중소기업정보기술융합학회논문지 제1권 제1호 pp.75~83, 2011

UML State Machine Diagram을 이용한 소프트웨어 시스템의 데드락 탐지

민현석*
(주) 다한 테크

Deadlock Detection of Software System Using UML State Machine Diagram

Hyun-Seok Min
Dahan Tech Inc.

요 약 Unified Modeling Language (UML)는 산업계에서 소프트웨어 설계 표준 언어로서 인정되고 있으며 특히 UML State Machine Diagram은 클래스의 동적인 행위(behavior)를 묘사하는데 많이 사용되고 있다. 이 논문은 UML State Machine Diagram을를 이용하여 시스템의 데드락 (deadlock)을 찾는 방법에 대해서 논한다. 보통 State Machine Diagram는 개별의 클래스의 행위를 나타내는 데 사용되므로 시스템 범위의 행위를 알고 싶으면 시스템에 있는 클래스들 중 관심 있는 클래스들의 State Machine Diagram을 합하여 시스템의 행위를 나타낼 수 있는 State Machine Diagram이 필요하여진다. 일반적으로 이러한 시스템 수준의 State Machine Diagram은 매우 복잡하고 실제로는 타당하지 않은 State나 Transition들을 포함하게 된다. 실제 시스템의 행위를 나타내기 위해서 synchronization과 externalization을 적용하여 State Machine Diagram을 유효한 수준으로 줄이는 것이 필요하다. 이렇게 만들어진 State Machine Diagram은 시스템의 행위를 나타내는데 사용될 수 있으며 통상의 모델 체킹 방법이 적용될 수 있다. 이 논문은 데드락 탐지를 하는 방법을 간단한 예제를 통해서 보여준다. 모든 과정은 툴에서 자동으로 지원되며 필요한 알고리즘도 같이 설명된다.

키워드 : UML, State Machine Diagram, 소프트웨어 시스템, 데드락

Abstract Unified Modeling Language (UML) is widely accepted in industry and particularly UML State Machine Diagram is popular for describing the dynamic behavior of classes. This paper discusses deadlock detection of System using UML State Machine Diagram. Since a State Machine Diagram is used for indivisual class' behavior, all the State Machine Diagrams of the classes in the system are combined to make a big system-wide State Machine Diagram to describe system behavior. Generally this system-wide State Machine Diagram is very complex and contains invalid state and transitions. To make it a usable and valid State Machine Diagram, synchronization and externalization are applied. The reduced State Machine Diagram can be used for describing system behavior thus conventional model-checking technique can be applied. This paper shows how deadlock detection of system can be applied with simple examples. All the procedures can be automatically done in the tool.

Key words : UML, State Machinr Diagram, Software system, Deadlock

*교신저자 (hsmin@dahan.co,kr)
접수일자(2011년 10월 7일), 심사완료일(2011년 11월 4일)

1. 서론

모델 체킹은 오랜 시간 동안 존재되어 왔지만 아직 산업계에서의 사용은 많은 편이 아니다. 여기에는 많은 이유가 있는데 가장 큰 이유 중 하나는 개발자들이 모델 체킹을 위한 새로운 툴과 언어를 배우는데 있어서 부담을 느끼기 때문이다. 다른 큰 이유는 이러한 일이 보통 분석 단계에서 행하여지기 때문에 실제 구현은 보다 일반적인 프로그래밍 언어를 통하여 하게 되고 그 결과 개발자들이 2개의 일을 서로 다른 별개의 일로 생각하게 되기 때문이다.

UML의 인기가 높아지며 사용이 늘어나면서 이와 같은 상황은 조금 개선될 수 있다. UML은 소프트웨어 업계에서 널리 받아들여지고 있으며 그 중에서도 State Machine Diagram은 클래스의 행위를 나타내는 데에 많이 쓰이고 있다. State Machine Diagram은 정형적으로 정의된 그림이어서 이 그림에서 소스 코드를 생성하거나 테스트 케이스를 생성하는 것이 가능하다. 또한, State Machine Diagram을 이용한 모델 체킹도 가능하여서 여기에 관련된 많은 연구들이 진행되어왔다. 몇몇의 상업적인 CASE 툴들은 이 기술들을 이미 상용화하여 사용하고 있기도 하다.

2. 관련 연구

Petri-Net은 시스템의 행위를 나타내기 위해 아카데믹 서클에서는 오래 전부터 많이 쓰여 왔다. [1] Petri-Net은 token을 이용하여 리소스를 표현할 수 있는 방법을 제공해 데드락이 일어날 수 있는 작동을 설명하기가 좋다. 하지만 Petri-Net은 산업계에서는 그리 많이 사용되고 있지가 않은데 이는 Petri-Net이 설계에 쓰일 경우 이 설계가 곧장 코드의 구현에 쓰이기 힘들기 때문이다. 개발자는 이를 부담으로 여기게 되어 이를 실제 개발에 적용시키기가 쉽지 않다.

State Machine Diagram 역시 데드락 탐지에 사용할 수 있고 이에 관한 연구들도 여러 가지가 있는데 그 중 하나는 Kaveh의 연구이다[2]. 이 논문은 State Machine Diagram을 이용하고 있으며 동기화를 이루어야 할 부분을 <<synchronous>>라는 Stereotype을 이용하고 있다. State Machine Diagram들은 FSP (Finite state Processes)로 변환되어 실제 분석은 FSP로 하게 된다. [3의 경우는 보다 많이 알려진 Promela를 사용하여 실제 분석을 하게 되는데 vUML[3]은 UML State Machine Diagram를 이용하여 설계를 하면 이 설계에서 자동으로 Promela specification을 생성하여 실제 분석은 잘 알려진 SPIN Checker를 이용하게 된다. 이 논문은 '식사하는 철학자들' 예제를 사용하고 있는데 이 예제는 본 논문에서도 사용되고 있다.

최근의 CASE (Computer Aided Software Engineering) 도구들은 UML State Machine Diagram에서 소스 코드를 생성해 낼 수 있다. CASE 도구들의 도움을 받아 개발자들은 소프트웨어를 설계하고 이를 그대로 소스 코드로 변환할 수 있다. 이는 설계와 구현의 일관성을 보장해주게 된다. 특히 State Machine Diagram이 이러한 모델-코드 변환의 큰 부분을 담당하고 있다. 하지만, 일반적으로 State Machine Diagram은 하나의 개별 클래스의 동작을 설명하는데 쓰이므로 이러한 State Machine Diagram을 통상적인 Model-Checking기법으로 사용하기에는 한계가 있게 된다. 예를 들어 이 논문에서 언급하고 있는 deadlock은 멀티 쓰레드 프로그램에서 생기므로 이를 탐지하기 위해서는 시스템 범위의 동작이 필요하게 된다.

이 논문에서는 위의 문제를 해결하기 위해 하나의 프로그램에 있는 여러 개의 State Machine Diagram들을 모아서 하나의 커다란 시스템 범위의 State Machine Diagram을 만드는 방법을 논한다. 이러한 State Machine Diagram은 시스템 동작을 정확하게 설명할 수 있으므로 일반적인 Model-Checking 기법을 이용하여 시스템을 해석할 수 있게 된다.

3. 시스템의 데드락 탐지

이 논문에서 사용하는 기법은 기본적으로는 모델 체킹 기법을 시스템적으로 사용하는 것이라고 할 수 있다. 이를 위해서는

1) 개별의 State Machine Diagram들을 합쳐서 하나의 커다란 시스템 범위의 State Machine Diagram으로 만든다.
2) 모델 체킹을 하기 위해 이 커다란 State Machine Diagram을 미리 정한 Stereotype을 이용한 synchronization과 externalization을 이용하여 크기를 줄인다.

3) 데드락이 탐지됨을 보인다.

3.1 Stereotype 확장

Stereotype은 UML을 확장하는 방법이다. 시스템 범위의 State Machine Diagram을 만들기 위해서 새로운 Stereotype을 추가하여 이 Stereotype을 이용하여 시스템 State Machine Diagram을 만드는데 사용한다.

● <<Blocking>> Stereotype

소켓의 receive 함수나 semaphore/mutex의 lock 함수들은 프로그램의 실행을 block하게 된다. State Machine Diagram은 이러한 blocking에 대한 표기법이 없다. 그래서 실제 설계에서는 이러한 blocking 함수가 transition의 action 파트에 서술되어 프로그램의 실행이 transition에서 멈추게 되는 것이 보편적인 표기가 된다. 하지만 이는 직관적이지 못하고 프로그램 실행이 blocking이 되어있다는 표시로도 부족하여 <<Blocking>> Stereotype을 추가하여 이 Stereotype을 가지는 State는 현재 blocking이 되어있다는 것을 쉽게 표현하도록 한다. 실제 blocking 함수는 이 State의 entry action에 표시를 하도록 하며, 이 State는 단 하나의 밖으로 나가는 transition을 가지게 되는데 이 transition은 trigger가 없는, 즉 null-triggered transition이 된다. 그림 1에 <<Blocking>> Stereotype을 가지는 State의 예를 보인다. Entry action에 blocking 함수인 semaphore의 take함수가 사용됨도 알 수 있으며, null-triggered transition도 보이고 있다.

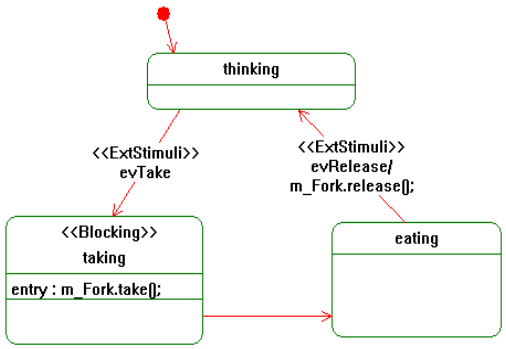

그림 1. <<Blocking>> stereotype
Fig 1. <<Blocking>> stereotype

● <<Available>>/<<Taken>> Stereotype

ssemaphore와 mutex는 시스템을 잠그는 lock 자원으로 많이 사용된다. 프로그램의 실행을 동기화하기 위하여 쓰레드들은 lock을 얻어야지만 다음으로 진행을 하게 된다. 이러한 lock으로 사용되는 semaphore/mutex의 동작을 State Machine Diagram으로 설명하려면 이 자원을 얻을 수 있는지를 표시해주어야 좋다. State의 이름으로 정해줄 수도 있지만 좀 더 쉬운 접근 방법으로 새로운 Stereotype으로 <<Available>>과 <<Taken>>을 추가한다.

<<Blocking>> Stereotype과 <<Available>> / <<Taken>> Stereotype은 같이 쓰일 경우 해석에 도움을 주게 된다. State들이 합쳐지게 되면 한 State가 여러 개의 Stereotype을 가지게 되는데, <<Taken>> Stereotype을 가지는 State에서 나가는 null-triggered transition은 유효한 transition이 아니다. 이는 이미 lock 자원을 누군가 가지고 있기 때문에 이 자원이 available한 상태가 되어야 이를 획득하여 다음으로 나아갈 수 있기 때문이다. 그러므로 이러한 transition은 안전하게 무시될 수 있다. 그림 3은 이러한 무효한 transition을 보여준다. Fork를 philB가 가지고 있는 상태이므로 philA는 그 자신의 State를 바꿀 수 없다.

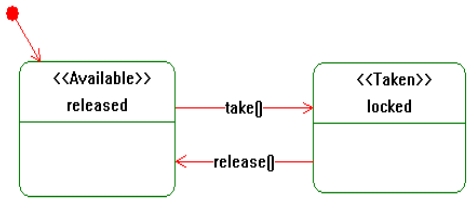

그림 2. <<Available>>/<<Taken>> stereotype
Fig 2. <<Available>>/<<Taken>> stereotype

● <<ExtStimuli>> Stereotype

<<ExtStimuli>> Stereotype을 설명하려면 작은 내장형 시스템이 가장 적절할 것이다. 만약 작은 내장형 시스템이 몇 개의 버튼을 가지고 있다면 이 버튼들을 클릭하는 것은 각각 다른 이벤트로 매핑 하는 것이 가능할 것이다. 전체 시스템이 많은 transition을 가지고 있다고 하더라도 실제로 유저가 직접 일으키게 되는 transition의 수는 많지 않게 된다. 시스템의 설계자는 <<ExtStimuli>> Stereotype을 그러한 transition에 추가하게 된다. <<ExtStimuli>>를 가지고 있지 않은 transition은 내부

적인 transition으로 취급되며 이는 사용자가 외부에서 볼수 없는 transition으로 가정된다. 이 Stereotype은 그림 1 에서 사용되고 있는데, 'evTake'와 'evRelease'가 사용자가 직접 발생시키는 event가 된다. 그림 2에는 'take()'와 'release()'가 trigger로 사용되고 있는데 이 trigger들은 시스템에 의해서 내부적으로 불리고 있는 것들이다.

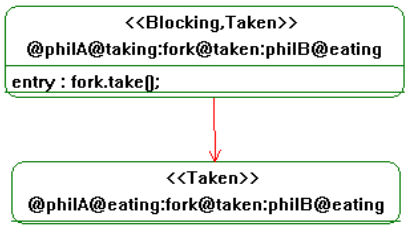

그림 3. 무효한 트랜지션
Fig 3. Invalid transition

3.2 데드락이 없는 시스템

제안하는 방법이 작동되는 것을 설명하기 위해 간단한 예제를 보여준다. 첫 번째는 데드락이 없는 시스템으로 잘 알려진 '식사하는 철학자들' 예제를 이용하도록 하겠다. 우선 가장 간단한 세팅으로 2명의 철학자가 하나의 포크를 같이 쓰는 경우를 보는데, 여기서 포크는 mutex로 생각하면 된다. 2명의 철학자가 번갈아 가면서 포크를 사용할 것이므로 이 시스템은 데드락이 되지 않는다. 이 시스템의 Class Diagram은 그림 1이다. Philosopher 클래스는 굵은 수직선을 가지는 'Active'클래스로 표시되어 있음을 주목하자. 철학자는 그 자신의 컨텍스트를 가지고 움직이며 이는 개별의 쓰레드로 표시가 된다. Message Queue를 가지는 구현을 사용한다면, 각각 개별적인 Message Queue를 가지고 필요한 이벤트를 받는 걸로 해석도 가능할 것이다.

철학자와 포크는 각각 State Machine Diagram을 가지게 되는데 그림 1은 철학자의 State Machine Diagram이고, 그림 2는 포크의 State Machine Diagram이다. 실제 프로그램이 돌아갈 때는 두 명의 철학자와 하나의 포크가 사용될 것이다. Builder 클래스가 각 클래스의 객체를 만들고 관계를 맺어주는데 사용된다(그림 3).

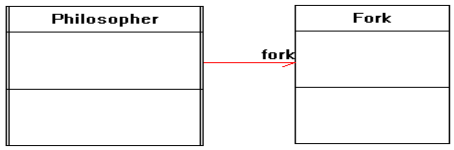

그림 4. 연관관계의 두 클래스
Fig 4. Two classes with association relationship

이 간단한 예제의 경우 전체 시스템의 행위는 3개의 State Machine Diagram을 하나로 합치는 것으로 설명될 수 있다. 그림 6에 여러 State가 하나로 합쳐진 것을 보인다. 그림 6에서 심벌 ⓧ는 merging 오퍼레이터와 그 수학적인 특성이다.

- 규칙 1) 교환 법칙
  SMD1 ⓧ SMD2 = SMD2 ⓧ SMD1
- 규칙 2) 결합 법칙
  (SMD1 ⓧ SMD2) ⓧ SMD3 =
  SMD1 ⓧ (SMD2 ⓧ SMD3)

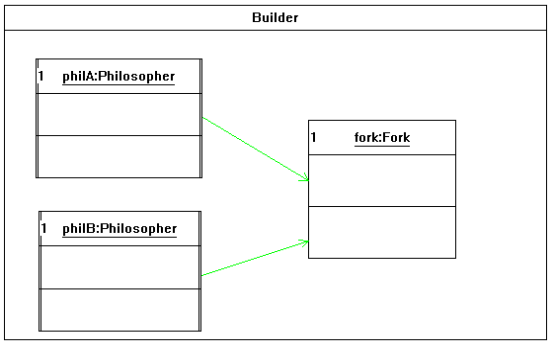

그림 5. 3개의 인스턴스를 가진 빌더
Fig 5. Builder with 3 instances

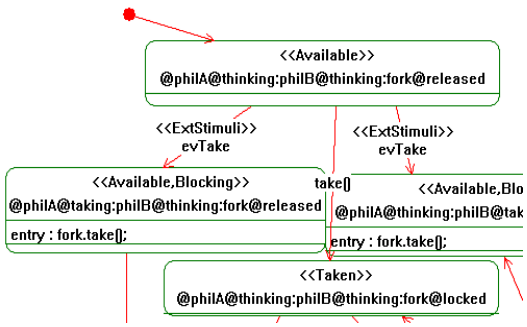

그림 6. 병합 상태
Fig 6. Merged States

State Machine Diagram들을 합하는 것은 쉽다. 이것은 근본적으로 Concurrent State들이 제거된 EFSM (Extended Finite state Machine)과 같다. ([4]) 이 예제에서 2개의 State Machine Diagram들이 각각 3, 2개의 State들을 가지므로, 결과로 나오는 State Machine Diagram은 3 x 3 x 2 = 18 개의 State를 가지게 된다. 모든 State들이 하나의 들어오는 transition과 하나의 나가는 transition을 가지므로 3개의 State가 하나로 합해지면 그 State는 3개의 나가는 transition을 가지게 된다. 이를 모두 합하면 54개의 transition이 된다. State Machine Diagram들을 합하는 것은 프로그램에서 자동으로 할 수 있고 그 결과를 그림 7에 보인다.

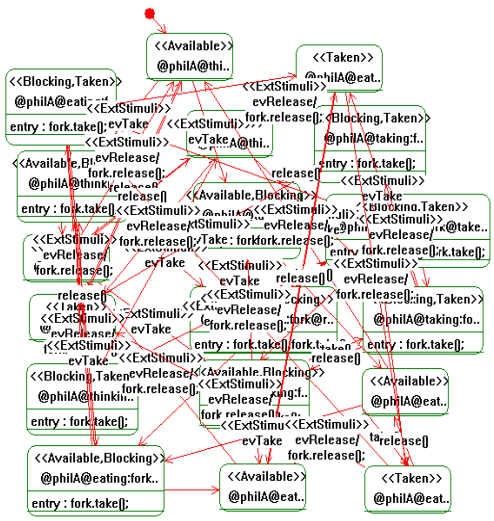

그림 7. 통합된 State Machine Diagram
Fig 7. Merged State Machine Diagram

Merged State Machine Diagram은 보기에 굉장히 복잡하다. 또한, 이 복잡한 State Machine Diagram은 실제 시스템의 행위를 설명하는 것과도 차이가 있으며, 자세히 살펴본다면 실제 행위와 더 가깝게 State Machine Diagram을 간단하게 만들 수 있다.

Synchronization을 살펴보자. Synchronization의 개념은 CSP (Communicating Sequence Process) [5]에 잘 설명이 되어있으며 SPIN과 같은 정형 기법에는 channel을 통해서 설명하고 있다. [6]에서도 Synchronization을 사용하고 있다. 이 논문의 예제에서는 2가지 종류의 synch- ronization 이 일어날 수 있는데, 기본적으로 둘은 같은 원리로 발생한다. Philosopher의 State Machine Diagram을 보면 'eating' state에서 'thinking' state로 가는 transition에 'fork. release()'라는 액션이 있음을 볼 수 있다. 이 'release()'라는 함수는 Fork의 State Machine Diagram에서는 'taken' state에서 'released' state로 가는 trigger 역할을 함을 볼 수 있다. 즉, Philosopher의 State Machine Diagram에 있는 transition은 Fork의 State Machine Diagram에서의 transition을 발생하게 되고 여기에서 동기화 즉, synchronization이 일어나게 된다. 그림 8에서 이를 설명한다.

Blocking을 하는 State에 대해서 또 하나의 synchronization을 정의할 수 있다. Philosopher의 state 'taking'은 그 엔트리 액션으로 'fork.take()'를 가지고 있다. Fork의 State Machine Diagram에서는 'take()'가 transition을 일으키는 trigger로 사용되고 있다. 만약 사용자가 evTake라는 이벤트를 Philosopher의 객체에게 보내면 이는 다시 Fork의 객체에게 trigger를 주어서 transition을 일으키게 되며 이러한 동기화로 2개의 transition은 하나로 만들어 질수 있다. 이것은 그림 9에서 설명되고 있다. (하지만, 이것이 곧 동기화되는 2개의 transition 사이에 있는 State를 없앨 수 있다는 것은 아니다. 이 State는 다른 transition들도 가지고 있을 수 있기 때문이다.)

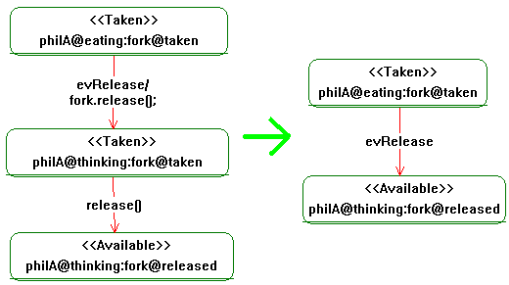

그림 8. 동기화 1
Fig 8. Synchronization I

여기에 하나 더 동기화에 관련된 과정이 있는데 이는 <<Available>>/<<Taken>> stereotype과 관련이 있다. Null-triggered 트랜지션은 그 이전의 State가 <<Available>> stereotype을 가지고 있어야만 한다. (즉, 이전의 State가 <<Taken>> Stereotype을 가지면 널 trigger가 허용되지 않는다.) 그러므로 이전 State가 <<Taken>> stereotype을 가지는 트랜지션은 없애는 것이 가능하다. 그림 9는 안전하게 없앨 수 있는 트랜지션

을 보여주고 있다.

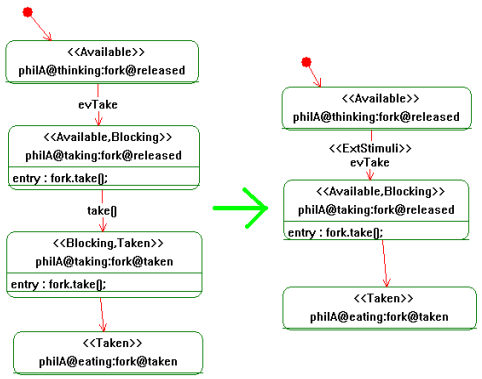

그림 9. 동기화 2
Fig 9. Synchronization II

Synchronization은 동기화된 트랜지션을 없애주지만 State를 없애지는 않는다. 이 방법이 위의 샘플에 적용이 된다면 State Machine Diagram이 줄어들게 되어 18개의 State와 40개의 트랜지션을 가지게 되는데, 이것 역시 분석을 하기에는 매우 복잡하다. 복잡도만 따진다면 처음의 State Machine Diagram과 크게 다를 바가 없다. 또한 복잡할 뿐만 아니라 이 State Machine Diagram은 실제 시스템의 행위를 정확히 표현하고 있지도 않다. 그래서 제대로 State Machine Diagram을 이용할 수 있도록 Stereotype <<ExtStimuli>>를 사용하여 externalization을 할 수 있다. 다음은 상기의 Stereotype을 이용하여 State들을 줄이는 규칙들이다.

규칙 1) 만약 <<ExtStimuli>> Stereotype을 전혀 가지고 있지 않다면 그 State Machine Diagram은 완전히 무시될 수 있다.

규칙 2) 만약 어떤 State가 <<ExtStimuli>> 트랜지션(들어오는 거든 나가는 거든)을 가지지 않는다면 그 State는 없앨 수 있다.

규칙 3) <<ExtStimuli>>를 가지지 않는 트랜지션은 없앨 수 있다.

규칙 4) Null-Trigger 와 Timeout은 <<ExtStimuli>>로 간주된다.

위의 규칙을 적용하면 샘플의 State Machine Diagram은 다음과 같이 된다. (18개의 State와 26개의 transition을 가지게 된다.)

Externalization은 무효한 internal transition 들을 없애는 과정이라고 볼 수 있다. 유효한 internal transition은 이미 synchronization 과정에서 없어졌으므로 synchronization 후의 State Machine Diagram에 남아있는 <<Ext>>가 없는 transition들은 모두 무효한 것들이다. 그림 10에는 모두 18개의 State가 있으나 자세히 살펴보면 시작점인 Default State에서 도달할 수 없는 State들이 있음을 알 수 있다. (Fig. 10에서는 아래쪽에 모여 있는 State들이 Default State에서 도달할 수 없는 State들이다.)

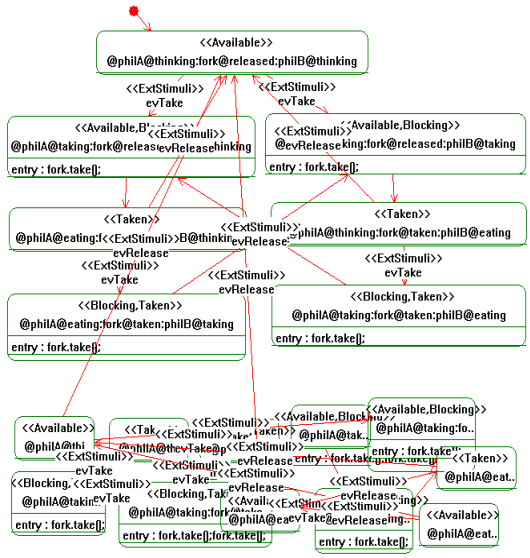

그림 10. Externalized State Machine Diagram
Fig 10. Externalized State Machine Diagram

이러한 State들은 유효한 것들이 아니므로 State들을 더 줄이기 위해서 Reachability Test 를 각각의 State들에 해줄 수 있다. 이 테스트는 각각의 State가 Default State로부터 도달할 수 있는지를 체크한다. deadlock 샘플에서 우리는 데드락을 체크하는 방법으로 '데드락이 없는 것은 모든 State들이 최소한 하나의 밖으로 나가는 (outgoing) transition을 가져야 한다.'는 간단한 규칙을 정하여서 이 Reachability Test는 꼭 필요한 테스트이다. 이 테스트를 거친 마지막 State Machine Diagram을 그림 11에 보인다.

그림 11. State Machine Diagram with RT
Fig 11. State Machine Diagram with RT

이 State Machine Diagram은 최초의 18개의 State와 비교하여 7개의 State만 가지고 있으며 최초의 54개 트랜지션과 비교하면 단지 10개의 트랜지션만 가지고 있다. 표 1은 각각의 경우에 대해 State와 트랜지션의 개수를 보여주고 있다.

표 1. 데드락이 없는 시스템의 상태와 트랜시션 수
Table 1. Number of States and Transitions of deadlock-free system

	#States	#Transition
Original	18	54
Synchronized	18	40
Externalized	18	26
Externalized with RT	7	10

최종적으로 우리는 이 시스템이 데드락이 없는지 체크해볼 수 있다. 데드락이 있는지 검사하는 방법은 앞에서 언급한 바와 같이 매우 간단하여 각각의 State에서 나가는(outgoing) 트랜지션이 있는지를 검사해보면 된다. 위의 샘플은 그러한 State를 가지고 있지 않으므로 이 시스템은 데드락이 없는 시스템이다.

Externalized State Machine Diagram은 최초 State Machine Diagram의 줄어든 형태이다. Externalization은 안전한 줄임 방법인데 이는 기존의 State Machine Diagram을 변경 없이 줄이는 형태이기 때문이다. 이상의 방법은 3개의 과정으로 되어있다.

- Synchronization - 만약 트랜지션이 이벤트를 보내는 액션을 가지고 있어서 이 이벤트가 다른 State Machine Diagram의 트랜지션을 발생시키면 2개의 트랜지션은 하나로 합쳐질 수 있다.
- Externalization - <<ExtStimuli>> stereotype을 가지지 않는 트랜지션은 없앨 수 있다.
- Reachability Test - Default State에서 도달할 수 없는 State들은 없앨 수 있다.

3.3 데드락 시스템

이번에는 데드락이 일어나는 시스템의 경우를 살펴본다. 두 명의 철학자가 2개의 젓가락을 사용하는 경우이다. 항상 왼쪽 젓가락을 먼저 들어 올리고 그 후 오른쪽 젓가락을 사용한다. Chopstick의 State Machine Diagram은 Fork의 State Machine Diagram과 동일하며 여기에 보이지 않는다. Philosopher의 State Machine Diagram은 조금 더 복잡해진다. Philosopher는 이제 2개의 blocking state를 포함하여 6개의 state를 가지는 State Machine Diagram을 가진다.

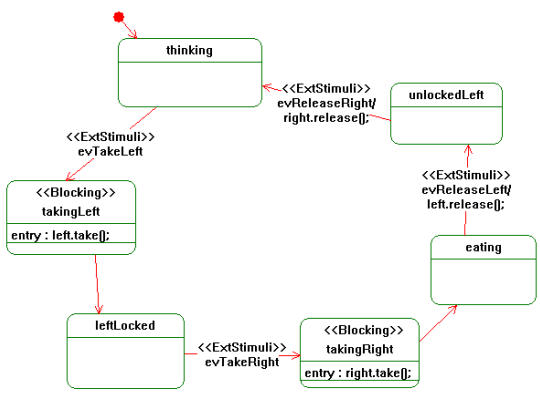

그림 12. State Machine Diagram of Philosopher
Fig 12. State Machine Diagram of Philosopher

모든 State Machine Diagram을 합하게 되면 나오는 State Machine Diagram은 6 x 6 x 2 x 2 = 144 states를 가지게 된다. 모두 4개의 객체가 있어서 (모든 State들은 단지 하나의 들어오고 나가는 트랜지션을 가진다.) 총 트랜지션의 개수는 144 x 4 = 576이 된다. 이는 그리기에는 너무 복잡한 그림이어서 여기에 보이지는 않는다.

만약 synchronization이 여기에 적용된다면 트랜지션의 개수는 440개로 줄어들고 State의 개수는 변하지 않는다. 여기에 다시 externalization이 적용되면 트랜지션의 개수는 216개로 줄어들게 되며 State의 개수는 변하지 않

는다. 문제는 단순히 숫자가 많아서 복잡하다는 것만이 아니고 이 State Machine Diagram이 시스템의 행위를 제대로 표현하지 못하고 있다는 것이어서 현 상태로 deadlock을 검사해보면 이 시스템은 deadlock-free로 나오게 된다. 하지만 마지막으로 Reachability Test를 하게 되면 State의 개수는 21개로 줄어들고 트랜지션의 개수는 28개로 줄어들게 된다. 이 State Machine Diagram 역시 그림으로 보이기에 간단하지는 않으나 그림 13 에 마지막 State Machine Diagram을 보인다.

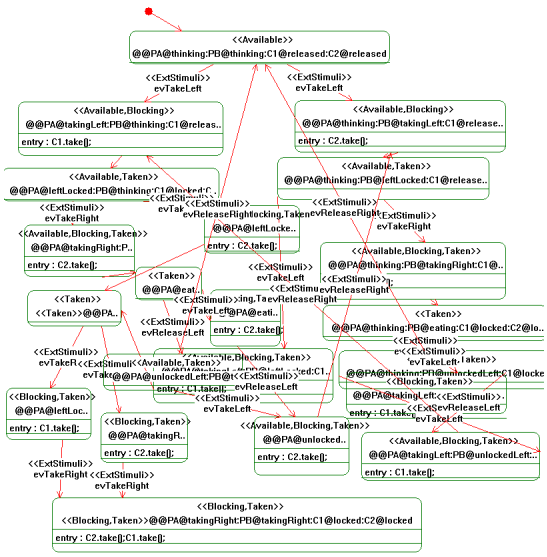

그림 13. Final State Machine Diagram
Fig 13. Final State Machine Diagram

그림 13을 자세히 보면 가장 아래에 있는 State가 deadlock이 일어나는 State이다. 그림으로 State Machine Diagram을 보이고는 있지만 실제 deadlock이 존재하는 지 검사하는 방법으로는 밖으로 나가는 트랜지션을 각 State들이 가지고 있는 지만 검사하는 걸로 충분하다. 모델 체킹에서는 어떤 특별한 케이스에 도달하는 path를 보여주는 경우가 많다. (이를 보통 counter-example이라고 한다.) 이를 수행하는 방법도 무척 쉬운데 이는 이미 우리가 Reachability test를 각 State에 했고, 그 State가 그 테스트를 이미 통과했기 때문이다. 표 2 에는 이 경우의 State와 트랜지션의 개수를 보인다.

표 2. 데드락 시스템의 상태와 트랜지션 수
Table 2. Number of States and Transitions of deadlock system

	#States	#Transition
Original	144	576
Synchronized	144	440
Externalized	144	216
Externalized with RT	21	28

4. 결론

이 논문은 시스템에 있는 모든 State Machine Diagram 들을 더하여서 시스템의 행위를 나타내는 시스템 범위의 State Machine Diagram을 만드는 방법을 제안한다. 이 방법은 synchronization과 externalization을 사용하며 프로그램에서 자동으로 적용된다. 이러한 시스템 행위의 유용성을 검증하기 위해 잘 알려진 '철학자들의 식사' 예제를 이용하여 deadlock의 탐지가 잘 됨을 확인하였다.

최신의 CASE(Computer Aided Software Engineering) 툴들은 State Machine Diagram에서 소스코드를 생성해 낼 수 있다. 그 결과 모델과 구현의 동일성을 확보할 수가 있는데 이 논문에서 제안하는 시스템 State Machine Diagram을 자동으로 만드는 방법과 모델 체킹 기법의 적용은 설계자와 개발자 모두에게 도움이 될 것이다.

참 고 문 헌

[1] R. G. Pettit, H. Gomaa, "Validation of dynamic behavior in UML using colored Petri nets", In Proc. of UML'2000, 2000

[2] N. Kaveh and W. Emmerich. "Deadlock Detection in Distributed Object Systems". In Joint Proc. of the 8th European Software Engineering Conference and the 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Vienna, Austria, pp. 44~51. ACM Press, 2001.

[3] J. Lilius and I. Paltor. "A tool for verifying UML models". In IEEE International Conference on Automated Software Engineering, volume 14, 1999.

[4] YG Kim, HS Hong, SM Cho, DH Bae and SD Cha. "Test Cases Generation from UML state Diagrams". IEE Proceedings Software Vol. 146 No.4 1999.

[5] Milner R., “Communication and Concurrency”, Prentice -Hall, 1st Edition, 1995.
[6] Jean Hartmann, Claudio Imoberdorf, Michael Meisinger. “UML-Based Integration Testing”, ISSTA’00, Portland, Oregon, 2000
[7] Hoare C. A. R, Communicating Sequential Processes. Prentice Hall, 1987.
[8] E. M. Clarke, Jr., O. Grumberg and D. A. Peled, “Model Checking”. MIT Press, 1999.
[9] S. Chaki, J. Ivers, N. Sharygina, and K. Wallnau. “The comfort reasoning framework”. In Computer Aided Verification, 2005.
[10] K. M. Chandy, J. MISRA, and L. Haas. “Distributed deadlock detection”. ACM Trans. Comput. Syst. I,2, 144-156, 1983

저 자 소 개

민 현 석(Hyun-Seok Min)

- 1992년 2월 : 서울대학교 공과대학원 기계공학과 (학사)
- 1994년 2월 : 서울대학교 공과대학원 기계공학과 (석사)
- 2006년 12월 ~ 현재 : 다한테크 수석 컨설턴트

<관심분야> : Software Engineering, 객체지향 모델링, 모델링 및 시뮬레이션