특집 02

정형논리를 이용한 모델링 및 분석

목 차

1. 서 론
2. 정형 논리
3. 병행 소프트웨어 검증
4. 결 론

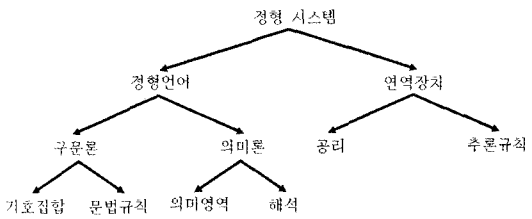
박사천 · 권기현
(경기대학교)

1. 서 론

컴퓨팅 환경이 유비쿼터스 및 임베디드와 같이 복잡한 환경으로 변화되면서 소프트웨어 검증이 더욱 중요해지고 있다. 테스팅이나 시뮬레이션과 같은 전통적인 검증 기법으로는 복잡한 환경에서 동작하는 소프트웨어를 철저하게 검증할 수 없기 때문에, 철저한 소프트웨어 검증을 위해서 많은 연구자들이 정형 논리 활용에 관심을 기울이고 있다. 그래서 본 기고에서는 정형 논리의 개념 및 소프트웨어 검증 활용 예를 살펴본다.

정형 논리는 일종의 정형 시스템이기 때문에, 정형 시스템을 먼저 살펴본다. 정형 시스템은 모델링 및 분석을 정형화한 것으로서 정형 언어와 연역 장치로 구성된다. 정형 언어는 모델링에 사용될 언어의 구문론 및 의미론을 정의한다. 구문론의 역할은 언어를 정의하는 것이다. 이를 위해서 구문론은 사용될 기호들의 모임을 정의한 기호 집합과 이들 기호들을 나열하여 올바른 문자열을 생성하는 문법 규칙으로 구성된다. 문법 규칙에 의해 생성될 수 있는 모든 문자열 집합을 언어라고 부른다. 의미론의 역할은 생성된 문자열에 값을 배정하는 것이다. 문자열이 가질 수 있는 값의 범위를 의미 영역이라 부르며, 주어진 해석 하에서 생성된 문자열의 값을 결정할 수 있다.

한편, 정형 시스템을 구성하는 또 다른 요소인 연역 장치는 문자열을 조작해서 새로운 문자열을 만들어내는 장치이다. 이를 위해서 연역 장치는 공리와 추론 규칙으로 구성된다. 공리로부터 시작해서 추론 규칙을 반복 적용하여 새로운 문자열을 만들어낸다. 지금까지 설명한 정형 시스템의 구성 요소를 요약하면 다음 그림과 같다.



(그림 1) 정형 시스템의 구성 요소

2. 정형 논리

다양한 정형 논리가 전산학에서 활용되고 있다. 대표적인 것으로는 명제 논리, 술어 논리, 그

리고 시제 논리이다. 이들 논리는 상호 간에 장점 및 단점을 갖는다. 명제 논리는 기계 처리가 쉬운 반면에 표현력이 낮다. 한편, 술어 논리 및 시제 논리는 표현력이 높은 반면에 기계 처리가 어렵다. 어떤 논리를 선택할 것인가는 응용에 좌우된다.

여기서는 명제 논리를 살펴본다. 명제 논리를 선택한데에는 두 가지 이유가 있다. 첫째, 기계 처리가 쉬워서 명제 논리를 다루는 우수한 도구들이 많이 존재하기 때문이다. 둘째, 시제 논리나 술어 논리는 명제 논리로 간주해서 처리할 수 있기 때문이다.

명제 논리는 정형 시스템이기 때문에, 전 장에서 살펴본 정형 시스템의 구성 요소 위주로 명제 논리를 설명한다. 다음은 명제 논리에서 사용되는 기호 집합이다.

$$\{\top, \bot, p, q, r, \cdots, \neg, \vee, \wedge, \Rightarrow, \Leftrightarrow, (,)\}$$

여기서 $\top$는 항상 참을 나타내며, $\bot$는 항상 거짓을 나타내는 상수이다. 그리고 $p,q,r,...$은 명제 기호로서 더 이상 분해할 수 없는 단순 명제를 나타낸다. 그리고 명제 기호를 결합해서 보다 복잡한 논리식을 형성하기 위해 $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$ 연산자가 사용된다. 또한 복합한 논리식을 쉽게 구분하기 위해서 괄호를 사용한다. 위의 기호들을 나열해서 올바른 명제 논리식을 형성하는 문법 규칙은 다음과 같다.

$$\phi, \psi ::= \top \mid \bot \mid p \mid (\neg\phi) \mid (\phi \vee \psi) \mid (\phi \wedge \psi) \mid (\phi \Rightarrow \psi) \mid (\phi \Leftrightarrow \psi)$$

위의 문법 규칙에 생성된 모든 논리식의 집합이 명제 논리 언어 PL이다.

의미론은 생성된 논리식에 값을 부여한다. 논리식이 취할 수 있는 값 집합을 의미 영역이라 부른다. 의미 영역으로는 {1, 0}이다. 즉, 모든 논리식은 참(1) 또는 거짓(0) 값을 갖는다. 명제 논리식에 참 또는 거짓 값을 배정하기 위해서는 명제 기호에 대한 해석이 주어져야 한다. 그래서 해석은

$$\sigma : AP \rightarrow \{1, 0\}$$

이다. 여기서 AP는 논리식에 사용된 명제 기호 집합이다. 해석이 주어지면, 논리식의 값을 결정할 수 있다. 만일 어떤 해석 σ하에서 논리식 ϕ값이 참이라면

$$\sigma \models \phi$$

로 표현한다. 이것을 만족성 관계라고 부른다. 이러한 관계를 이용해서 논리식의 의미를 재귀적으로 정의하면 다음과 같다.

$\sigma \models \top$

$\sigma \not\models \bot$

$\sigma \models p$ iff $\sigma(p) = 1$

$\sigma \models \neg\phi$ iff $\sigma \not\models \phi$

$\sigma \models \phi \wedge \psi$ iff $\sigma \models \phi$ and $\sigma \models \psi$

$\sigma \models \phi \vee \psi$ iff $\sigma \models \phi$ or $\sigma \models \psi$

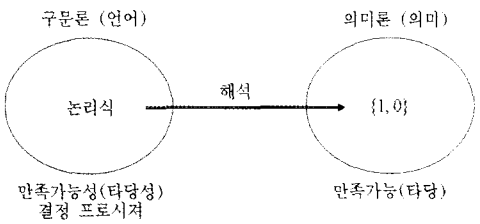
$\sigma \models \phi \Rightarrow \psi$ iff $\sigma \not\models \phi$ or $\sigma \models \psi$

$\sigma \models \phi \Leftrightarrow \psi$ iff $\sigma \models \phi \Rightarrow \psi$ and $\sigma \models \psi \Rightarrow \phi$

의미론 입장에서 볼 때 모든 논리식은 세 가지 그룹으로 분류된다. 첫째는 '만족가능'한 논리식으로서 논리식을 참이 되게 하는 해석이 존재하는 경우이다. 둘째는 '타당'한 논리식으로서 논리식의 값이 모든 해석 하에서 참이 되는 경우이다. 셋째는 '만족불가능'한 논리식으로서 논리식의 값이 전혀 참이 될 수 없는 경우이다.

주어진 논리식이 어느 그룹에 속하는지를 결정하는 것은 정형 논리에서 매우 중요하다. 명제 논리의 경우, 가장 간단한 방법은 진리표를 사용하는 것이다. 주어진 논리식에 대한 진리표를 구축한 이후에 각 행을 조사해 가면서 논리식에 맞는 그룹을 찾는다. 그러나 진리표는 의미론 세계에서 사용하는 도구로서, 모든 해석을 다 고려해야 하기 때문에 논리식의 크기가 큰 경우에 진리표 작업은 매우 힘들다.

한편, 연역 장치는 구문론 세계에서 사용되는



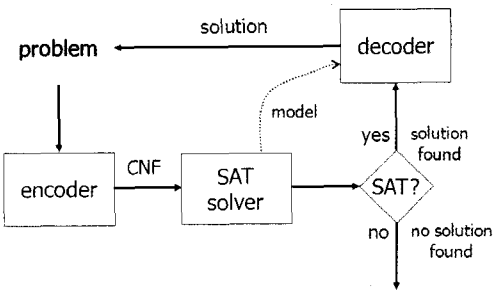
(그림 2) 명제 논리의 구성 요소

도구로서 의미를 고려하지 않고 오직 구문적인 조작으로 논리식을 처리하는 장치이다. 비록 의미를 고려하지 않고 오직 구문적으로만 처리하지만, 처리 결과는 의미론 세계에서 작업한 결과와 동일하다. 다시 말해서, 연역 장치를 이용해서 주어진 논리식이 속하는 그룹을 결정한 것은 진리표를 이용해서 판정한 결과와 동일하다. 주어진 논리식이 '타당'한 논리식인지를 결정하는데 사용되는 연역 장치를 '타당성 결정 프로시져' 또는 '정리 증명기'라고 부른다. 한편, 주어진 논리식이 '만족가능'한 논리식인지를 결정하는 것을 '만족가능성 결정 프로시져' 혹은 줄여서 'SAT 문제'라고 부른다. 그리고 SAT 문제에 사용되는 도구를 SAT 처리기라고 부른다.

SAT 문제는 이론 전산학에서 중요한 연구 주제이다. 왜냐하면 SAT 문제는 최초의 NP-complete 문제였기 때문이다. 뿐만 아니라 SAT 문제는 다양한 문제 풀이에 널리 활용되고 있다. 주어진 문제를 SAT 문제로 간주하여 해결하는 전형적인 흐름은 아래와 같다. 주어진 문제는 SAT 처리기의 입력 형식인 CNF(Conjunctive Normal Form)로 변환한다. 만약 SAT 처리기 실행 결과가 '만족가능'이면, 문제에 대한 해답이 존재하는 것이다. 이 경우 SAT 처리기가 찾아낸 모델이 문제를 풀 수 있는 답이다.

3. 병행 소프트웨어 검증

이번 장에서는 병행 소프트웨어 검증을 SAT 문제로 다루게 되는데, 병행성(concurrency)은 반응성과 처리량을 중요시 하는 여러 분야의 애플리케이션에서 빈번하게 사용되는 프로그래밍 기법이다. 또한 자바와 같이 널리 쓰이는 언어에서 자체로 병행 프로그래밍 구문을 내장하고 있다는 사실은 병행 프로그램의 작성이 더 이상 운영 체제나 내장형 실시간 애플리케이션을 개발하는 소수의 프로그래머에게로 국한되지 않음을 의미한다. 즉, 대부분의 프로그래머들이 안전성-위험(safety-critical) 시스템을 구현하는 일에 종사하는 것은 아니지만, 많은 프로그래머들이 비교적 덜 전문적인 분야에서 병행 프로그램을 사용하고 있다.



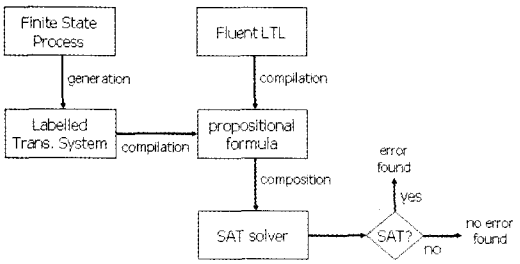
(그림 3) 문제 풀이를 SAT 문제로 간주

이런 분야의 애플리케이션이나 시스템에서 발생하는 에러가 비록 직접적으로 생명을 위협하는 것은 아닐지라도 삶의 질을 해하고 심각한 재정상의 문제를 야기할 수 있다. 그런데, 잘못된 병행 프로그램으로 인한 에러는 발생빈도가 매우 낮기 때문에 문제를 파악하는데 많은 시간이 소요된다. 그러므로 병행 프로그램은 작성하기도 어렵고 디버깅하기도 어려울 뿐만 아니라, 시뮬레이션이나 테스팅 같은 방법으로 에러를 찾아내기도 어렵다.

따라서 병행 프로그램은 사전에 정형적으로 모델링 되어야하고 자동 검증 도구를 이용해서 철저하게 분석해야 한다. 이렇게 병행 프로그래밍의 원리를 이해하고, 모델링하며, 분석을 통해서 시스템의 정확성을 확인하는 과정은 소프트

웨어를 전공하는 학생들이 배워야할 기본지식이고 소프트웨어공학 전문가들이 꼭 갖추어야 할 배경지식이다. 시스템을 철저하게 분석하는 정형 방법 중에 가장 많이 쓰이는 기법이 모델 체킹이다[2].

모델 체킹은 80년대 하드웨어 시스템 검증을 위해 고안되었는데 BDD(Binary Decision Diagram)를 내부 자료구조로 사용해서 크게 성능이 개선되었고 현재까지 널리 사용되고 있다. 그러나 모델의 크기에 비해서 검사해야할 상태 공간이 지수적으로 증가하는 상태폭발이라는 문제를 안고 있다. 이러한 상태폭발 문제를 극복하기 위한 적극적인 노력들이 있었고 상당한 성과를 이루었다. 근래에 상태폭발의 문제를 해결하기 위한 대안으로써 모델 체킹을 SAT 문제로 간주하는 연구들이 활발하게 진행되고 있다[3,4].



(그림 4) 병행 소프트웨어 검증을 SAT 문제로 간주

(그림 4)는 만족성 문제와 모델 체킹의 연관성을 설명한다. 소프트웨어의 모델을 표현하는 언어는 유한 상태 프로세스 *FSP*(Finite State Process)이다[1]. 모델에 대한 상태 공간은 *LTS*(Labeled Transition System)로 나타낸다. 프로그램에 있을법한 에러를 *FLTL*(Fluent Linear Temporal Logic)로 명세하고, 모델과 함께 명제 논리식으로 변환한다. 변환된 논리식은 SAT 처리기의 입력 형태인 CNF으로 변환되고, SAT 처리기는 프로그램에서 에러를 찾아준다[5,6]. 이번 장에서는 멀티 스레드 프로그램이 어떻게 모델링 되고, 인코딩 되어서 분석 되는지 그 과정을 보인다.

3.1 유한 상태 프로세스

유한 상태 프로세스는 1989년 밀러(Milner)에 의해서 제안된 *CCS*(Calculus of Communicating Systems)나 1985년 호어(Hoare)에 의해서 제안된 *CSP*(Communicating Sequential Processes)와 같은 일종의 프로세스 계산식(process calculus)이다. *FSP*는 발생되는 액션에 의해서 상태 전이가 일어나는데 병행 프로그램을 모델링하고 추론하기에 적합하다. 아래의 〈표 1〉은 *FSP* 구문을 요약한 것이다.

일반 프로그램 언어에서 사용되는 변수를 프로세스로 모델링하면, 다음과 같이 모델링 된다. const는 상수를 선언하는 구문이다. 즉 N을 2로 정의해서 사용하겠다는 뜻이다. range는 범위를 지정하는 명령어인데 T의 범위를 0부터 N까지 하겠다는 의미이다. VAR 프로세스에는 읽고 쓰는 액션이 기술되어 있는데, 각각 u와 v라는 내부 변수를 사용해서 0부터 2까지의 값을 쓰고 읽는 행위를 묘사할 수 있다.

```
const N = 2
range T = 0..N

VAR      = VAR[0],
VAR[u:T] = (read[u]   ->VAR[u]
           |write[v:T]->VAR[v]).
```

위의 프로세스를 (그림 5)와 같이 그래프로도 표현할 수 있는데, 이것은 곧, 전이 위에 액션이 레이블 된 *LTS*이다. 아래 그림은 상태들과 상태들의 관계 즉, 액션들로 레이블된 전이들로써 이루어져 있는데, 각 상태 안의 숫자는 단순히 상태를 구별하기 위함이다. 상태 0는 VAR[0]를 나타내고, 상태 1은 VAR[2]를 나타낸다. 이것은 프로그램의 변수가 각각 0의 값과 2의 값을 갖는다는 것을 모델링 한 것이다. 상태 2는 VAR[1]을 나타내는데, 모델링한 변수에 값 1이

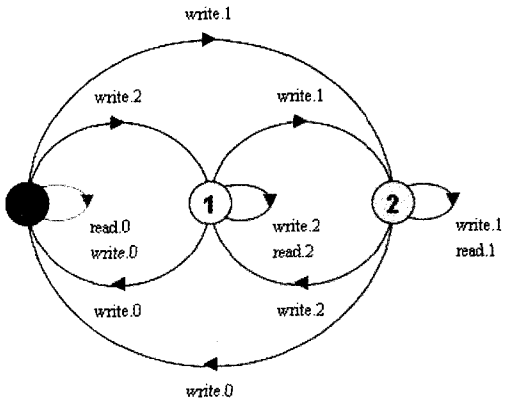
〈표 1〉 FSP 구문의 요약

기호	이름	설명
->	액션 접두사	만일 x가 액션이고 P가 프로세스라면 (x->P)는 액션 x가 발생 한 후에 프로세스 P에 해당하는 행위를 한다.
|	선택	만일 x와 y가 액션이라면 (x->P|y->Q)는 액션 x와 y 중 하나를 우선 수행하는 프로세스이다.
when	가드 액션	(When B x->P|y->Q)은 가드 B가 참이라면 액션 x와 y 둘 다 선택될 수 있지만, B가 거짓 이라면 액션 x는 선택되지 않음을 의미한다.
+	알파벳 확장	프로세스의 알파벳은 그 프로세스에서 사용되는 액션들의 집합이다. P+S 는 프로세스 P의 알파벳에 집합 S에 포함된 알파벳을 추가한다.
||	병렬 결합	만일 P와 Q가 프로세스라면 (P||Q) 는 P와 Q의 병행 실행을 나타낸다.
forall	반복자	forall[i:1..N] P(i)은 병렬 결합 (P(1)|| … ||P(N))이다.
:	프로세스 레이블	a:P는 P의 각 알파벳들의 접두어로 a를 붙인다.
::	프로세스 공유	{a1,…,ax}::P는 P의 알파벳으로 있는 모든 n을 $a_1.n$, …, $a_x.n$로 바꾼다. 즉, P의 모든 전이 (n->Q)의 정의가 ({a.n, …, ax.n }->Q)로 변경된다.
if then else	조건문	프로세스 if B then P else Q는 조건 B가 참이면 프로세스 P처럼 동작하고 그렇지 않으면 프로세스 P처럼 동작한다. 만일 else Q가 생략되고 B가 거짓이라면, 동작은 프로세스 STOP과 같이 된다.
/	리레이블링 (Re-labelin g)	리레이블링은 프로세스 내의 액션들의 이름을 변경한다. *{newlabel_1/oldlabel_1, …, newlabel_n/ old_label_n}*
@	인터페이스	프로세스 P에 인터페이스 연산자 @{a_1, …,a_x}를 적용하면 집합 {a_1,…,a_x}에 포함되지 않은 P의 알파벳들은 모두 숨겨진다.

들어 있다는 의미이다. 이 상태에서 1을 읽고 (read.1) 그 상태에 머물든지 0부터 2까지의 값을 쓰고(write.0, ...,write.2) 해당 상태로 이동하는 전이들이 가능하다. 이렇게 변수가 사용될 수 있는 범위를 미리 한정하기 때문에 유한(finite) 상태 프로세스라고 한다.

모델을 *FSP*와 같은 정형 언어로 만들면 정형적인 분석이 가능해 진다. 그런데, 정형 분석을 위해서 *FSP*를 직접사용 하기 보다는 기계적 계산이 용이한 상태로 변환해야 하는데 요컨대 위의 그림과 같은 *LTS*로 변환해서 분석할 수 있다[7-9]. 다음 절에서 *LTS*에 대한 정의를 알아본다.

3.2 상태공간의 표현



(그림 5) 프로세스 VAR에 대한 그래프 표현

앞 절에서는 *FSP*를 설명하고 간단한 모델링을 해보았다. 그런데 이러한 *FSP*의 의미는 레이블된 전이 시스템 *LTS*로 설명될 수 있다. *FSP*의 각 구문이 어떻게 *LTS*로 변환되는 지에 대한 자세한 내용은 [1]의 부록 C에서 참조할 수 있다.

정의 1. *LTS* M은 다음과 같이 네 개의 튜플 $\langle S, A, \Delta, q\rangle$로 구성된다:

- S : 상태들의 유한 집합,
- A : M에서 사용되는 액션들의 집합,
- $\Delta \subseteq S\times A \cup \tau\times S$: 상태가 액션을 만나 다음 상태로 전이하는 전이 관계. τ는 외부에서는 관찰 할 수 없는 내부 액션,
- $q \in S$: M의 초기 상태.

LTS $M = \langle S,A,\Delta,q\rangle$이 액션 a의 발생으로 *LTS* $M' = \langle S,A,\Delta,q'\rangle$으로 전이할 때, $M \xrightarrow{a} M'$ 표현하고, 이것은 $(q,a,q') \in \Delta$과 같다.

위 (그림 5)의 *LTS*를 정의 1의 표현으로 옮기면 다음과 같다. $S=\{0,1,2\}$, $A=\{read.0, read.1, read.2\}$, $\Delta=\{(0,read.0,0),(0,write.0,0)(0,write.1,1),(0,write.2,2),(1,read.1,1),(1,write.0,0),(1,wri$

te.1,1),(1,*write*.2,2),(2,*read*.2,2),(2,*write*.0,0),(2,*write*.1,1),(2,*write*.2,2)}, 마지막으로 $q=0$이다.

액션들이 연속적으로 발생함에 따라서 시스템이 동작하게 되는데 연속된 액션에 의해서 발생하는 전이들의 연속을 실행경로 혹은 경로라고 한다. LTS M에서 연속된 액션 $\langle a_1, a_2,...a_n\rangle$에 의한 전이의 연속을 σ라고 하자. $\sigma=<(s_1,a_1,s_2),(s_2,a_2,s_3),...,(s_n,a_n,s_{n+1})>$이다. 이때 $s_1=q$이고 $\forall 1\le i\le k.(s_i,a_i,s_{i+1})\in\Delta$이다. 여기서 경로의 길이 $|\sigma|$는 n이 된다. σ가 M의 경로라면, 경로의 i-번째 스텝은 (s_i,a_i,s_{i+1})이 된다.

정의 2. 어떤 상태 s_i에서 더 이상 전이가 발생하지 않는다면, 그 상태를 데드락 상태라고 하고, $|\sigma|=n$인 경로의 마지막 상태 s_{n+1}이 데드락 상태라면 σ를 데드락 경로라고 한다.

LTS집합 $\{M_1,...,M_n\}$이 있을 때, 이들의 병렬결합은 $M_1\|...\|M_n$로 표현한다. 병렬 결합된 LTS M에서 각각의 LTS들은 동일한 액션이 발생할 때에는 동시(synchronous)에 전이하고, 특정 LTS에만 레이블 된 액션이 발생할 때에는 그 LTS만 동작(interleave)하는 방식을 취한다. 따라서 병렬 결합된 LTS 상에서도 한 순간에 하나의 액션만 발생하게 된다.

정의 3. 병렬 결합된 LTS $M=M_1\|...\|M_n$은 $M_i=\langle S_i,A_i,\Delta_i,q_i\rangle$일 때, 다음과 같이 정의된다.

- $S=S_1\times...\times S_n$,
- $A=A_1\cup...\cup A_n$,
- $\Delta=\{([s_1,...,s_n],a,[t_1,...,t_n])\in S\times A-\{\tau\}\times S \mid \forall 1\le i\le n.(a\in A_i\Rightarrow(s_i,a,t_i))\vee(a\notin A_i\Rightarrow s_i=t_i))\}\cup\{([s_1,...,s_n],\tau,[t_1,...,t_n])\in S\times\{\tau\}\times S \mid \exists 1\le i\le n.(s_i,\tau,t_i)\in\Delta_i\wedge\forall 1\le j\le n.i\ne j\Rightarrow s_i=t_i\}$,
- $q=[q_1,...,q_n]$.

위의 정의에서처럼 n개의 LTS가 병렬 결합된 LTS의 상태는 n-튜플로 표현된다. 그러나 각 액션 집합들은 마치 프로그램의 전역 변수처럼 하나의 액션 집합으로 표현된다. 병렬 결합된 LTS 상에서 액션이 발생하면, 해당 액션을 가지고 있는 각 LTS들의 상태들은 전이되고 해당 액션이 없는 LTS는 멈춘다. 내부 액션 τ이 발생하면, 그에 해당하는 LTS만 전이한다.

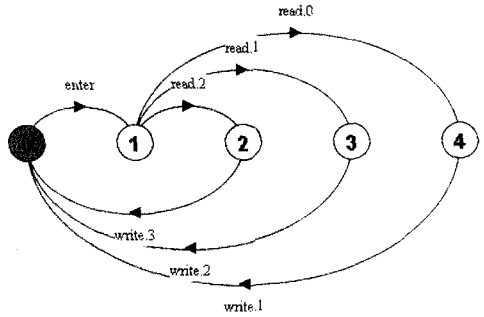
아래의 모델은 병행 결합된 프로세스의 예이다. 0부터 2까지의 값을 저장하는 변수를 모델링한 VAR 프로세스와 회전문을 모델링한 TURNSTILE이라는 프로세스의 결합을 나타낸 것이다.

```
const N = 2
range T = 0..N

VAR      = VAR[0],
VAR[u:T] = (read[u]   ->VAR[u]
           |write[v:T]->VAR[v]).

TURNSTILE = (enter -> INCREMENT),
INCREMENT = (read[x:T] -> write[x+1]-> TURNSTILE
            )+{write[0]}.

||TURN_VAR = (TURNSTILE || VAR).
```

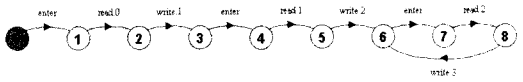
위에서 TURNSTILE이라는 프로세스는 enter 액션 이후에 INCREMENT라는 프로세스로 진행된다. INCREMENT 프로세스는 read[x] 액션과 write[x+1] 액션을 수행하고 TURNSTILE으로 돌아간다. 여기서 x의 범위가 T이므로 이 두 액션은 read[0] -> write[1], read[1] -> write[2], read[2] -> write[3]과 같은 액션의 연속으로 전개된다. 즉 0을 읽고 1을 쓰고, 1을 읽고 2를 쓰는 것을 모델링한 것이다. 이때, TURNSTILE과 VAR을 결합시켜서 GUN의 shot 액션이 수행된 횟수를 VAR의 상태에 저장하는 것을 모델링 한다. (그림 6)은 GUN 프로세스를 LTS로 나타낸 것이다.

다시 TURNSTILE 프로세스의 정의를 살펴보면 알파벳 확장 +{write[0]}이 보인다. 원래 TURNSTILE에는 없는 액션인 write[0]을 추가하게 되면, VAR의 write[0]액션과 동기화 되어서 병행 결합된 프로세스에서는 나타나지 않게 된다. 이 예제에서 0을 쓴다는 액션은 의미가 없

으므로 알파벳 확장을 통해서 모델에서 제거된다. 병행 결합된 프로세스의 LTS는 (그림 7)과 같다. 두 프로세스의 공통된 액션은 즉, read[T]와 write[T]는 모두 동기화 된다.

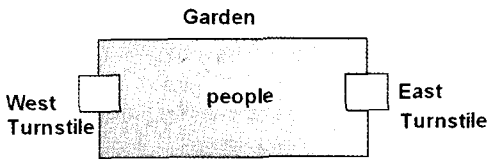


(그림 6) TURNSTILE 프로세스의 LTS



(그림 7)TURN_VAR의 LTS

3.3 멀티 스레드 모델링



(그림 8) 식물원

이번 절에서는 (그림 8)의 멀티 스레드의 모델링 예제로 1993년 알란 번스와 고프 데이비스에 의해서 소개된 식물원 문제를 다룬다[1]. 이 식물원엔 사람들이 입장할 수 있는 문이 양쪽에 있고 관리자는 언제든지 식물원 안에 있는 사람의 수를 알기 원한다. 문제를 더 간단히 하기 위해서 사람들은 식물원에 입장하기만 한다고 하자. 이 식물원에서 입장객 수를 세는 병행 프로그램을 모델링 하는 것으로 문제를 축소한다. 이 프로그램은 두 개의 병행 스레드와 공유되는 하나의 카운터 객체로 구성된다. 각각의 스레드는 양쪽의 문을 제어하고 입장객이 문으로 들어올 때, 카운터 객체를 증가시킨다.

```
const N = 4
range T = 0..N

VAR      = VAR[0],
VAR[u:T] = (read[u]   ->VAR[u]
           |write[v:T]->VAR[v]).

TURNSTILE = (go    -> RUN),
RUN       = (arrive-> INCREMENT
            |end   -> TURNSTILE),
INCREMENT = (value.read[x:T]
             -> value.write[x+1]->RUN
            )+{write[0]}.

DISPLAY =(value.read[T]->
                  DISPLAY)+{value.write[T]}.

||GARDEN = (east:TURNSTILE || west:TURNSTILE
           || display:DISPLY
           || {east,west,display}::value:VAR)
           / { go/{ east,west}.go,
               end/{ east,west}.end }.
```

식물원 문제를 모델링하기 위해서 읽기와 쓰기 접근을 묘사하는 VAR 프로세스를 포함시켰다. 양쪽 문에 해당하는 각각의 스레드 객체를 east와 west로 레이블링 한 TURNSTILE로 표현했다. 공유 변수 VAR은 east와 west뿐 아니라, 검사 목적으로 사용되는 display에서도 공유된다. DISPLAY 프로세스는 VAR가 어떤 상태에 있는지 읽는 역할을 수행한다.

멀티 스레드 모델에서 에러가 있는지 찾기 위해서는 많은 시나리오를 고려해야 한다. 에러를 찾기 위해서는 에러의 경로를 포함하는 프로세스를 정의해야 한다. 그런 후 프로그램을 모델링한 프로세스와 에러를 정의한 프로세스를 병행 결합한다. 이렇게 병행 결합한 프로세스에서 데드락 존재 여부를 검사하게 된다. 에러를 찾기 위해 정의한 프로세스 TEST는 다음과 같다.

```
set VarAlpha = {value.{read[T],write[T]}}

TEST      = TEST[0],
TEST[v:T] =
    (when (v<N) {east.arrive,west.arrive }->TEST[v+1]
    |end->CHECK[v]
    ),
CHECK[v:T] =
   (display.value.read[u:T] ->
     (when (u==v) right -> TEST[v]
     |when (u!=v) wrong -> ERROR
     )
   )+ {display.VarAlpha}.
```

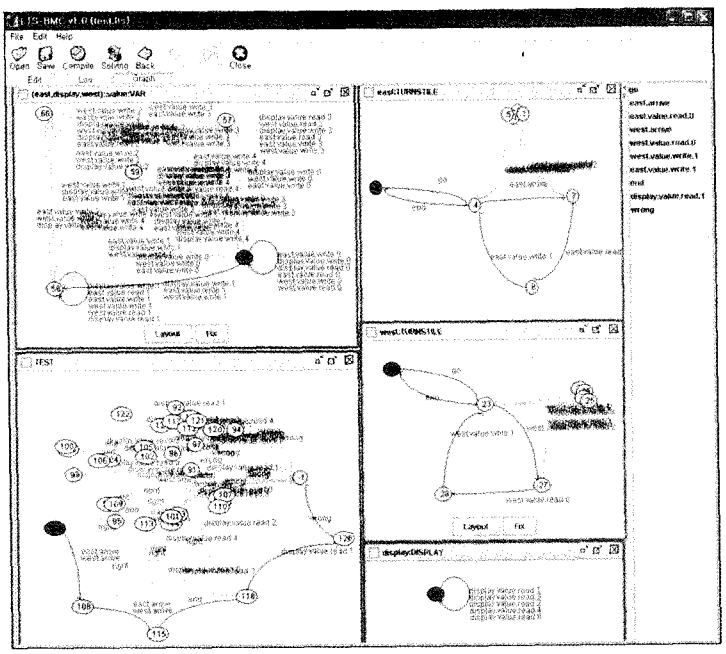
위의 프로세스는 east.arrive와 west.arrive 액션의 전체 수를 센다. end 액션이 발생 했을 때, 그래서 결국 공유 변수의 갱신이 끝났을 때, 공유된 값이 전체 도착한 이벤트의 수와 일치하는지 검사한다. 만약에 일치하지 않는다면, ERROR 상태로 이동하는 에러 경로가 생성된다. ERROR는 FSP에서 정의된 예약어이고 LTS에서는 해당 상태에 -1이 레이블 된다. 프로세스 TEST는 아래와 같이 식물원 모델과 결합된다.

```
||TESTGARDEN = (GARDEN || TEST).
```

이제 TEST에서 ERROR 상태에 도달할 수 있는지 알아보기 위해 분석 도구를 통해서 철저한 탐색을 할 수 있다. 에러 경로의 예는 다음과 같다.

```
go
east.arrive
east.value.read.0
west.arrive
west.value.read.0
east.value.write.1
west.value.write.1
end
display.value.read.1
wrong
```

이 트레이스는 분명히 모델의 문제점을 지적하고 있다. 공유변수가 원자적으로(atomically) 갱신되지 않았기 때문에 공유변수의 갱신이 잘못 수행 되었다. 위의 경로는 동쪽과 서쪽 문에서 동시에 0을 읽고 1을 쓰게 된 것이다. 만일 서쪽의 증가가 시작되기 전에 동쪽의 증가가 끝난다던지, 그 반대에 경우라면, 결과는 2가 되었을 것이다. (그림 9)는 분석도구를 사용해서 얻어낸



(그림 9) TESTGARDEN 분석 결과

결과 화면이다. 다섯 개의 작은 창은 각각 TEST, east:TURNSTILE, west:TURNSTILE, display: DISPLY, {east,west,display}::value: VAR 등의 프로세스들이다. 그림에서 작은 원이 상태를 나타내고, 곡선들은 전이를 나타낸다. 짙은 색으로 표현된 전이가 에러 경로에 해당하는 전이들이다. 이와 같은 특징을 갖는 에러를 인터리빙에 의해 발생하는 간섭(interference)이라고 한다.

위와 같은 문제는 설계를 잘 못 해서 발생한 것이다. 앞서 언급했던 바와 같이 수행되는 액션들, 즉 프로그램 편에서 말하자면, 실행되는 문장들이 원자성을 가져야 할 때가 있는데, 위의 모델은 이러한 특징을 반영하지 못했다. 자바 언어에서는 원자성을 부여하는 구문이 존재하는데 모델에서도 이러한 속성을 추가함으로 해서 올바른 모델을 만들어 낼 수 있다.

모델을 수정하는 간단한 방법은 모델에 원자적 액션들이 실행될 동안 다른 프로세스의 접근을 불허하는 잠금 장치를 추가하는 것이다. 잠금 장치는 아래와 같이 모델링 될 수 있다.

```
LOCK = (acquire->release->LOCK).
||LOCKVAR = (LOCK || VAR).
```

GARDEN 프로세스의 VAR를 LOCK과 VAR를 결합한 LOCKVAR로 대치한다. 그리고 알파벳의 집합인 VarAlpha를 추가한다. 마지막으로 반드시 변수에 접근하기 전에 lock을 획득하고 접근한 후에 lock을 해제하도록 변경하면 모델 수정은 완료된다. 이렇게 변경한 모델은 이전에 정의했던 test를 사용해서 검사해도 에러를 발견할 수 없다.

```
TURNSTILE = (go    -> RUN),
RUN       = (arrive-> INCREMENT
            |end   -> TURNSTILE),
INCREMENT = (value.acquire
             -> value.read[x:T]->value.write[x+1]
             -> value.release->RUN)+VarAlpha.

set VarAlpha = {value.{read[T],write[T],acquire, release}}
```

3.4 분석방법

이 장에서는 분석이 어떻게 이루어지는지에 대해서 다룬다. 분석 방법은 모델의 실행 경로 σ의 각 스텝을 나타내는 논리식과 속성을 나타내는 논리식의 논리곱이 만족 가능한 식인지 결정하는 문제이다. 이제 모델과 속성이 어떻게 인코딩 되어서 SAT 처리기의 입력부로 변환되는지를 설명한다.

3.4.1 모델 인코딩 방법

상태공간을 나타내게 되는 LTS는 내부 액션 τ를 생략하고 초기 상태는 하나로 제약한다. 먼저 인코딩을 간략하게 설명하면, 병행 결합에 참여하는 각 로컬 LTS들에서 초기상태와 상태 집합, 액션 집합 그리고 발생 가능한 액션들에 대한 조건들이 인코딩 되고 글로벌 액션집합에 대한 조건과 글로벌 액션과 로컬 액션의 연관관계가 인코딩 된다. 글로벌 액션은 병행 결합된 LTS의 액션을 의미한다. 마지막으로 각 LTS의 전이를 인코딩 한다. 다음은 인코딩 규칙들이다.

(1) 각 LTS의 초기 상태는 스텝 1에서 모두 참이다. 따라서 앞의 식 (1)과 동일하게 인코딩 된다. 그러나 초기 상태가 하나이므로 q_i가 L_i의 초기 상태라고 할 때, 아래와 같이 인코딩 된다. $in(q_i,\theta)$는 "q_i상태가 θ스텝에서 참이다."라는 의미이다.

$$\bigwedge_{1 \le i \le n} in(q_i,1)$$

(2) 각 LTS의 상태들은 스텝 θ에서 오직 하나의 상태만이 선택 된다. $card_1^1\{...\}$는 집합 내의 원소 중 정확히 하나만 선택할 때 참이 되는 술어이다.

$$\bigwedge_{1 \le i \le n} card_1^1\{in(s,\theta) | s \in S_i\}$$

(3) 로컬 LTS에서 액션들은 최대 한 개 까지만 발생할 수 있다. $card_0^1\{...\}$는 집합 내의 원소

중 적어도 하나가 선택할 때 참이 되는 술어이다. $ex(a,i,\theta)$는 로컬 액션 a가 i번째 LTS에서 스텝 θ에 실행된다는 의미의 술어이다.

$$\bigwedge_{1\le i\le n} card_0^1 \in \{ex(a,i,\theta)\,|\,a\in A_i\}$$

(4) 로컬 상태마다 가능한 액션들을 인코딩해 준다. LTS L_i의 전이 집합 Δ_i중에서 상태 s를 시작상태로 하는 전이들의 집합을 $\Delta_i^s=\{(s,a,s')\in\Delta_i\}$로 정의한다. 그리고 $act(t)$는 전이 t의 액션을 돌려주는 함수이다.

$$\bigwedge_{1\le i\le n}\bigwedge_{s\in S_i}\left(in(s,\theta)\Rightarrow\bigvee_{t\in\Delta_i^s} ex(act(t),i,\theta)\right)$$

(5) 이제 글로벌 액션들에 대한 조건을 인코딩한다. 글로벌 액션 또한 한 스텝에서 기껏해야 하나의 액션만 실행된다. $ex(a,\theta)$는 글로벌 액션 a가 스텝 θ에 실행된다는 의미의 술어이다.

$$card_0^1\in\{ex(a,\theta)\,|\,a\in A\}$$

(6) 그리고 로컬 액션이 발생되지 않았다면, 글로벌 액션도 발생하지 않게 된다.

$$\bigwedge_{1\le i\le n}\bigwedge_{a\in A_i}(\neg ex(a,i,\theta)\Rightarrow\neg ex(a,\theta))$$

(7) 이제 글로벌 액션이 반드시 하나 발생되는 조건을 인코딩 한다. $C(a)$는 액션 a를 가지고 있는 LTS의 인덱스를 돌려주는 함수 이다. 동기화 되지 않은 액션 집합을 $NA\subseteq A$라고 하고 동기화에 참여한 액션 집합을 $SA\subseteq A$라고 하자. 그리고 $NA\cap SA=\varnothing$이다.

$$\left(\left(\bigvee_{a\in NA} ex(a,C(a),\theta)\right)\vee\left(\bigvee_{a\in SA}\bigwedge_{i\in C_a} ex(a,i,\theta)\right)\right)\Rightarrow\bigvee_{a\in A} ex(a,\theta)$$

(8) 마지막으로 로컬 LTS의 전이들은 아래와 같이 인코딩 된다.

$$\bigwedge_{1\le i\le n}\bigwedge_{(s,a,s')\in\Delta_i} in(s,\theta)\wedge ex(a,i,\theta)\Rightarrow$$

$$((ex(a,\theta)\Rightarrow in(s',\theta+1))\wedge(\neg ex(a,\theta)\Rightarrow in(s,\theta+1)))$$

(9) 따라서 LTS모델 M에 대한 인코딩은 아래와 같이 (1)식부터 (8)식까지 논리곱으로 묶은 식이 된다.

$$(1)\wedge\bigwedge_{1\le\theta\le k}\bigwedge_{2\le\alpha\le 8}(\alpha)$$

식 (1), (2), (3), (4)는 로컬 LTS에 대한 식으로 현재 참인 상태에서 발생 가능한 액션들을 제공한다. 식 (5), (6), (7)은 글로벌 액션에 관한 식으로 글로벌 액션의 제약사항과 발생 조건을 인코딩한다. 식 (7)는 동기화에 참여하지 않는 로컬 액션들 중 하나가 발생할 때와 동기화된 액션들이 동기화 되어서 발생할 때, 글로벌 액션은 반드시 발생되어야 한다. $C(a)$는 동기화에 참여하지 않은 액션에만 적용되는 함수이다. 즉, $a\in A_i\Rightarrow\wedge_{1\le i\ne j\le n}(a\notin A_j)$이다. 따라서 $a\in A_j$일 때, $C(a)=i$이다. 식 (8)은 로컬 상태와 로컬 액션이 선택될 때, 글로벌 액션이 참이면, 다음상태로 전이하고 글로벌 액션이 거짓이면, 제자리에 머물게 됨을 나타낸다.

모델을 위의 규칙 (9)처럼 표현하면, 이제는 속성을 SAT 처리기의 입력으로 표현해 주어야 한다. 이 논문에서 다루고자 하는 속성은 데드락이다. 데드락은 정의 2에서 설명한 바와 같이 어떠한 액션도 발생되지 않는 상태이다. 이것은 아래와 같이 글로벌 액션들이 발생 되지 않는 것으로 표현될 수 있다.

$$\bigvee_{1\le\theta\le k}\left(\bigwedge_{a\in A}\neg ex(a,\theta)\right)$$

식 (1), (2), (3), (4)는 로컬 LTS에 대한 식으로 현재 참인 상태에서 발생 가능한 액션들을 제공한다. 식 (5), (6), (7)은 글로벌 액션에 관한 식으로 글로벌 액션의 제약사항과 발생 조건을 인코딩한다. 식 (7)는 동기화에 참여하지 않는 로컬 액션들 중 하나가 발생할 때와 동기화된 액션들이 동기화 되어서 발생할 때, 글로벌 액션은 반드시 발생되어야 한다. $C(a)$는 동기화에 참여하지 않은 액션에만 적용되는 함수이다. 즉, $a\in A_i\Rightarrow\wedge_{1\le i\ne j\le n}(a\notin A_j)$이다. 따라서 $a\in A_j$ 일 때, $C(a)=i$이다. 식 (8)은 로컬 상태와 로컬 액션이 선택될 때, 글로벌 액션이 참이

면, 다음상태로 전이하고 글로벌 액션이 거짓이면, 제자리에 머물게 됨을 나타낸다.

3.4.2 속성 인코딩

모델을 위의 규칙 (9)처럼 표현하면, 이제는 속성을 SAT 처리기의 입력으로 표현해 주어야 한다. 데드락은 정의 2에서 설명한 바와 같이 어떠한 액션도 발생되지 않는 상태이다. 이것은 아래와 같이 글로벌 액션들이 발생 되지 않는 것으로 표현될 수 있다. 아래의 식은 시스템을 k 스텝까지 실행 했을 때, 글로벌 액션이 전혀 발생하지 않을 때가 존재 한다고 하는 선언이다. 만약에 만족한다면, 즉 데드락이 존재한다면, SAT 처리기는 그 증거를 제시하게 된다.

$$\bigvee_{1 \le \theta \le k} \left(\bigwedge_{a \in A} \neg ex(a,\theta) \right)$$

이제 속성을 보다 일반적으로 표현할 수 있는 $FLTL$의 정의와 인코딩 방법을 알아보자. 지면상 LTS식의 의미와 인코딩 방법은 [8,9]를 참조하기 바란다. 본 논문에서는 변량(fluent)이 어떻게 정의되고 인코딩 되는지를 중심으로 기술한다.

LTS는 액션 기반의 모델 언어이다. 따라서 크립키 구조에서 LTL식을 표현하고 해석하는 방법을 LTS에 바로 적용할 수는 없다. 따라서 변량을 원소명제로 사용하는 LTL로 속성을 기술하는데, 정의 4의 변량 Fl은 액션의 발생 여부에 의해서 진위 값이 변하는 명제 변수이다.

정의 4. 변량 Fl은 액션 집합 I_{Fl}과 T_{Fl}로 아래와 같이 정의된다. 여기서 액션 집합 I_{Fl}과 T_{Fl}은 각각 시작액션 집합과 종료액션 집합이라고 부르고, 정의 3의 A의 부분집합이다. $b_{Fl} \in \{true, false\}$는 Fl의 초기값이다.

$$Fl = \langle I_{Fl}, T_{Fl} \rangle \, b_{Fl}$$

Fl은 LTS에서 시작액션 집합에 속하는 액션이 발생할 대 참이 되고, 종료액션 집합에 속하는 액션이 발생할 때, 거짓이 된다. $I_{Fl} \cap T_{Fl} = \varnothing$이고, $E_{Fl} = A \backslash (I_{Fl} \cup T_{Fl})$에 속하는 액션이 발생하면 Fl의 값은 변하지 않는다. 실행 스텝 i에서 변량 Fl가 만족한다는 것은 다음 식이 참이 된다는 것과 같다.

$$\bigvee_{a \in I_{Fl}} (ex(a,i))$$

비슷하게 실행 스텝 i에서 변량 Fl가 불 만족한다는 것은 아래의 식이 참이 된다는 것과 같다.

$$\bigvee_{a \in T_{Fl}} (ex(a,i))$$

따라서 변량 Fl을 인코딩하면 아래의 규칙 (10)번과 같이 인코딩 될 수 있다.

(10) $$b_{Fl} \wedge \bigwedge_{1 \le i \le n} \Big(\bigvee_{a \in I_{Fl}} (ex(a,i) \Rightarrow Fl_i) \wedge \bigvee_{a \in T_{Fl}} (ex(a,i) \Rightarrow \neg Fl_i) \wedge \bigvee_{a \in E_{Fl}} (ex(a,i) \wedge Fl_{i-1} \Rightarrow Fl_i) \wedge \bigvee_{a \in E_{Fl}} (ex(a,i) \wedge \neg Fl_{i-1} \Rightarrow \neg Fl_i) \Big)$$

아래에서 기술한 변량과 $FLTL$식으로 앞 절의 모델을 검사하면, 동일한 결과를 얻을 수 있다.

```
fluent F1 = <end>
fluent F2 = <wrong>
assert A = <>(F1 && <> F2)
```

첫 번째 줄에서 〈end〉는 시작액션 집합이 하나의 원소 end로 이루어졌음을 의미한다. 이렇게 시작액션 집합만 기술한 경우, 종료액션은 나머지 모든 액션이 된다. 〈〉는 언젠가는 참이라는 뜻이고 &&는 논리곱을 나타낸다. 따라서 위 식은 식물원 개장이 종료된 후 입장객 수를 잘못 세는 경로가 있는지 묻는 속성이다.

4. 결 론

본 논문은 복잡한 환경에서 동작하는 소프트웨어를 검증하기 위해 정형 논리의 개념 및 소프트웨어 검증 활용 예를 살펴보았다. 정형 논리는 일종의 정형 시스템인데, 정형 시스템은 모델링 및 분석을 정형화한 것으로서 정형 언어와 연역 장치로 구성된다. 정형 논리 중에 가장 많이 쓰이는 논리가 명제 논리이다. 명제 논리는 기계처

리가 용이하기 때문이다. 명제논리의 만족가능성을 결정하는 문제인 SAT 문제는 전산학의 많은 분야에 응용하고 있는데 소프트웨어의 검증도 그 중 한 분야이다.

병행 소프트웨어는 다양한 프로그래밍 환경에서 사용되고 있지만, 작성하고 검사하는 것이 매우 어렵다. 또한 에러가 존재하더라도 발생 빈도가 매우 낮기 때문에 테스팅을 통해서 걸러내기가 매우 힘들다. 따라서 병행 프로그램은 정형논리를 통해서 모델링하고 정형 분석 도구를 사용해서 철저하게 분석해야 한다. 더불어 소프트웨어를 논리적으로 모델링하고 분석하는 과정은 이 분야를 공부하는 학생들과 IT분야 실무자들이 반드시 갖추어야할 지식이다.

참고문헌

[1] J. Magee and J. Kramer, Concurrency - State Models and Java Programs, Chichester, John Wiley & Sons, 1999.

[2] E. M. Clarke, O. Grumberg and D. Peled, Model Checking, MIT Press, 1999.

[3] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without BDDs", In Proceeding of Workshop on Tools and Algorithms for the Construction and Analysis of Systems, LNCS, Springer-Verlag, 1999.

[4] A. Biere, A. Cimatti, E. Clarke, Ofer Strichman, and Y. Zhu, "Bounded Model Checking", Vol. 58 of Advances in Computers, 2003. Academic Press (pre-print).

[5] M.W. Moskewicz, C. Madigan, Y. Zhao, L. Zhang and S. Malik, "Chaff: Engineering an Efficient SAT Solver," In Proceedings of Design Automation Conference, 2001.

[6] http://www.cs.chalmers.se/Cs/Research/For-malMethods/MiniSat/

[7] T. Jussila, "BMC via dynamic atomicity analysis," In Proceedings of the International Conference on Application of Concurrency to System Design, IEEE Computer Society, June 2004.

[8] T. Jussila, K. Heljanko, and I. Niemela, "BMC via on-the-fly determinization," In Proceedings of the 1st International Work shop on Bounded Model Checking, 2003.

[9] T. Jussila, "On Bounded Model Checking of Asynchronous System," PhD thesis, Helsinki University, 2005.

[10] S. Park and G. Kwon, "SAT based Verification Tool for Labeled Transition System," In Proceedings of SERA2007, IEEE Computer Society, pp.221-226, 2007.

저자약력

박 사 천

2001년 경기대학교 전산학과(학사)
2003년 경기대학교 전산학과(석사)
2004년~현재경기대학교 전산학과 박사과정
관심분야 : 모델 체킹 , 정형기법, 소프트웨어 공학
이 메 일 : sachem@kyonggi.ac.kr

권 기 현

1985년 경기대학교 전산학과(학사)
1987년 중앙대학교 전산학과(이학석사)
1991년 중앙대학교 전산학과(공학박사)
1999년~2000년 미국 카네기멜론대학 전자계산학과 방문교수
2006년~2007년 미국 카네기멜론대학 전자계산학과 방문교수
1991년~현재경기대학교 정보과학부 교수
관심분야 : 소프트웨어 모델링, 소프트웨어 분석, 정형 기법, 소프트웨어 공학
이 메 일 : khkwon@kyonggi.ac.kr