병렬 객체지향 시스템의 검증
(Model Checking of Concurrent Object-Oriented Systems)

조 승 모 † 김 영 곤 † 배 두 환 †† 변 성 원 ††† 김 상 택 †††
(Seung Mo Cho) (Young Gon Kim) (Doo Hwan Bae) (Sung Won Byun) (Sang Taek Kim)

요 약 모델체킹은 검증하려는 대상 시스템의 동작 모델이, 그 시스템이 만족해야 할 성질을 만족시키는지를, 시스템의 상태공간을 검사해 봄으로써 알아보는 정형 검증 기법의 하나이다. 이러한 모델체킹 기법을 병렬 객체지향 시스템에 적용하기 위해 기존의 모델체커인 SPIN에서 지원하는 모델링 언어인 Promela를 병렬객체지향 개념을 추가하여 확장한 언어인 APromela를 제안하였다. 이는 Promela가 프로세스를 단위로 하는 병렬성만을 지원하는데 반해, 액터 모델에 기반한 객체지향 병렬성을 지원한다. 또한 우리는 이 언어로 작성된 모델을 자동으로 Promela로 변환하는 규칙을 제안하였다. 이를 통해, 기존의 모델체커를 이용해 병렬 객체지향 시스템의 검증을 수행할 수 있다. 이 언어의 응용으로 UML 로 기술된 명세의 검증을 수행하는 과정을 제시하였다.

Abstract Model checking is a formal verification technique which checks the consistency between a requirement specification and a behavior model of the system by explorating the state space of the model.We apply model checking to the formal verification of the concurrent object-oriented system, using an existing model checker SPIN which has been successful in verifying concurrent systems. First, we propose an Actor-based modeling language, called APromela, by extending the modeling language Promela which is a modeling language supported in SPIN. APromela supports not only all the primitives of Promela, but additional primitives needed to model concurrent object-oriented systems, such as class definition, object instantiation, message send, and synchronization.Second, we provide translation rules for mapping APromela's such modeling primitives to Promela's. As an application of APromela, we suggest a verification method for UML models. By giving an example of specification, translation, and verification, we also demonstrate the applicability of our proposed approach, and discuss the limitations and further research issues.

1. 서 론

프로그램의 복잡성을 제어하기 위한 수단으로 객체지향 프로그래밍 기법은 이제 소프트웨어 공학의 기본적인 방법론의 하나로 굳게 자리잡았다고 할 수 있다. 또한 실세계의 객체들이 병렬적으로 분산되어 동작하는 것을 보면, 객체지향 프로그래밍에서 병렬성을 지원하는 쪽으로 발전하는 것은 당연한 귀결이라고 볼 수 있다.

† 비 회 원 : 한국과학기술원 전산학과
seung@salmosa.kaist.ac.kr
ygkim@salmosa.kaist.ac.kr
†† 종신회원 : 한국과학기술원 전산학과 교수
bae@salmosa.kaist.ac.kr
††† 비 회 원 : 한국통신 멀티미디어연구소 연구원
논문접수 : 1999년 2월 2일
심사완료 : 1999년 10월 25일

이런 병렬 객체지향 시스템을 만들때는 만드는 시스템의 정확성에 대한 검증을 수행할 방법이 필요하다. 이 경우, 일반적인 순차적 프로그램과 마찬가지로 테스팅이 주로 사용되는 방법이다. 하지만 병렬성을 테스팅하는데 드는 어려움으로 인해, 만드려는 시스템을 모델링하고 그것을 대상으로 자동적인 검증을 수행하여 에러를 검출하고 정확성을 검증하는 정형 검증 기법들이 제시되어 왔다. 이론적인 발전과 컴퓨팅 파워의 증가로 인해, 이런 식으로 검증할 수 있는 모델의 복잡도는 계속 증가해 왔다.

모델체킹[5, 6, 8] 은 이러한 정형 검증 기법의 하나이다. 소프트웨어 혹은 하드웨어 시스템의 설계자는 대상 시스템의 동작을 (일반적으로 추상화를 거쳐) 모델링한다. 모델은 정형적 의미를 가지는 언어로 기술되어 검

증 도구에 입력된다. 또한 모델이 만족해야 할 성질들도 설계자가 기술하게 된다. 이는 데드록 등의 주로 병렬성에 관한 성질이 된다. 검증도구(모델체커)는 그 모델이 가질 수 있는 모든 상태들을 검사하여 이 성질이 만족되는지의 여부를 조사한다. 만일 에러가 발견되면 그런 상태로 도달가능한 경로를 제시한다. 이 정보는 디버깅에서 유용한 정보로 사용되게 된다. 이러한 모델체킹 기법은 프로토콜 검증[15], 회로분석[6], 실시간 시스템[5] 등에 대해 제안, 사용되었다. 하지만, 병렬 객체지향 시스템에 대해서 모델체킹을 적용하는 방법에 대해서는 아직 본격적인 연구가 없는 형편이다.

본 연구에서는 기존의 병렬 프로그램을 위한 모델 체커인 SPIN[15, 16]을 이용하여 병렬 객체지향 시스템을 모델링하고 검증하는 방법을 제안하고자 한다. 이는 SPIN에서 지원되는 모델링 언어인 Promela에 객체지향을 지원할 수 있는 메커니즘을 추가하는 식으로 구현되었다. 이 언어로 사용자가 모델링 하면, 선처리기(preprocessor)가 그것을 Promela 로 변환해 주고, 그것이 다시 SPIN의 입력으로 사용되게 된다.

이러한 모델링 언어를 설계하기 위해서는 우선 병렬 객체지향 시스템의 기본이 되는 계산 모델을 선정해야 한다. 여러가지 모델이 제안되어 있는 중에서 우리는 액터 모델[3,4]을 선택했다. 이 모델은 여러 기존의 언어들[2, 17, 18, 19, 25]의 병렬 객체 모델로서 사용되었고, 이론적으로 명확하고 이해하기 쉬우며 직관적이라는 장점을 가지고 있다.

본 논문의 구성은 다음과 같다. 우선 2장에서 본 연구의 전체적인 흐름과 개발 과정에서의 위치에 대해 서술한다. 3장에는 관련 연구로 액터 모델과 SPIN 모델 체커에 대해서 살펴본다. 4장에서는 병렬 객체시스템 검증을 위한 모델링 언어인 APromela를 제안하고, 5장에서 APromela로 작성된 명세를 Promela 명세로 바꾸는 방법을 정의한다. 6장에서는 적용예로서 이 언어를 사용하여 UML 로 작성된 명세를 검증하는 한 방법을 제시한다. 그리고 7장에서 결론과 향후 연구과제에 대해 살펴본다.

2. 방법론 개요

시스템의 정확성의 검증은 일반적으로 그 시스템의 모델에 대해서 수행되게 된다. 이런 모델은 여러가지 정보로부터 생성될 수 있다. 시스템의 개발 과정 중에서, 모델은 요구사항 분석, 설계 등의 과정에서의 산출물로서 만들어진다. 개발이 끝난 후에는 결과물인 소스코드에서부터 역으로 좀더 추상적인 동작에 관한 모델을 추출할 수도 있다. 이때 일반적으로 모델링 자체는 사람에 의해 수행되게 된다. 따라서 모델링 언어는 일반적인 프로그래밍 언어와 유사하게, 표현력을 위해 여러가지 추상화 (abstraction) 을 지원해야 한다.

모델링할 대상이 병렬 객체지향 시스템일 경우, 모델링 언어가 그런 병렬 객체지향 개념을 지원하는 것이 좋다. 이는 모델링 작업을 쉽게 하고, 모델과 대상 시스템간의 불일치가 발생할 가능성을 줄여준다. 모델링 언어를 설계할때는 표현력 뿐만 아니라 분석력에도 주의를 기울여야 한다. 우리는 충분한 표현력을 위해, 기존의 병렬 객체지향 언어의 모델로 사용되는 액터 모델을 제안하는 모델링 언어의 기본 모델로 선택했다. 분석을 위해서는 이 언어를, 기존의 모델체킹 도구인 SPIN에서 사용되는 언어인 Promela의 확장으로 정의하였다. 새로 제안한 언어로 작성된 병렬 객체지향 모델은 기존의 Promela 모델로 변환되어 SPIN 을 이용하여 분석되게 된다.

이상에서 제시한 과정을 정리하면 그림 1과 같다. 여기서 굵은 실선으로 표시된 부분이 본 연구의 직접적인 범위에 포함되는 부분이다.

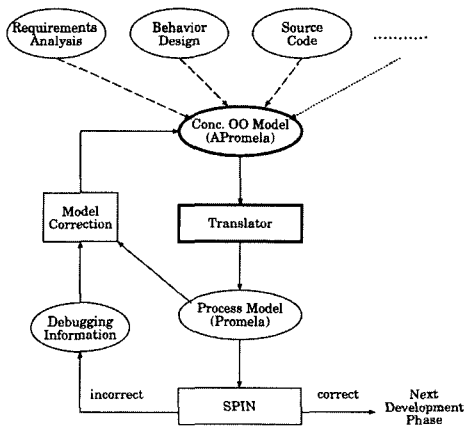

그림 1 방법론 개요

3. 연구 배경

3.1 액터 모델

액터(actor)는 비동기적, 자율적으로 동작하는 시스템의 구성요소들이다. 각각은 자기가 관리하는 자료들과 그 자료들을 조작할 수 있는 프로시듀어들로 정의된다. 메시지를 받으면 한 액터는 그 메시지에 해당하는 프로

시듀어를 호출하여 실행시킨다. 수행이 끝나면 다시 메시지큐에서 하나의 메시지를 꺼내어 처리한다. 따라서 액터들을 논리적으로 분산되어 병렬적으로 동작하는 객체들의 모델로 볼 수 있다.

메시지를 주고 받기 위해서 각 액터(객체)들은 고유한 주소를 가진다. 이 주소들은 다시 메시지내에 포함되어 서로간에 전달될 수 있다. 이는 액터에 기반한 시스템이 동적으로 통신 구조를 바꿀 수 있는 기능을 가짐을 의미한다. 각 프로지듀어 내에서 수행할 수 있는 일은, 새로운 액터의 생성(create), 다른 액터로의 메시지 전송(send), 그 액터의 상태 변화(assignment) 등이다. 메시지큐를 통해 순차적으로 메시지를 처리하므로, 각 액터의 상태변화는 원자적(atomic) 하다. 액터들간의 동기화는 메시지 단위로 일어나는데, 이는 현재의 자신의 상태를 참고하여 주어진 메시지를 처리할 것인지의 여부를 결정하는 국지 조건(local constraints)로 기술하게 된다.

3.2 SPIN과 Promela

모델체커를 사용할때 사용자는, 검증하고자 하는 시스템의 모델과 그 모델에 대해 검증하고자 하는 성질, 이 두가지를 입력으로 제공해야 한다. 이중에서 검증하고자 하는 성질은 주로 시제논리로 기술하게 된다. 그리고 시스템의 모델을 기술하기 위해서 필요한 모델링 언어를 지원하는데, 그 언어가 Promela 이다.

*Promela*는 통신 규약과 같은 병렬 시스템을 모델링하기 위해 설계된 언어이다. Promela로 작성된 프로그램은 여러개의 병렬 프로세스들로 구성되게 되는데, 각 프로세스들은 채널을 통한 비동기적 통신이나 랑데뷰에 의한 동기적 통신을 수행한다. 이 언어는 C 언어의 문법에 상당부분 기반하고 있으며, 시스템을 추상적 레벨에서 모델링하는 것을 목적으로 하므로 제한된 타입만을 지원한다.

Promela로 기술된 모델은 SPIN 도구에 의해 분석된다. SPIN은 모델체킹 외에도 시뮬레이션이나 랜덤 테스팅 등의 분석 기능도 제공한다. 따라서 사용자는 이러한 기능들을 통해 어느정도 정확성에 대해 확신을 가질 수 있는 모델을 작성한 다음 최종적으로 모델체킹을 통해 검증을 수행하게 된다. 모델체킹의 효율을 높이기 위해 여러가지 상태 검사 알고리즘이 구현되어 있다.

4. APromela

본 장에서 우리는 액터 모델에 기반한 Promela 언어의 확장인 APromela(Actor-based Promela)를 제안한다. 이 언어를 사용함으로써, 설계자는 병렬 객체지향 시스템의 모델을 보다 직접적이고 직관적인 형태로 기술할 수 있게 된다. 이를 위해 몇개의 언어구조를 추가하였다. 이들은 클래스 정의, 객체 생성, 메시지 전송, 동기화 등이다.

4.1 APromela의 구문

APromela의 구문은 [15] 에 제시된 Promela의 구문에 필요한 구조를 추가한 것이다. 따라서 다음과 같은 부분은 기존의 것과 동일하다.

- Lexical conventions
- Comments
- Identifiers
- Constants

그림 2에 APromela의 구문을 BNF 스타일로 제시하였다. 밑줄친 부분이 원래의 *Promela*에서 변경, 확장된 부분이다. + 는 한번 이상의 반복을, * 는 0회 이상의 반복을 나타낸다. [ ] 는 생략가능한 부분을 나타낸다. 구문 트리의 말단 (terminals) 에 해당하는 부분은 대문자로, 그렇지 않은 부분은 소문자로 나타내었다.

```
program ::= { actor-def } * MAIN sequence
actor-def ::= ACTORTYPE NAME '(' [ acq_vars ] [ init ] { method } * '}'
acq_vars ::= VAR decl_lst
decl_lst ::= one_decl { ';' one_decl } *
one_decl ::= [ TYPE ivar { ',' ivar } * ]
ivar ::= var_dcl | var_dcl ASGN expr
var_dcl ::= NAME [ '[' CONST ']' ]
var_ref ::= NAME [ '[' expr ']' ]
init ::= INIT '(' [ decl_lst ] ')' sequence
method ::= METHOD NAME '(' [ decl_lst ] ')' [ RESTRAIN syn_cond ] sequence
syn_cond ::= '(' syn_cond ')'
        | syn_cond binop syn_cond
        | unop syn_cond
        | var_ref
        | CONST
sequence ::= step { ';' step } *
step ::= [ decl_lst ] stmnt
stmnt ::= var_ref ASGN expr
        | PRINT '(' STRING { ',' expr } * ')'
        | ASSERT expr
        | GOTO NAME
        | NAME ':' stmnt
        | IF options FI
        | DO options OD
        | BREAK
        | var_ref '<-' NAME '(' [ arg_lst] ')'

options ::= { SEP sequence } +

binop ::= '+' | '-' | ...
unop ::= '~' | '-' | SND
expr ::= '(' expr ')'
```

```
    | expr binop expr
    | unop expr
    | CREATE NAME '(' [ arg_lst] ')'
    | var_ref
    | CONST
    | var_ref '.' var_ref
    | var_ref '::' NAME

arg_lst ::= expr { ',' expr } *
```

그림 2 APromela 의 구문정의

4.2 APromela의 개요

Apromela로 작성된 모델은 두개의 부분으로 나뉘어 질 수 있다. 하나의 메인 프로그램과 여러 액터 정의 부분이다. 메인 프로그램은 몇개의 액터를 생성하고 메시지를 보냄으로써 수행을 초기화 한다. 액터 정의 부분들은 클래스 선언에 해당되는 것이다. 상속(inheritance)는 이 언어가 모델링 언어이기 때문에 단순성을 위해 도입되지 않았다. 또한 액터모델에 따라 전역변수는 사용되지 않는다.

4.2.1 액터 타입 정의

액터 타입 정의는 일반적인 객체지향언어에서의 클래스 선언에 해당하는 것으로, 그 액터가 가지는 인스턴스 변수들과 메소들로 구성된다. 특별한 메소드로 init 메소드를 가질 수 있는데, 이는 생성 시에 한번 수행된다. 각 메소드들은 인수를 가질 수 있으며, 그 내부에서 액터생성, 상태전이, 메시지 전송 등의 일을 할 수 있다.

4.2.2 액터 생성

액터는 create 명령으로 동적으로 생성된다. 그 명령은 정의된 액터 타입 이름과 init 메소드에 전달될 인수들을 인수로 가진다. 수행결과로 그 액터를 다루는데 필요한 주소를 넘겨받게 된다. 이는 그 액터와의 통신을 위해 필요한 것으로, 액터간에 주소를 주고받음으로써 동적인 객체 연결 구조를 구현할 수 있다.

4.2.3 메시지 전송

액터간의 비동기적 메시지 전송은 <- 연산자에 의해 기술된다. 이 연산자는 좌변에 메시지를 보낼 대상 액터의 주소를 가지고, 우변에 전송될 메시지와 그에 해당하는 인수를 가진다. 이 메시지는 비동기적으로 전송되고, 전송하는 액터는 바로 다음 동작을 수행한다.

4.2.4 동기 제약

객체간의 통신시에 전송하는 액터는 전송받는 액터의 상태를 알 수 없다. 따라서 그 메시지가 제대로 처리될 수 있기 위해서는 전송받는 액터가 자기 자신의 상태에 따라 메시지의 수신을 거부할 수 있는 기능이 필요하다. 이는 객체의 인스턴스 변수의 값에 대한 조건으로 기술된다.

4.3 예제

APromela의 사용을 보여주는 예제로, 생산자-소비자 문제를 모델링한 것을 제시한다. 이 시스템은 생산자(producer), 소비자(consumer), 유한버퍼(buffer), 이렇게 3개의 액터로 구성된다. 메시지들은 비동기적으로 전달되는데, 버퍼는 자기의 상태에 따라 부절절한 메시지 수신을 뒤로 연기하는 식으로 동기화를 한다. 생산자는 put 메시지를 버퍼에 보냄으로써 버퍼에 내용을 넣고, 소비자는 get 메시지를 버퍼에 보냄으로써 요구를 한다. 이 요구는 후에 버퍼가 소비자에게 retrieved 메시지를 보냄으로써 완료된다.

```
#define SIZE 3
actortype buffer {
   var
      bit data[SIZE]; byte count = 0; byte front = 0
   method put ( bit a ) restrain (count < SIZE)
      data[(front + count) % SIZE] = a; count ++

   method get ( actor ret ) restrian (count > 0)
      ret <- retrieved (data[front]);
      front = (front + 1) % SIZE; count --;
}
actortype producer {
   var
      bit next; actor buf
   init ( actor a )
      buf = a; this <- putting ()
   method putting ()
      buf <- put (next); next = ! next; this <- putting ()
}

actortype consumer {
   var
      actor buf;
   init ( actor a )
      buf = a; this <- getting ()
   method getting ()
      buf <- get (this)
   method retrieved ( bit b )
      printf("read num = %d\n", b); this <- getting ()
}

main {
   actor buf;
   actor proc;
   actor cons;
   buf = create buffer ();
   proc = create producer (buf);
   cons = create consumer (buf);
}
```

5. APromela에서 Promela로의 변환

새 언어를 제안하면서 그에 맞는 시뮬레이터와 검증기를 새로 처음부터 만드는 것 보다, APromela에서 새롭게 추가된 요소들을 Promela로 변환함으로써, SPIN이 제공하는 기능들을 사용할 수 있도록 고안하였다.

5.1 메시지 큐

각 액터들이 가지는 메시지 큐들은 Promela의 채널로 나타낸다. 이때, 원래 액터의 메시지큐는 무한한 길이를 가지지만, Promela의 채널은 유한하다는 점이 문제가 될 수 있다. 하지만 무한한 동작을 유한하게 제약시킨 모델도 검증시에 유용한 결과를 낼 수 있다는 점이 일반적으로 받아들여지고 있으므로[9], 이는 별 문제가 되지 않는다.

```
actortype actor_name {
  var
    var_declarations
  init ( init_parameters )
    init_method_body
  method method1_name ( method1_parameters )
    method1_body
  method method2_name ( method2_parameters )
    method2_body
  ⋮
}
```

그림 3 액터타입 정의

그림 3과 같이 정의된 액터에 대해서, 그 액터가 가지는 메시지큐를 나타내는 채널은 그림 4와 같이 구현될 수 있다. 이 채널의 각 요소는 메소드 이름과, 각 메소드들에 대한 인수들의 합집합이다. LENGTH는 채널의 사이즈로, 검증시에 사용가능한 리소스를 고려하여 사용자가 정해주게 된다.

```
chan this = [ LENGTH ] of { byte, method1_parameters,
method2_parameters, ... }
```

그림 4 액터를 위한 메시지큐의 정의

5.2 액터 정의

액터는 스레드와 상태, 메소드, 그리고 메시지큐를 가지는 객체이다. 큐의 구현은 전 절에서 살펴보았다. 액터는 이런 큐와 연결된 SPIN의 프로세스로 구현된다. 액터의 상태를 나타내는 인스턴스 변수들은 그 프로세스의 지역변수가 된다. 액터에서 메소드들은 받은 메시지에 따라서 호출되고, 중간에 인터럽트되지 않는다. 따라서 메시지들은 do ... od 구조 안에, 적절한 조건문과 함께 들어가게 된다. 이를 고려하면, 그림 3에 있는 액터 정의는 그림 5와 같은 Promela의 프로세스 타입 정의로 변환될 수 있다.

```
proctype actor_name ( chan temp_chan, init_parameters ) {
  var_declarations
  parameter_declarations      /* all parameters for the methods */
  initializing
  do
    :: atomic {
         this ? method1_name, parameters
         -> method1_body
       }
    :: atomic {
         this ? method2_name, parameters
         -> method2_body
       }
     ⋮
  od
}
```

그림 5 그림 3의 정의에서 변환된 프로세스 정의

액터 생성시의 상호작용에 관계된 부분인 *initializing* 과, 동기 조건이 부과될때의 수정에 관해서는 뒤에 설명한다.

5.3 액터 생성

액터의 생성은 Promela의 프로세스 생성 기능을 이용하여 구현될 수 있다. 이는 그 액터의 메시지큐인 채널을 생성하는 과정을 포함한다.

액터 생성은 다음과 같은 식으로 이루어 진다.

```
actor a;
⋮
a = create actor_name ( init_parameters ) ;
```

그림 6 액터 생성의 일반적 형식

actor 타입은 chan 타입으로 변환된다. 그리고 액터의 생성 시에는 그 바디에 해당하는 프로세스를 생성하고, 채널을 생성하게 된다. 이들은 동적으로 생성될 수 있다. Promela에서 동적으로 생성되는 요소는 프로세스이므로, 채널은 프로세스 내에서 선언되어야 한다. 이는 다시 액터를 생성한 객체쪽으로 돌려져서, 추후의 통신에 사용되게 된다. 이를 위해 임시적인 채널로 temp_chan를 사용하는데, 이는 모든 액터 생성시에 공통적으로 사용된다.

이런 방법으로 그림 6과 같은 액터 생성은 그림 7과

같은 Promela 구문으로 변환된다.

```
chan a;
chan temp_chan = [0] of { chan };
 :
atomic {
  run actor_name ( temp_chan, init_parameters ) ;
temp_chan ? a ;
}
```

그림 7 그림 6에서 변환된 프로세스 생성 부분

그리고 액터 생성시의 채널 초기화와 관련된 그림 5 내의 initializing 부분은, 그림 8과 같이 변환된다.

```
chan this = [ LENGTH ] of { byte, method1_parameters,
method2_parameters, ... }
init_method_body
temp_chan ! this;
```

그림 8 초기화를 위한 코드

5.4 동기 조건

액터의 동기조건은 각 메소드들을 if 구문으로 감싸는 것으로 쉽게 구현될 수 있다. 메시지를 받았을때, 액터는 현재의 자기 상태가 그 메소드에 기술된 동기조건에 부합하는지를 검사한다. 만일 부합하지 않는다면, 그 메시지는 다시 큐에 넣어져 추후에 다시 처리되게 된다.

```
method  method_name ( parameters)
  restrain condition
  method_body
```

그림 9 동기조건을 가진 메소드 정의

그림 9는 동기조건이 메소드에 부과되는 일반적인 형식이다. 이는 그림 10과 같은 형식으로 변환된다.

```
:: atomic { this ? method_name, parameters ->
  if
    :: condition -> method_body
    :: ! condition -> this ! method_name, parameters;
  fi  }
```

그림 10 그림 9에서의 변환

5.5 예제

이상의 규칙들을 가지고 APromela로 작성된 모델을 Promela 프로그램으로 변환시킬 수 있다. 4.3절에서 제시된 모델을 변환한 결과를 아래에 제시하였다. 이 결과를 SPIN 모델체커로 검증해 본 결과 수초 정도의 수행으로 데드락이 없음을 검증할 수 있었다.

```
#define SIZE 3
#define LENGTH 3

mtype = {buffer_put, buffer_get, producer_putting,
         consumer_getting, consumer_retrieved}

proctype buffer (chan temp_chan) {              /* actor buffer */
   bit data[SIZE]; byte count = 0; byte front = 0;
   bit a; chan ret;
   chan this = [LENGTH] of {byte, bit, chan};
   temp_chan ! this;
   do
     :: atomic { this ? buffer_put, a, ret ->      /* method put */
        if
          :: (count < SIZE) -> data[(front + count) % SIZE] = a;
           count ++
          :: ! (count < SIZE) -> this ! buffer_put, a, ret
        fi
        }
     :: atomic { this ? buffer_get, a, ret ->      /* method get */
        if
          :: (count> 0) -> ret ! consumer_retrieved, data[front];
                           front = (front + 1) % SIZE; count --
          :: ! (count > 0 ) -> this ! buffer_get, a, ret
        fi
        }
   od
}
proctype producer (chan temp_chan; chan a) {  /* actor producer */
   bit next; chan buf;
   chan dummy;
   chan this = [LENGTH] of {byte};
   buf = a;
   this ! producer_putting;
   temp_chan ! this;
   do
     :: atomic { this ? producer_putting ->  /* method putting */
           buf ! buffer_put, next, dummy ;
           next = ! next;
           this ! producer_putting
        }
   od
}

proctype consumer (chan temp_chan; chan a)   /* actor consumer */
   chan buf;
   bit b;
   bit dummy;
   chan this = [LENGTH] of {byte, bit};
   buf = a;
   this ! consumer_getting, dummy;
   temp_chan ! this;
```

```
    do
      :: atomic { this ? consumer_getting, dummy ->
                                        /* method getting */
           buf ! buffer_get, dummy, this
         }
      :: atomic { this ? consumer_retrieved, b ->
                                        /* method retrieved */
           printf("read num = %d\n", b);
           this ! consumer_getting, dummy
         }
    od
}

init {
    chan buf;
    chan proc;
    chan cons;
    chan temp_chan = [0] of {chan};

    atomic{                                  /* create buffer */
      run buffer (temp_chan);
      temp_chan ? buf
    }
    atomic{                                  /* create producer */
      run consumer (temp_chan, buf);
      temp_chan ? cons
    }
    atomic {                                 /* create consumer */
      run producer (temp_chan, buf);
      temp_chan ? proc
    }
}
```

6. UML 명세에서의 변환과 검증

이 장에서는 UML로 모델링 된 시스템의 요구사항 검증을 수행하는 방법을 제시한다. 이는 UML의 여러 다이어그램에서 정형적 명세를 추출하는 작업으로부터 시작된다. UML로 기술할 수 있는 대상 영역들이 넓고, UML 자체의 의미가 정형적으로 정의되어 있지 않기 때문에, UML로 작성된 모델들은 대개의 경우 그 의미가 불확실하여 바로 정형적 검정의 대상이 될 수 없다. 따라서 정형적 의미가 정의될 수 있는 UML 언어의 부분집합을 명확히 정의하고, 주어진 UML 모델이 있을때 거기서 불명확한 부분을 지적, 보충한 후 정형적 모델로 변환하는 것이 필요하다.

본 논문에서 제시하고자 하는 검증에 관한 사항은 주로 병렬성(concurrency)에 대한 것이다. 그러므로 UML로 기술된 모델이 가지는 병렬적 의미를 명확히 정의하는 것이 필요하다. UML 자체에는 그 모델이 가지는 병렬적 의미에 대해 명확히 정의되어 있지 않으므로, 먼저 이것을 정의하여야 한다. 여기서는 Actor 모델에 기반하여 동적 모델을 해석하는 의미론을 제시한다. 그리고 UML 모델이 그렇게 해석되기 위해 필요한 제한 조건들을 정의한다.

UML의 여러 다이어그램 중에서 검증을 위해 사용되는 다이어그램은 클래스 다이어그램, 상태 다이어그램, 순차(sequence) 다이어그램이다. 클래스 다이어그램은 시스템의 정적인 구조와 클래스로 부터 생성되는 객체들의 인터페이스를 정의하고, 객체들간의 상호 관계를 표현하고 있다. 이들은 APromela에서 클래스의 정의 및 변수, 메소드 선언부로 대응되며, 클래스 간의 관계는 검증하고자 하는 속성을 기술할 때 중요한 역할을 하게 된다. 상태 다이어그램은 각 객체들의 동적인 상태 변화를 기술한다. 마지막으로 순차 다이어그램은 객체 간의 상호 메세지 교환을 구체적으로 기술하므로 이를 이용하여 검증하고자 하는 속성을 기술할 수 있으며, 이를 이용하여 변환된 APromela 모델을 이용하여 검증을 수행할 수 있다.

6.1 UML 다이어그램의 병렬적 의미

본 방법론이 대상으로 하고 있는 시스템은 병렬 객체지향 시스템이다. [11]에서는 UML 모델의 병렬 해석에 대해 다음과 같은 사항을 지적하고 있다.

- 순차 모델과 병렬 모델의 큰 차이는 수행 단위인 스레드(thread)가, 시스템 내에 한개 존재하는가 여러개 존재하는가의 차이이다.
- 다중스레드 모델을 객체지향과 결합하는 방법은, 명시적(explicit)으로 스레드를 사용하는 방법과, 묵시적(implicit)으로 객체와 스레드를 결합하는 방법이 있다. UML은 묵시적 모델에 더 잘 어울린다.
- 묵지적 모델에서 병렬성의 단위는 활성객체(active object) 이다. 하나의 활성객체는 하나의 스레드를 가진다. 비활성객체(passive object) 는 일반적인 순차적 객체로, 자신의 스레드를 갖지 않고, 메시지를 받을때 스레드를 함께 받아 수행하고, 종료와 함께 스레드를 반환한다.
- 묵시적 모델을 사용할 경우 개발의 초기단계인 분석단계에서는 모든 객체는 활성객체로 취급된다. 이후 불필요한 병렬성을 줄이는 일이 설계과정에서 수행된다.
- 모델링 과정에서는 활성객체 간의 통신은 이벤트와 메시지에 의해 기술된다.

이상과 같은 사항을 바탕으로, 여기서는 활성객체들로 구성되어 있고, 각각은 논리적으로 분산되어 있는 시스

템을 대상으로 한다. 객체들은 서로간에 메시지를 전달함으로써 통신을 하고, 메시지들은 각 객체의 상태를 바꾸거나 다른 메시지를 보내는 등의 일을 일으킨다. 객체들간의 동기화는 메소드들 단위로 이루어 진다. 이는 Actor 모델이라는 병렬 객체 모델과 유사하다. 따라서 UML 모델을, Actor 모델을 기본으로 하는 APromela라는 언어로 변환함으로써 병렬 객체 모델을 모델체킹으로 검증할 수 있다.

요약하면 본 절에서 제시하는 검증방법은 다음과 같은 UML 모델에 적용될 수 있다. 이 조건이 만족되지 않는 모델에 대해서는 기술되지 않은 부분을 채우거나 동작을 추상화함으로써 이러한 모델로 변환하여 검증할 수 있다. 이러한 정형화 작업은 어떤 정형적 검증 방법을 사용하더라도 우선적으로 수행되어야 할 과정이다.

1. 클래스 다이어그램의 조건
   - 모든 객체는 활성객체 (active object) 이다.
   - 클래스 변수는 사용되지 않는다.
   - 연관 관계는 유한한 복수성(multiplicity)를 가진다.
2. 상태 다이어그램
   - 검증의 대상이 되는 시스템을 이루는 모든 객체들에 대해 상태 다이어그램으로 동작이 명세되어 있어야 한다.
   - 각 객체가 받아들이는 메시지는 그 객체의 메소드에 대응한다.

6.2 클래스 다이어그램(class diagram)의 변환

UML에서 정의된 클래스 다이어그램은 각 클래스의 이름, 속성, 메소드를 정의하며, 클래스 간의 여러가지 관계를 표현할 수 있다. 이를 다음과 같은 방법을 이용하여 APromela로 변환한다.

6.2.1 클래스의 정의 부분의 변환

클래스 다이어그램은 클래스의 이름과 속성(attribute), 메소드들을 정의한다. 이 중에 이름과 타입, 인수 등은 Apromela에서 그대로 표현할 수 있다. 가시성에 대한 정보는 무시되고, 클래스 변수는 사용하지 않는다. 메소드에 대해서는, 여기서 선언되지만 실제 동작은 상태 다이어그램에서 정의되므로, 이 단계에선 무시된다.

클래스의 속성은 bit, byte, bool, short, int로 제한되며, 다른 타입에 대해서는 적절한 타입으로 변형해야 한다. 예를 들어 string과 같은 경우에는 byte의 배열로 선언하거나, real을 int로 단순화하는 등의 일을 말한다. 모델체킹의 검증시 복잡도는 상태공간의 크기에 크게 좌우되므로, 가능한 작은 범위의 변수들을 사용하도록 한다. 따라서 추상화 하거나, 가능하면 검증시 사용되지 않는 변수 (즉, 상태 다이어그램이나 검증하려는 성질에 관계없는 속성) 들은 생략한다.

6.2.2 연관 관계(association relationship)의 변환

클래스간의 연관 관계는 여러 가지 요소로 구성될 수 있다. 하나의 연관 관계는 이름, 방향성, 복수성(multiplicity)을 가지게 된다. 이를 APromela로 변환하기 위해서는 우선 방향성과 복수성이 명시적으로 정해져야 한다. 또한 모델체킹은 유한한 상태공간을 가지는 시스템에 적용되므로, 복수성은 제한된 값으로 한정시키는 것이 좋다. (그렇게 하지 않고, 검증할때 사용할 수 있는 최대값을 검증기가 묵시적으로 지정하도록 할 수도 있다.)

이렇게 방향성과 복수성, 이름이 정해지면, 연관관계는 한 클래스내에서 다른 객체를 참조하는 객체 참조형 변수의 형태로 기술되게 된다.

- To-One 관계: 이 경우에는 단순히 객체 참조형 변수를 선언한다. 만약 B가 A와 연관된 클래스이며, B의 역할 이름이 role일 때 클래스 A내에

  actor role;

  와 같이 선언한다. role 의 타입이 클래스 B 라는 정보는 실제로 role 객체를 생성할때 기술하게 된다. 이는

  role = create B ();

  와 같이 기술된다. 원래의 UML 모델이 별도로 객체 생성에 대한 정보를 가지고 있지 않을 경우, 각각의 객체는 main 부분에서 초기에 생성된다고 본다.
- To-Many 관계: APromela 에서 to-many 관계는 기본적으로 배열을 이용하여 기술한다. 복수성이 명시되어 있는 경우에는 해당하는 만큼의 배열을 선언해 주고, * 의 경우에는 합리적인 최대값으로 선언한다.
- Ordered To-Many 관계: 이 경우에는 배열을 이용하여 To-many 관계로 변환한다.

집합 관계(aggregation relationship)의 경우는 연관 관계의 하나로 볼수 있으며 각 클래스가 부분과 전체의 개념을 갖는 경우이다. 하나의 클래스가 다른 클래스에 포함되거나 일부분으로 속할 수 있는 경우에 사용하며 연관 관계의 한 형태로 볼 수 있으므로 복수성을 고려하여 같은 방법으로 APromela로 변환할 수 있다. 이 경우에는 M:N이나 1:N의 관계를 이루는 경우가 주로 많게 된다. 또한, 하나의 객체에 의해 다른 객체가 생성

되는 경우가 많으므로, 의미에 맞게 객체 생성 (create)을 해준다.

6.3 상태 다이어그램(state diagram)

상태 다이어그램은 객체가 가질 수 있는 상태를 기술하고, 각 상태에서의 동작을 설명하며, 다른 상태로의 전이를 일으키는 이벤트들을 기술하게 된다. 이것은 실제 시스템내의 객체들의 동적인 행위를 나타낸다. 이 절에서는 이러한 상태 다이어그램을 이용하여 APromela로 시스템을 모델링하는 방법을 설명하고자 한다.

6.3.1 상태(state)와 전이(transition)의 변환

하나의 상태는 상태명, 상태 변수의 치환부, 행위(activity) 기술부로 구성된다. 상태명은 idle, paid, moving과 같이 현재 상태를 적절히 나타내는 이름으로 지정하게 되고, 상태 변수의 치환부는 그 상태에서 행해지는 변수값의 치환을 나타낸다. 마지막으로 행위 기술부에서는 그 상태에 도달한 시점부터 그 상태를 빠져나가는 시점까지의 객체의 행위를 기술하게 된다.

UML에서 각 전이는

event-signature [guard-condition] / action-expression

과 같이 기술할 수 있다. event-signature의 이벤트가 발생하고 guard-condition이 만족할 경우에만 상태의 전이가 실제로 발생하게 된다.

객체간의 통신을 담당하는 메세지 전달 방법은 여러가지 형태로 구현될 수 있는데, APromela로 모델링할 경우에는 비동기화 메세지 전달 방법을 이용하는 것으로 가정한다. 즉 객체들은 각각 mailbox (message queue)가 있어서, 한 객체에서 다른 객체로 보내는 메시지는 비동기적으로 처리되게 된다. 이는 병렬 시스템에 올리는 통신방법이다. 이때, 상태 다이어그램의 이벤트는 메소드 호출을 의미하는 것으로 메시지의 도착이나 메시지 전송에 대응되게 된다.

이상과 같은 상태와 전이의 해석을 가지고, 다음과 같은 방법으로 한 클래스의 상태 다이어그램을, 그 클래스에 대한 APromela 명세로 바꿀 수 있다.

1. 그 객체가 가지는 메소드들은 그 객체가 가지는 전이들에 기술된 events의 집합이다. 이는 클래스 다이어그램에서 선언된 메소드들의 부분집합이어야 한다. 클래스 다이어그램에는 선언되어 있지만 상태 다이어그램에 나타나지 않는 메소드들은 APromela 명세로의 변환과정에서 생략된다. 이는 그 메소드의 동작이 UML 모델에 명시되지 않았기 때문이다.
2. 클래스 다이어그램에서 정의된 객체의 속성에 추가하여, 상태를 나타내는 속성을 첨가한다. 이는 byte 타입의 변수로 정의되는데, 이 변수가 가질 수 있는 값, 즉 상태 이름은 APromela의 mtype 명령을 이용하여 정의할 수 있다.
3. 각 메소드의 바디 부분에서는, 현재 상태를 나타내는 변수와 guard condition 을 검사하여 참일때만, 해당하는 action expression과 상태전이를 수행하도록 기술한다. 이는 if - fi 문으로 기술된다.
4. 행위 기술부의 행위에서 다른 이벤트를 발생시키는 경우에는 대응되는 객체로의 메시지 전송으로 기술한다.
5. 상태에 기술된 상태변수 치환부는 그 상태가 가지는 의미에 대한 주석과 같은 역할을 하므로 변환과정에서는 생략된다.
6. 상태에 기술된 entry, exit action에 대해서는 그것들을 일으킬 수 있는 메소드 부분에서 기술한다.
7. 자연어로 기술된 행위에 대해서는 APromela의 여러가지 연산자를 이용하여 모델링한다.

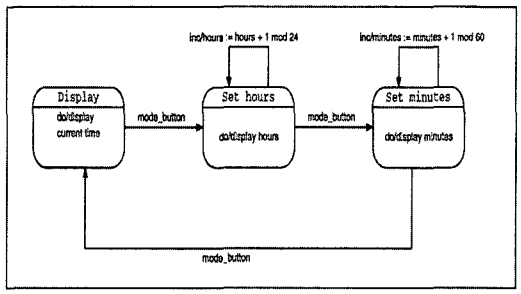

그림 11 디지털 시계의 상태 다이어그램

이상과 같은 과정을 바탕으로, 그림 11을 다음과 같은 형태로 변형할 수 있다.

```
mtype = { Display, SetHours, SetMinutes }
actortype Watch {
  var
    byte state;
    int ours, minutes
  method mode_button()
    if
    :: state == Display -> { state = SetHours }
    :: state == SetHours -> { state = SetMinutes }
    :: state == SetMinutes -> { state = Display }
    fi
  method inc()
    if
```

```
    :: state == SetHours -> { hours= (hours+ 1) % 24 }
    :: state == SetMinutes -> { minutes= (minutes+ 1) % 60 }
    fi
}
```

6.3.2 상태 다이어그램의 계층 구조 변환

상태 다이어그램은 계층 구조로 표현될 수 있다. 계층 구조를 위해서 *and*-상태와 *or*-상태를 지원하며, and-state의 경우 병렬로 수행되는 두개 이상의 하위 상태 구조를 갖게 된다. or-상태는 하나의 상태 내에 여러 개의 상태를 갖게 되며 한 순간에 그 중 하나의 상태에 있다는 것을 의미한다. and-상태는 병렬적으로 수행되거나 혹은 비결정적으로 수행되는 경우를 표현하기 위해 주로 사용되며, or-상태는 and-상태내에 포함되어 병렬적으로 수행되는 각각의 수행 흐름을 표현하거나, 각각의 작업을 모델링하는 데 있어서 표현을 쉽고 용이하게 해 주는 기능을 한다.

계층 구조가 포함된 경우를 APromela로 변환하기 위해서는 다음과 같은 작업을 수행한다.

- or-상태의 경우에는 각각의 상태를 하나의 상태로 보고 변환한다. or-상태의 경우에는 여러개의 상태에 대해 하나의 상위 상태를 지정한 경우이므로 각각을 다른 상태로 보고 변환하는 것이 가능하다.
- and-상태로 구분된 상태 구조들을 펼쳐서 (unfolding) and-상태가 없는 상태 그래프로 바꾼다.

6.4 순차 다이어그램(sequence diagram)

순차 다이어그램에서는 객체들간의 상호 작용을 표현하며, 특히, 시간의 흐름에 따른 메세지의 교환을 나타낸다. 일반적으로는 가로축에 객체들의 이름을 나열하며, 세로축은 시간의 흐름을 의미하게 되고, 객체 간에 상호 교환되는 메세지들을 화살표를 이용하여 표현하게 된다. 순차 다이어그램은 일반적(generic)이거나 개별적(instance) 일 수 있는데, 여기서는 개별적 순차 다이어그램만을 다룬다. 이는 조건이나 분기, 반복 등이 없는, 객체 간의 하나의 특별한 시나리오를 기술한다.

APromela에서는 객체의 이름이 정해져 있으므로, 순차 다이어그램상에 표기된 객체에는 받을 수 있는 모든 메세지들이 메소드의 형태로 기술되어 있어야 한다. 메세지에 대한 조건과 전달 순서, 동기화된 메세지 전달은 APromela를 통해 모델링된 코드와 일치해야 한다. 따라서, 순차 다이어그램을 이용하여, 검증하고자 하는 속성을 기술하는 데 사용할 수 있다.

예를 들어 그림 12를 보면 Print(ps-file)이라는 요청이 들어오면 큐에 아무것도 없을 경우 바로 출력을 하고 Success라는 응답을 보내오는 과정을 나타내고 있다. 따라서, 이러한 다이어그램으로부터 위와 같은 요구사항을 찾아낼수 있으며, 이를 시제 논리를 이용하여 기술할 수 있다. 위와 같은 경우에는

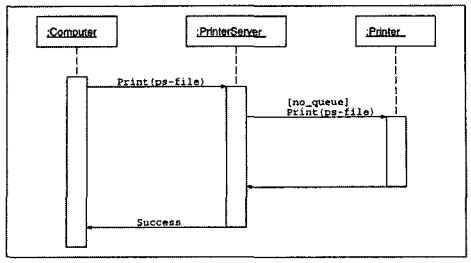

그림 12 프린터 출력 시나리오의 순차 다이어그램

$$PrinterServer.print \wedge (PrinterServer.no\_queue = true) \rightarrow \Diamond (Printer.print \wedge \Diamond Computer.Success)$$

와 같이 기술 할 수 있다. 이는 Computer 객체에서 PrinterServer 객체로 Print라는 요청을 보내고 그때 no_queue의 값이 참이면, Printer 객체로 Print 요청을 보내고 다시 Computer 객체로 Success 메시지를 보낸다는 의미이다.

위에서 보듯이 이 방법은 순차 다이어그램을, 시스템이 만족해야 할 요구사항을 그림으로 나타낸 것으로 본다. 반면 순차 다이어그램의 원래 목적이 명확한 동작 기술이 아니라, 개발자의 이해를 돕기 위한 것인 만큼, 여러가지 해석이 있을 수 있게 된다. 따라서 검증시에 순차 다이어그램을 참조하여 요구사항을 정형적으로 기술할때, 다시 정확한 해석이 필요하다.

요구사항을 순차 다이어그램을 거치지 않고, 바로 시제 논리로 기술하는 것도 가능하다. 또한 데드락 등의 일반적인 성질에 대해서는 모델 체커 자체에서 지원해 주기 때문에 특별히 성질을 따로 기술하지 않아도 검증 가능하다.

7. 결 론

우리는 SPIN 모델체커가 지원하는 Promela 언어를 병렬 객체지향으로 확장한 언어인 APromela를 제안하였다. 이 언어는 검증하려는 시스템을 객체지향적인 방법으로 모델링할 수 있게 해 준다. 만들어진 모델은 변환기를 거쳐 모델체커의 입력으로 들어가 검증이 수행되게 된다. 검증 과정중에 발견되는 에러들은 논리적 에러의 디버깅에 유용한 정보로 기능하게 된다. 이 언어를 이용하는 예로서, 바로 모델링 언어로 사용하는 경우와,

미리 작성된 UML 명세의 검증에 사용하는 경우, 두가지를 제시했다.

이 방법의 한계로 우선 지적할 수 있는 것은, 이 언어가 지원하는 객체지향적 특징이 완전하지 않다는 점이다. 일반적으로 한 언어가 객체지향적이라는 것은, 객체를 기반으로 하는 추상화(encapsulation)을 지원하고, 그 객체들은 클래스(class)를 기반으로 하여 생성(instantiation)되며, 클래스들은 상속(inheritance) 관계에 의해 계층화되는 것을 뜻한다. 이중에서 문제가 되는 것은 상속이다. 제안된 언어는 상속의 개념을 지원하지 않는다. 이런 경우 객체지향이란 용어 대신에 객체기반(object-based) 라는 용어를 사용하는 경우도 있다. 하지만, 일반적으로는 객체지향이란 용어를, 두가지 경우 모두 포함하는 것으로 보는 것이 보통이다. 또한 이 논문의 기반이 되는 Actor 모델 역시 상속을 지원하지 않지만, 일반적으로 객체지향 모델로서 간주되고 있다[24].

추가적인 연구방향으로는 다음과 같은 것들을 생각할 수 있다. 우선 직접 모델링하는 것 외에도, 이 언어는 액터 모델에 기반한 병렬 객체지향 프로그래밍 언어들의 병렬성 검증에도 이용될 수 있다. 이들은 거의 유사한 병렬성 지원 구조들을 가지므로, 코드에서 병렬성에 관한 부분을 추출, APromela로 변환함으로써, 코드의 검증을 수행할 수 있다. 또한, 데드락 이외의 다른 성질에 대한 검증에 관한 문제들도 있다. SPIN은 시제논리(temporal logic)에 기반한 성질 기술과 검증을 지원한다. 이때, APromela에서 기술해 줄 수 있는 시제논리식을 어떻게 자동적으로 Promela로 바꿀 수 있는지에 대한 연구가 필요할 것이다.

참 고 문 헌

[1] R. J. Anderson, P. Beame, S. Burns, W. Chan, F. Modugno, D. Notkin, and J. D. Reese, "Model Checking Large Software Specifications," in Proc. the Forth ACM SIGSOFT Symposium on the Foundation of SE, pp. 156-166, 1996.

[2] W. Athas and C. Seitz, "Cantor User Report Version 2.0," Technical Report 5232:TR86, California Institute of Technology, Pasadena, CA, Jan. 1987.

[3] G. Agha, S. Frolund, W. Kim, R. Panwar, A. Patterson, and D. Sturman, "Abstraction and Modularity Mechanisms for Concurrent Computing," in IEEE Parallel and Distributed Technology: Systems and Applications, 1(2):3-14, May 1993.

[4] G. Agha, "Actors: A Model of Concurrent Computation in Distributed Systems," MIT Press, 1986.

[5] R. Alur, T.H. Henzinger, and P.H. Ho, "Automatic Symbolic Verification of Embedded Systems," IEEE Transactions of Software Engineering, 22(3): 181-201, 1996.

[6] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang, "Symbolic model checking : 10^{20} states and beyond," Information and Computation, 98(2):142-171, 1992.

[7] P. Cousot and R. Cousot, "Abstract interpretation frameworks," in Journal of Logic and Computation, 2(4):511-547, Aug., 1992.

[8] E.M. Clarke, E.A. Emerson, and A.P. Sisla, "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications," ACM Transactions on Programming Languages and Systems, 8(2):244-263, 1986.

[9] J. C. Corbett, "Constructing Compact Models of Concurrent Java Programs," in Proc. ISSTA 98, 1998.

[10] G. Duval, "Specification and Verification of an Object Request Broker," in Proc. ICSE 98, 1998.

[11] H. Eriksson and M. Penker, UML Toolkit, John Willey and Sons, Inc., 1998.

[12] S. Frolund and G. Agha, "A Language Framework for Multi-Object Coordination," in Lecture Notes in Computer Science 627, 1993.

[13] P. Godefroid, "Partial-Order Methods for the Verification of Concurrent Systems," Lecture Notes in Computer Science 1032, Springer-Verlag, 1996.

[14] C. Hewitt, "Viewing Control Structures as Patterns of Passing Messages," in Journal of Artificial Intelligence, 8(3):323-364, 1977.

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

[16] G. Holzmann, "The Model Checker SPIN," IEEE Transactions on Software Engineering, Vol. 23, No. 5, pp. 279-295, May 1997.

[17] W. Kim, "THAL: An Actor Ststem for Efficient and Scalable Concurrent Computing," Ph.D Thesis, Univ. of Illinois at Urbana-Champaign, 1997.

[18] D. Kafura, M. Mukherji, and G. Lavender, "ACT++: A Class Library for Concurrent Programming in C++ Using Actors," in Jounal of Object-Oriented Programming, Oct. 1993.

[19] H. Lieberman, "Concurrent Object-Oriented Programming in ACT1," in Object-Oriented Concurrent programming, MIT Press, Cambridge, MA, 1987.

[20] E. Najm and F. Olsen, "Reactive EFSMs, Reactive PROMELA/RSPIN," in Proc. Tools and Algorithms for the Construction and Analysis of Systems, (TACA96), pp. 349-368, LNCS 1055, Springer-Verlag, Mar. 1996.

[21] M. Staskauskas, "Tales from the Front: Industrial Experience with Formal Validation," in Proc. First SPIN Workshop, INRS Quebec, Canada, Oct. 1995.
[22] S. Tripakis and C. Courcoubetis, "Extending PROMELA and SPIN for real-time," in Proc. Tools and Algorithms for the Construction and Analysis of Systems, (TACA96), pp. 329-348, LNCS 1055, Springer-Verlag, Mar. 1996.
[23] C. Weise, "An incremental formal semantics for PROMELA," in Proc. Third Spin workshop, Apr., 1997.
[24] P. Wegner, "Concepts and Paradigms of Object-Oriented Programming," {\it ACM OOPS Massenger}, Aug., 1990.
[25] A. Yonezawa, "ABCL An Object-Oriented Concurrent Systems," MIT Press, Cambridge, Mass., 1990.

조 승 모

1990년 ~ 1994년 한국과학기술원 전산학과(학사). 1994년 ~ 1996년 한국과학기술원 전산학과(석사). 1996년 ~ 현재 한국과학기술원 전산학과 박사과정 재학중. 관심분야는 정형기법, 실시간시스템, 병렬객체지향시스템 명세와 검증

김 영 곤

1983년 경북대학교 전자공학과 졸업(공학사). 1985년 연세대학교 대학원 전자공학과 졸업(공학석사). 1985년 ~ 현재 한국전기통신공사 멀티미디어연구소 선임연구원. 1996년 ~ 현재 한국과학기술원 전산학과 박사과정 재학중. 관심분야는 분산멀티미디어 컴퓨팅, 객체지향 개발방법론,테스팅등

배 두 환

1980년 서울대학교 조선공학과 학사. 1987년 위스콘신-밀워키대학 전산학 석사. 1992년 플로리다 대학 전산학 박사. 1992년 ~ 1994년 플로리다 대학 전산학과 조교수. 1995년 ~ 1996년 한국과학기술원 정보 및 통신공학과 조교수. 1996년 ~ 현재 한국과학기술원 전산학과 조교수.

변 성 원

1986년 숭실대학교 전산학과 졸업(공학사). 1989년 한국과학기술원 전산학과 졸업(공학석사). 1989년 ~ 1992년 삼보컴퓨터 기술연구소 근무. 1993년 ~ 현재 한국전기통신공사 멀티미디어연구소 선임연구원. 관심분야는 Direct Assistance Service, 대용량 DBMS Solution, Network 설계, 관리 및 보안

김 상 택

1980년 고려대학교 전자공학과 졸업(공학사). 1985년 고려대학교 대학원 전자공학과 졸업(공학석사). 1980년 ~ 1983년 한국전기통신연구소(KETRI) 근무. 1984년 ~ 현재 한국전기통신공사 멀티미디어연구소 영상미디어 연구실장. 관심 분야는 전자상거래, 영상정보처리