은 *ASSIGN*이라는 예약어 뒤에 나온다. 어떤 변수 *var*에 대해서, init(var)는 초기 상태에서 *var*의 값을 나타낸다. 예를 들면, 아래의 SMV 코드는 변수 b의 초기 값을 0으로 설정한다.

```
ASSIGN
   init(b) := 0;
   next(b) := !b;
```

그리고, 전이 관계를 정의하기 위해서, next(var)는 다음 상태에서의 *var*의 값을 나타낸다. 위의 코드에서 변수 b의 다음 상태 값이 현재 값의 부정이라는 것을 나타낸다. 즉, b의 값이 0과 1을 계속해서 토글하는 것을 나타낸다.

다음 상태 값을 정의하는 일반적인 방법은 case expression을 사용하는 것이다.

```
ASSIGN
   next(x) := case
         x < 7 : x + 1;
         1 : 0;
   esac;
```

위의 예는 *x*의 현재 값이 7보다 작으면 다음 상태의 *x*의 값을 1만큼 증가시키고, 위의 조건을 만족하지 않으면, 다음 상태의 *x*의 값을 0으로 하는 것을 보여 준다. 즉, *x*는 modulo-8 counter가 된다. (1은 *true*를 의미하므로, 두 번째 경우인 1 : 0;은 default case가 된다.)

V. SMV를 이용한 RACE 프로토콜의 모델링

이 번 장에서는 RACE 프로토콜을 SMV로 검증하기에 앞서 모델링을 위한 몇 가지 가정들과 추상화 모델, 그리고 구현 등에 관해 설명한다.

1. 가정

본 논문의 검증은 다음과 같은 가정을 따른다.

- 노드의 수는 4개로 제한한다.
- 각 노드의 프로세서의 수는 1로 한다. 노드 안의 캐

쉬 일관성 문제는 MESI 프로토콜에 의해 유지된다고 가정하기 때문에 노드간의 캐쉬 일관성 문제만을 대상으로 한다. 그러므로, 노드 안의 프로세서 수는 1로 한정할 수 있다.

- I/O는 캐쉬 일관성 문제와 무관하므로 배제한다.
- 검증의 범위를 하나의 캐쉬 라인에 대한 검증만으로 제한한다. 서로 다른 캐쉬 라인은 서로의 상태 전이(state transition)에 영향을 미치지 않기 때문이다.
- 디렉토리나 외부 접근 캐쉬가 자신의 노드에 있는 메모리 또는 프로세서 캐쉬와 주고받는 요청, 응답은 각각 pending 상태에 한 스텝 머무르는 동안 처리된다고 가정한다.

2. 추상화 모델

그림 6은 5.1절에서 설명한 가정을 기반으로 해서, 그림 2의 PDLSystem을 추상화시킨 것이다. 하나의 캐쉬 라인만을 검증 대상으로 하기 때문에 그림에서 왼쪽의 노드를 home node라고 하면, 나머지 3개의 노드는 remote node 1, remote node 2, remote node 3이 된다. 그러므로 home node에서는 RAC를 나머지 3개의 remote node들에서는 DIR을 생략할 수 있다. 그리고, 이들을 연결하는 inter-node network을 구현한다.

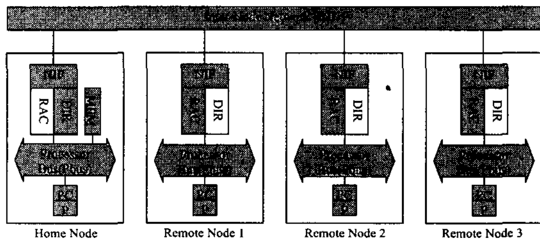


그림 6. PDLSystem의 추상화 모델
Fig. 6. Abstract Model of PDLSystem.

3. 명세

이 번 절에서는 RACE 프로토콜에서 가장 중요한 부분인 외부 접근 캐쉬와 디렉토리에 대한 명세에 대해 간략히 설명한다.

(1) 외부 접근 캐쉬(RAC)

다음에서는 외부 접근 캐쉬의 리모트 액세스에 관한 상태 전이 중 한 경우인 외부 접근 캐쉬가 I(invalid) 상태에서 coherent read가 발생한 경우, 외부 접근 캐쉬의 상태 전이를 SMV 입력 언어로 어떻게 바꾸는지를 보여 준다.

그림 7은 리모트 노드의 외부 접근 캐쉬가 I 상태에서 coherent read가 발생한 경우이다. 이 경우, 홈 노드의 디렉토리는 U(uncached)나 S(shared) 상태일 수 있다. 모든 오퍼레이션이 끝난 후, 외부 접근 캐쉬는 S(shared) 상태가 디렉토리는 S(shared) 상태가 된다. 위의 경우를 SMV 입력 언어로 나타내면 그림 8과 같다.

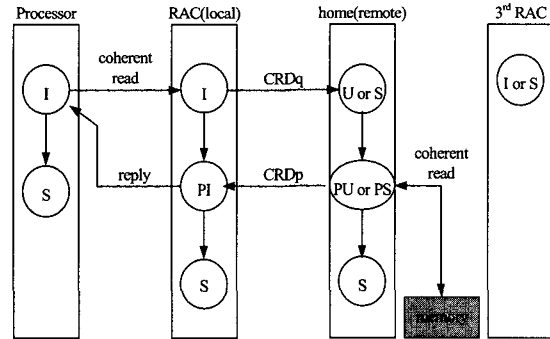


그림 7. Coherent remote read to RAC_I and DIR_U or DIR_S
Fig. 7. Coherent remote read to RAC_I and DIR_U or DIR_S.

```
init(state) := RAC_I;
next(state) := case
   state = RAC_I & bus = co_read : RAC_PIS;
   state = RAC_PIS & input = CRD_p : RAC_S;
   ...
esac;

init(output) := none;
next(output) := case
   state = RAC_I & bus = co_read : CRD_q;
   ...
esac;
```

그림 8. Coherent read의 SMV 명세
Fig. 8. The Specification of Coherent read.

(2) 디렉토리(Directory)

다음은 디렉토리의 리모트 액세스에 관한 상태 전이 중 한 경우인 외부 접근 캐쉬가 I(invalid) 상태에서 exclusive read가 발생한 경우, 디렉토리의 상태 전이를 SMV 입력 언어로 어떻게 바꾸는지를 보여 준다.

그림 9는 리모트 노드의 외부 접근 캐쉬는 I(invalid)

상태이고, 디렉토리는 S(shared) 상태에서 exclusive read가 발생한 경우이다. 모든 오퍼레이션이 끝난 후, 외부 접근 캐쉬는 M(modified) 상태가 디렉토리도 M(modified) 상태가 된다. 위의 경우를 SMV 입력 언어로 나타내면 그림 10과 같다.

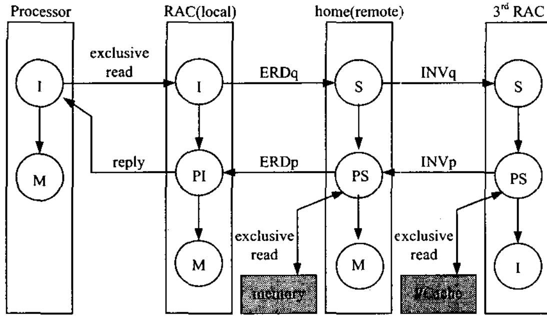


그림 9. Exclusive remote read to RAC_I and DIR_S
Fig. 9. Exclusive remote read to RAC_I and DIR_S.

```
init(state) := DIR_U;
next(state) := case
   state = DIR_S & input = ERD_q :
                                  DIR_PSM_erd;
   state = DIR_PSM_erd & wait = 0 : DIR_M;
   ...
esac;

init(buf_to_1) := none;
next(buf_to_1) := case
   state = DIR_S & input = ERD_q &
       is_1_shared & !(source = 1)   : INV_q;
   ...
esac;
```

그림 10. Exclusive read의 SMV 명세
Fig. 10. The Specification of Exclusive read.

VI. 검증

1. 검증 특성

(1). Diagnostic correctness

디렉토리와 외부 접근 캐쉬에서 비정상적인 state/input 조합이 발생하는지를 검사한다. 그림 11의 첫 번째 논리식은 홈 노드의 디렉토리에서 명세 이외의 state/input 조합이 발생하는지를 검사하는 것이고, 두 번째 논리식은 리모트 노드의 외부 접근 캐쉬에서 명세 이외의 state/input 조합이 발생하는지를 검사하는 것이다.

```
AG !(home.state = error)
AG !(rac1.state = error)
```

그림 11. 비정상적인 state/input 조합 발생 검사
Fig. 11. Verification of absence of abnormal state/input combination.

(2) Liveness

리모트 노드나 홈 노드의 프로세서에서 발생할 수 있는 모든 요청이 deadlock이나 livelock, starvation이 발생하지 않고 처리되는지를 검사한다. 그림 12의 논리식은 리모트 액세스에서의 liveness를 검사하는 것이며, 그림 13은 로컬 액세스에서의 liveness를 검사하는 것이다.

그림 12의 첫 번째 논리식은 외부 접근 캐쉬(RAC)가 I 상태에 있고, 버스에서 coherent read 요청이 들어 왔

```
AG (rac1.state = RAC_I & rac1.bus = co_read  -> AF rac1.state = RAC_S)
AG (rac1.state = RAC_I & rac1.bus = ex_read -> AF rac1.state = RAC_M)
AG (rac1.state = RAC_S & rac1.bus = ex_read -> AF rac1.state = RAC_M)
AG (rac1.state = RAC_S & rac1.bus = inval -> AF rac1.state = RAC_M)
AG (rac1.state = RAC_S & rac1.bus = write_ba -> AF rac1.state = RAC_I)
AG (rac1.state = RAC_M & rac1.bus = write_ba -> AF rac1.state = RAC_I)
AG (rac1.state = RAC_I & rac1.bus = write_th -> AF rac1.state = RAC_M)
AG (rac1.state = RAC_S & rac1.bus = write_th -> AF rac1.state = RAC_M)
```

그림 12. Remote access의 liveness property
Fig. 12. The liveness property of remote access.

```
AG (home.state = DIR_S & home.local_bus = ex_read -> AF home.state = DIR_U)
AG (home.state = DIR_S & home.local_bus = inval -> AF home.state = DIR_U)
AG (home.state = DIR_S & home.local_bus = write_th ->AF home.state = DIR_U)
AG (home.state = DIR_M & home.local_bus = co_read -> AF home.state = DIR_S)
AG (home.state = DIR_M & home.local_bus = ex_read -> AF home.state = DIR_U)
AG (home.state = DIR_M & home.local_bus = write_th -> AF home.state = DIR_U)
```

그림 13. Local access의 liveness property
Fig. 13. The liveness property of local access.

```
AG !(rac1.state = RAC_M & rac2.state = RAC_M)
AG !(rac1.state = RAC_M & (rac2.state = RAC_S | rac3.state = RAC_S))
```

그림 14. Liveness property
Fig. 14. Liveness property.

을 때, 어떠한 경우라도 외부 접근 캐쉬가 S 상태가 된다는 것을 검증하는 것이다. 두 번째 논리식은 외부 접근 캐쉬가 I 상태에 있고, 버스에서 exclusive read 요청이 들어 왔을 때, 어떠한 경우라도 외부 접근 캐쉬가 M 상태가 된다는 것을 검증하는 것이며, 나머지 논리식들도 비슷하게 해석할 수 있을 것이다.

다음으로 그림 13의 첫 번째 논리식은 디렉토리가 S 상태에서 local exclusive read 요청이 들어 왔을 때, 어떠한 경우라도 디렉토리가 U 상태가 된다는 것을 검증하는 것이다. 또한, 그림 13의 네 번째 논리식은 디렉토리가 M 상태에서 local coherent read 요청이 들어 왔을 때, 어떠한 경우라도 디렉토리가 U 상태가 된다는 것을 검증하는 것이며, 나머지 논리식들도 비슷하게 해석할 수 있다.

(3) Safety

그림 14의 특성은 safety에 관한 특성들이다. 그 중 첫 번째 것은 어떤 경우라도 둘 이상의 리모트 노드가 동시에 owning 노드가 될 수 없다라는 특성이고, 두 번째 특성은 하나의 리모트 노드가 owning 노드이면, 다른 리모트 노드들은 공유 노드(shared node)가 될 수 없다라는 특성을 검증하기 위한 논리식이다.

2. RACE 프로토콜 분석

본 논문의 검증은 RACE Protocol V0.3 Draft[8]를 바탕으로 실시한 검증으로, 검증 결과 시뮬레이션이나 테스팅 기법으로는 발견하기 힘든 명세서 상의 일부 모호한 점과 기아현상(starvation)을 발견하였다. 즉, 자연어를 이용한 명세가 명세자와 구현자 사이에 모호성(ambiguity)을 발생시킬 수 있는 가능성을 보여 준다. 설계자와 구현자가 동일하거나, 서로 간에 충분한 이해의 과정이 있는 경우는 큰 문제를 발생하지 않겠지만, 그렇지 않는 경우는 중대한 문제를 유발할 수도 있다.

(1) 명세의 모호성

(1) 그림 15는 홈 노드에서 coherent remote read 요청을 처리하고 있는 중간에 owning 노드로부터 writeback 요청이 들어 온 경우이다. 명세에서는 exclusive remote read 요청인 경우만이 명세되어 있는데, 그림 15에서 지적한 coherent remote read의 경우 역시 같은 처리가 필요하다는 것이다. 즉, owning 노드(1st RAC)가 CRDq를 받았을 때 PM 상태에 있으므로, 홈 노드의 디렉토리에 NACK를 보낸다. 홈 노드의 디렉토리는 owning 노드로부터 NACK를 받기 전에 WRBq를 받으므로, 메모리를 갱신하고 owning 노드에게 WRBp을 전달하고, 동시에 요청 노드에 CRDp를 보냄으로써 해결할 수 있다. 이와 같은 경우는 명세자가 exclusive remote read 요청 중간에 writeback 요청이 발생한 경우만을 명세하고, coherent remote read 요청이 발생하는 경우는 똑같은 방법으로 처리된다는 사실을 모호하게 기술함으로써 발생하는 경우라고 하겠다.

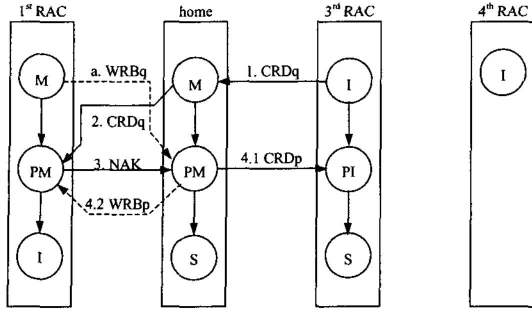


그림 15. Coherent remote read와 Writeback 요청이 동시에 처리되는 경우
Fig. 15. Coherent remote read and Writeback.

(2) 그림 16은 홈 노드에서 coherent local access를 처리하고 있는 중간에 owning 노드로부터 writeback remote request가 동시에 들어 온 경우이다. 명세서에서는 특별하게 명세가 되어 있지 않기 때문에 모든 처리가 끝난 후에 홈 노드의 상태를 어떻게 해야 하는지에 대해서 모호성(ambiguity)이 발생하게 된다. 즉, 최종 홈 노드의 상태를 S 상태로 하고, 나중에 ghost sharing을 통해서 1st RAC에 대한 shared 상태를 없애 줄 수 있고, 다른 방법으로는 새로운 transition을 첨가함으로써 홈 노드의 상태를 I 상태로 변경을 할 수도 있다. 두 가지 모두 올바르게 처리되겠지만, 명세서에서 기술하지 않는다면 구현자 임의대로 구현할 소지가 발생할 수 있으며, 유지 보수 과정에서도 문제가 발생한다.

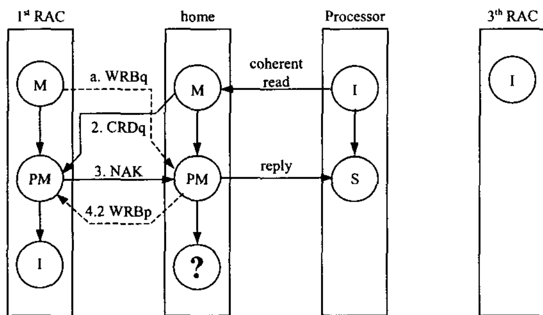


그림 16. Coherent local read와 Writeback 요청이 동시에 처리되는 경우
Fig. 16. Coherent local read and Writeback.

참고로 위의 두 가지 경우 모두 RAVE Protocol V1.1 Draft[7]에는 모두 명세된 것을 확인하였다.

(2) Starvation

그림 17은 pending 상태에 hit된 요청이 임의의 시간 후 재시도될 때마다, 다른 노드의 remote request나 홈 노드의 local request가 발생하는 경우를 보여 준다. 요청한 노드는 계속해서 NACK을 받고 임의의 시간 후에 재시도를 하지만, 계속해서 다른 노드들이 먼저 요청하여 응답을 받게 되고, 결국 기아현상(starvation)이 발생하게 된다. 다음과 같은 상황은 하나의 캐쉬 라인에 대해 계속되는 쓰기 접근이 발생할 경우에 일어날 수 있을 것으로 예상된다. 이러한 경우는 발생할 확률은 극히 적지만, 프로토콜 설계자들이 꼭 미리 고려해야 하는 경우라 하겠다.

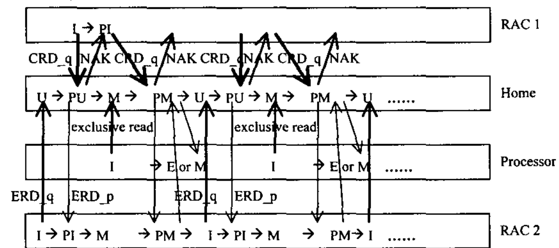


그림 17. Starvation이 발생하는 경우의 counterexample
Fig. 17. A starvation counterexample.

3. 검증 결과

그림 18은 SMV를 이용해서 RACE 프로토콜을 검증한 결과이다. 8.2.1과 8.2.2에서 제시한 부분을 제외한 검증한 결과로써 6.1절에서 설명한 특성들이 모두 참의 결과 값을 갖는다는 것을 알 수 있다. 만약, 아래의 검증 특성들 중 거짓(false) 값을 갖는 특성이 있었다면, counterexample을 출력해 줄 것이다.

따라서, RACE 프로토콜은 4개의 노드이하를 갖는 PDLSystem에서 diagnostic correctness, liveness, safety 특성을 만족한다는 것을 검증하였다.

Ⅶ. 테스팅

이 번 장에서는 본 논문에서 제안하는 모델 체킹 기법을 이용한 테스트 케이스 생성 방법에 대해서 설명한다. 2.2절에서 설명한 기존의 상태 기반 테스트 방법[21~24]들은 전체 상태 전이도(state transition diagram)를 완성해야 한다는 단점과 상대적으로 적은 수의 상태를 가지는 시스템에만 적용될 수 있다[5]는 단점이 있다. 반면에 본 논문에서 제안하는 방법은 모델 체킹 단계에서 만들었던 모델을 그대로 사용하며 심볼릭 모델

```
-- specification AG (!home.state = error) is true
-- specification AG (!rac1.state = error) is true
-- specification AG (rac1.state = RAC_I & rac1.bus = co_r... is true
-- specification AG (rac1.state = RAC_I & rac1.bus = ex_r... is true
-- specification AG (rac1.state = RAC_S & rac1.bus = ex_r... is true
-- specification AG (rac1.state = RAC_S & rac1.bus = inva... is true
-- specification AG (rac1.state = RAC_S & rac1.bus = writ... is true
-- specification AG (rac1.state = RAC_M & rac1.bus = writ... is true
-- specification AG (rac1.state = RAC_I & rac1.bus = writ... is true
-- specification AG (rac1.state = RAC_S & rac1.bus = writ... is true
-- specification AG (home.state = DIR_S & home.local_bus ... is true
-- specification AG (home.state = DIR_S & home.local_bus ... is true
-- specification AG (home.state = DIR_S & home.local_bus ... is true
-- specification AG (home.state = DIR_M & home.local_bus ... is true
-- specification AG (home.state = DIR_M & home.local_bus ... is true
-- specification AG (home.state = DIR_M & home.local_bus ... is true
-- specification AG !(rac1.state = RAC_M & rac2.state = R... is true
-- specification AG !(rac1.state = RAC_M & (rac2.state = ... is true

resources used:
user time: 3208.6 s, system time: 1.35 s
BDD nodes allocated: 2328008
Bytes allocated: 63316016
BDD nodes representing transition relation: 2058341 + 2917
```

그림 18. SMV를 이용한 검증 결과
Fig. 18. The SMV verification result.

체킹 도구를 사용함으로써, 위에서 설명한 기존의 상태 기반 테스트 케이스 생성 방법론들이 가지는 단점들을 보완할 수 있다.

1. 테스트 케이스 생성 방법

본 논문에서 제안하는 테스트 케이스 생성 방법을 개략적으로 나타내면 그림 19와 같다. 모델 체커인 SMV의 입력으로는 모델(finite state machine)과 test purpose를 나타내는 CTL 식이 주어진다. SMV는 test purpose을 만족시키는 counterexample을 생성하며, 이 counterexample에서 외부 입력 변수들의 값만을 추출하면 이것이 test purpose을 만족시키는 테스트 케이스가 된다.

시스템 모델은 검증 과정에 사용된 코드를 그대로 사용함으로써 유한 상태 시스템(finite state system)을 다시 구축하지 않아도 되며, 검증 과정에서도 유한 상태 시스템을 부분적으로 구현함으로 전체 유한 상태 시스템을 구현하는 복잡함을 배제할 수 있다.

다음으로는 test purpose를 나타낸다. 이러한 test purpose는 원하는 test trace를 나타내며, 구현상의 에러를 찾을 가능성이 높은 것으로 선택해야 한다. 그러므로, 어떤 에러가 발생할 것인가 또는 어떤 에러를 찾는 것이 더욱 중요할 것인가를 추측하는 것이 매우 중요한 일이다. Test purpose는 하나의 특성(property)이나 여러 개의 특성으로 간단히 나타낼 수 있으며,

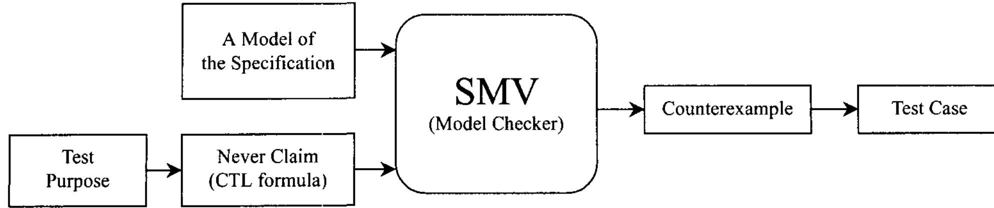


그림 19. 테스트 케이스 생성 방법
Fig. 19. The test case generation method.

SMV의 입력으로 제공되기 위해서는 temporal logic (CTL)으로 표현될 수 있어야 한다. 이렇게 나타낸 temporal logic에 negation을 취한 것을 nerver claim이라 한다.

SMV는 모델과 nerver claim을 입력으로 받아서 false 값과 함께 test purpose에 맞는 counterexample을 제공해 줄 것이다. 이러한 counterexample은 최종 상태까지의 각각의 스텝에서 입력 변수들에 대한 값으로 구성된다. 이 중 외부 변수들의 값만을 추출하면 테스트 케이스를 구성할 수 있다.

2. 적용 예제

이 번 절에서는 7.1절에서 설명한 방법에 대한 간단한 예제를 보인다. 예제는 RACE 프로토콜로써, 각각의 리모트 노드들의 외부 접근 캐쉬(remote access cache)

File Prop View Goto History Abstraction Help
Browser | Properties | Results | Cone | Using | Groups

Property	Result
test_gen	false

Source | Trace | Log
File Edit Run View

	0	2	0	3	0	1	0	0	0
	none	none	none	none	none	none	none	CRD_p	INV_q
	none	none	none	CRD_p	none	none	none	none	INV_q
	none	none	none	none	none	CRD_p	none	none	INV_q
	none	CRD_q	none	CRD_q	none	CRD_q	none	none	none
	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	1	1
	0	0	0	0	0	0	0	0	0
	0	0	0	1	1	1	1	1	1
	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	1	1	1	1
	none	none	none	none	none	none	none	ex_read	-
	0	2	0	3	0	1	0	0	0
	DIR_U	DIR_U	DIR_PUS	DIR_S	DIR_PSS	DIR_S	DIR_PSS	DIR_S	DIR_PSU
	none	none	none	none	none	none	none	CRD_p	INV_q
	none	none	none	CRD_p	none	none	none	none	INV_q
	none	none	none	none	none	CRD_p	none	none	INV_q
	none	CRD_q	none	CRD_q	none	CRD_q	none	none	none
	none	none	none	none	co_read	none	none	none	-
	none	none	none	none	none	none	none	CRD_p	INV_q
	none	none	none	none	none	CRD_q	none	none	none
	RAC_I	RAC_I	RAC_I	RAC_I	RAC_I	RAC_PIS	RAC_PIS	RAC_PIS	RAC_S
	co_read	none	none	none	none	none	none	none	-
	none	none	none	CRD_p	none	none	none	none	INV_q
	none	CRD_q	none	none	none	none	none	none	none
	RAC_I	RAC_PIS	RAC_PIS	RAC_PIS	RAC_S	RAC_S	RAC_S	RAC_S	RAC_S
	none	none	co_read	none	none	none	none	none	-
	0	0	0	0	0	0	0	0	0
	none	none	none	none	none	CRD_p	none	none	INV_q
	none	none	none	CRD_q	none	none	none	none	none
	RAC_I	RAC_I	RAC_I	RAC_PIS	RAC_PIS	RAC_PIS	RAC_S	RAC_S	RAC_S
	2	0	3	0	1	0	0	0	-

Property: test_gen I-search:

그림 20. 테스트 케이스 생성 예제
Fig. 20. A example of test case generation.

들의 상태가 동시 SHARED가 되기 위한 테스트 케이스 생성이다.

7.1절에서 설명한 바대로 RACE 프로토콜의 테스트 케이스 생성을 위해서 또 다시 시스템을 구현하는 것이 아니라 검증 시에 사용한 모델을 그대로 사용한다. 그리고, 위에서 설명한 test purpose를 CTL 식으로 나타내면 다음과 같다.

RAC1.state = RAC_S & RAC2.state = RAC_S & RAC3.state = RAC_S

위의 식을 never claim으로 바꾸면 아래와 같다.

AG !(RAC1.state = RAC_S & RAC2.state = RAC_S & RAC3.state = RAC_S)

위와 같은 방법으로 nerver claim을 나타낸 후 SMV를 이용해서 counterexample을 구하면, 그 counterexample에서 바로 테스트 케이스를 추출할 수 있다. 그림 20은 위의 CTL 식과 검증 과정에서 구현한 RACE 프로토콜의 코드를 더해서 SMV로 검증한 결과이다.

표 4. 생성된 테스트 케이스
Table 4. The generated test cast.

home_local_bus	rac1.bus	rac2.bus	rac3.bus
none	none	co_read	none
none	none	none	none
none	none	none	co_read
none	none	none	none
none	co_read	none	none
none	none	none	none
none	none	none	none
ex_read	none	none	none

Counterexample은 3개의 외부 접근 캐쉬가 9번째 스텝에서 모두 SHARED 상태가 됨을 보여 준다. 그리고, 그렇게 되기 위해서는 1번째 스텝에서 8번째 스텝까지 어떤 상황이 발생해야 하는지를 보여 준다. 이 중에서 외부 입력 변수인 home_local_bus, rac1.bus, rac2.bus, rac3.bus에 대한 값만을 추출하면 바로 테스트 케이스가 된다. 즉, 3개의 외부 접근 캐쉬가 동시에 SHARED 상태가 되게 하는 테스트 케이스는 표 4와 같다. 이 번 절에서 설명한 예제 외에도 테스트하고자 하는 상황을 CTL 식으로만 타나낼 수 있다면, 그 CTL 문장을 이용하여 위와 같은 방법으로 테스트 케이스를 생성할 수 있다.

Ⅷ. 결 론

정형 검증은 정형 논리와 수학에 기반을 둔 방법론으로써 자연어가 내포하는 애매모호함과 불확실성을 배제할 수 있으며, 시스템이 어떤 특성을 만족하는지를 검증하기 때문에 최소한 검증된 특성에 대해서는 완전히 믿을 수 있다. 특히, OBDD를 기반으로 한 심볼릭 모델 체킹은 대형 시스템에서도 잘 적용될 수 있으며, 다양한 특성들을 검증할 수 있다.

본 논문에서는 한국전자통신연구원에서 개발한 PDLSystem을 위한 디렉토리 기반 캐쉬 일관성 프로토콜인 RACE 프로토콜과 CCA 보드을 정형 검증 도구인 SMV를 이용하여 검증하였다. 본 논문의 검증을 통해, RACE 프로토콜은 4개 이하의 노드를 갖는 PDLSystem은 diagnostic correctness, liveness, safety 특성들을 만족하며, CCA 보드는 I-Link 버스와 각 모듈들이 큐 오버플로우 미발생, 중재 우선 순위, 송신 우선 순위, 중재 요청 신호의 관리, liveness등의 특성들을 만족한다는 것을 검증하였다.

본 검증을 통해서, 프로토콜 개발자들이 예상하지 못한 명세서 상의 모호성(ambiguity) 및 기아현상(starvation)을 발견하였으며, 본 검증 사례를 통하여 하드웨어 검증에 모델 체킹 기법이 효과적으로 이용될 수 있다는 것을 제안하였다.

또한, 검증 시에 구현된 모델을 그대로 이용하여 시뮬레이션이나 테스팅에 유용하게 사용될 수 있는 테스트 케이스를 자동적으로 생성할 수 있는 새로운 방법을 제안하였다.

참 고 문 헌

[1] David L. Dill and John Rushby. Acceptance of Formal Methods: Lessons from Hardware Design. IEEE Computer, Vol. 29, No. 4, pp. 16

~30, April 1996.

[2] Vaughan Pratt. Anatomy of the Pentium Bug. In TAPSOFT 95: Theory and Practice of Software Development, Vol. 915 of Lecture Notes in computer Science, pp. 97~107, May 1995.

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

[4] K. L. McMillan and J. C. Schwalbe. Formal Verification of the Gigamax Cache Consistency Protocol. Proceedings of the ISSM International Conference on Parallel and Distributed Computing, Oct. 1991.

[5] David Lee and Mihalis Yannalkakis. Principles and Methods of Testing Finite State Machines - A Survey. Proceedings of the IEEE, vol. 84, pp. 1090~1123, August 1996.

[6] R. K. Brayton et al. VIS: A System for Verification and Synthesis. In T. Henzinger and R. Alur, editors, 8th Conference on Computer Aided Verification, pp. 428~432. Springer-Verlag, 1996. LNCS 1102.

[7] Computer System Department of ETRI-CSTL. RACE Protocol: Remote Access Cache coherency Enforcement Protocol, V1.1 Draft. TM-3100-1999-012, August 1999.

[8] Computer System Department of ETRI-CSTL. RACE Protocol: Remote Access Cache coherency Enforcement Protocol, V0.3 Draft. January 1999.

[9] E. M. Clarke and E. A. Emerson. Synthesis of Synchronization skeletons for Branching Time Temporal Logic. Logic of Programs: Workshop, Vol. 131 of Lecture Notes in Computer Science, May 1981.

[10] Kenneth L. MaMillan. Symbolic Model Checking. Kluwer Academic Publishers, 1993.

[11] R. E. Bryant. Graph-based Algorithms for Boolean Function Manipulation. IEEE Transactions on Computers, Vol. 35, No. 6, pp. 677~691, August 1986.

[12] F. Pong, A. Nowatzyk, G. Aybay and M. Dubois. Verifying Distributed Directory-based Cache Coherence Protocols: S3.mp, a Case Study. Proceedings of the First International EURO-PAR Conference, pp. 287~300, August 1995.

[13] Jean-Loup Baer and Wen-Hann Wnag. Architectural Choices for Multilevel Cache Hierarchies. Proceedings of the 1st International Conference on Parallel Processing, pp. 258-261, August 1987.

[14] J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic Model Checking with Partitioned Transition Relations, Proceedings of the 1991 International Conference on VLSI, pp. 49~58, August 1991.

[15] E. M. Clarke and J. M. Wing. Formal methods: State of the Art and Future Directions. ACM Computing Surveys, 28(4), pp. 626~643, December 1996.

[16] E. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. Long, K. McMillan, and L. Ness. Verification of the Futurebus+ cache coherence protocol. Formal Methods in System Design, 6(2), pp. 217~232, March 1995.

[17] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol Verification as a Hardware Design Aid. IEEE International Conference on Computer Design: VLSI in Computers and Processors, pp. 522~525, 1992.

[18] Ulrich Stern and David L. Dill. Automatic Verification of the SCI Cache Coherence Protocol. Correct Hardware Design and Verification Methods: IFIP WG10.5 Advanced Research Working Conference Proceedings, 1995.

[19] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill. Symbolic Model Checking for Sequential Circuit Verification. IEEE Transactions on Computer-Aided Design of Integrated Circuits 13(4):401~424.

[20] Paul Sweazey and Alan Jay Smith. A Class of Compatible Cache Consistency Protocols and

Their Support by the IEEE Futurebus. Proceedings of the 13th Annual International Symposium on Computer Architecture, pp. 414 ~423, 1986.

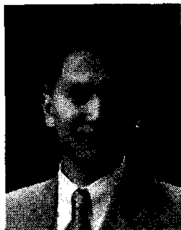
[21] G. Gonenc. A Method for the Design of Fault Detection Experiments. IEEE Transactions on Computing, vol. C-19, pp. 551~558, June 1970.

[22] Deepinder P. Sidhu and Ting-Kau Leung. Formal Methods for Protocol Testing: A Detailed Study. IEEE Transactions on Software Engineering 15(4), pp. 413~426 April 1989.

[23] S. Naito and M. Tsunoyama. Fault Detection for Sequential Machines by Transition Tours. Proceedings of IEEE Fault Tolerant Computing Conference, pp. 238~243, 1981.

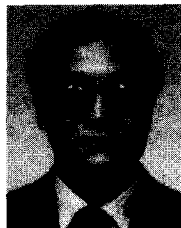
[24] K. Sabnani and A. Dahbura. A Protocol Test Generation Procedure. Computer Networks ISDN System, vol. 15, pp. 285~297, 1988.

저 자 소 개



南 元 洪(學生會員)

1998년 고려대학교 컴퓨터학과 학사. 2001년 고려대학교 대학원 컴퓨터학과 석사. 2001년~현재 University of Pennsylvania, Dept. of Computer & Information Science 박사 과정. <주관심분야 : 모델 체킹(Model Checking), 정형 검증(Formal Verification), 정형 명세(Formal Specification), 정형 기법(Formal Methods), 실시간 시스템(Realtime System), 하이브리드 시스템(Hybrid System) 등>



崔 振 榮(正會員)

1982년 서울대학교 컴퓨터공학과 학사. 1986년 Drexel University Dept. of Mathematics and Computer Science 석사. 1993년 University of Pennsylvania, Dept. of Computer & Information Science 박사. 1993년~1996년 University of Pennsylvania, Research Associate. 1996년~1999년 고려대학교 컴퓨터학과 조교수. 1999년~현재 고려대학교 컴퓨터학과 부교수. <주관심분야 : 계산 이론(Computing Theory), 전산 논리(Computational Logic), 실시간 컴퓨팅(Real-Time Computing), 정형 기법(Formal Methods), 프로그래밍 언어(Programming Languages), 프로세스 알제브라(Process Algebras), 소프트웨어 공학(Software Engineering), 프로토콜 공학(Protocol Engineering) 등>

韓 宇 宗(正會員)

1981년 고려대학교 전자공학과 학사. 1984년 고려대학교 전자공학과 석사. 1995년 고려대학교 전자공학과 박사. 1985년 한국전자통신연구소 연구원. 1986년~1988년 미국 AIT사 파견 연구원. 1989년 전자계산기 분야 기술사. 1997년~1998년 한국전자통신연구원 프로세서연구실장. 1998년~2001년 한국전자통신연구원 병렬시스템연구팀장, 책임연구원. 현재 Bethel Info, U.S.A. 재직 중. <주관심분야 : 컴퓨터구조(Computer Architecture), 마이크로프로세서 구조(Microprocessor Architecture), 병렬처리 구조(Parallel Processing Architecture), 계층메모리 구조(Hierarchical Memory Architecture) 등>