병행성 분석을 위한 액션 기반의 LTS 바운드 모델 체커

(An Action-based LTS Bounded Model Checker for Analyzing Concurrency)

박 사 천 † 권 기 현 ††
(Sachoun Park) (Gihwon Kwon)

요 약 병행 컴포넌트를 포함하는 소프트웨어는 디버깅하기가 매우 어렵다. 따라서 철저하면서도 자동화된 검증 도구의 사용이 필수적이다. 이러한 도구 개발의 노력 중 하나가 바운드 모델 체킹 도구이다. 바운드 모델 체킹은 주어진 바운드 k 안에서 시스템의 실행 경로에 에러가 존재하는지를 철저히 검사한다. 본 논문에서는 LTS로 모델링 된 병행 프로그램을 검증하는 바운드 모델 체킹 도구를 소개한다. 이 도구에서 속성은 FLTL로 기술되는데 FLTL은 LTS 모델에서 액션을 가지고 속성을 표현하기에 적합하다. 우리는 기존 모델 체커들과의 실험을 통해서 개발된 도구의 성능을 비교분석한다.

키워드 : 정형 검증, 바운드 모델 체킹, FLTL(Fluent Linear Temporal Logic), LTS(Labeled Transition System)

Abstract Since concurrent software is hard to debug, the verification of such systems inevitably needs automatic tools which support exhaustive searching. Bounded Model Checking(BMC) is one of them. Within a bound k, BMC exhaustively check some errors in execution traces of the given system. In this paper, we introduce the tool that performs BMC for LTS, modeling language for concurrent programs. In this tool, a property is described by a FLTL formula, which is suitable to present the property with actions in aLTS model. To experiment with existential model checkers and out tool, we compare and analysis the performance of the developed tool and others.

Key words : Formal verification, Bounded model checking, FLTL(Fluent Linear Temporal Logic), LTS(Labeled Transition System)

1. 서 론

병행성(concurrency)은 매우 편리하고 유용한 프로그래밍 기술이다. 병행 프로그램은 운영체제나 실시간 내장형 시스템에서뿐 아니라 반응성(responsiveness)이나 처리량(throughput)이 중요한 대부분의 시스템에서 사용되고 있다. 그러나 병행 프로세스간에 예기치 않는 상호작용은 디버깅을 매우 어렵게 한다. 왜냐하면 실행 경로에 대한 경우의 수가 너무나 많기 때문이다[1].

모델 체킹은 사용자의 개입 없이 시스템의 실행 경로를 철저히 조사할 수 있는 검증 기법이다. 80년대에는 이 기법이 하드웨어 검증을 위해서 많이 사용되었으나 점차로 기술이 발전하여 최근에는 소프트웨어 모델 체킹이 가능하게 되었다. 소프트웨어는 하드웨어에 비해서 구조적이지 못하고 무한한 요소들이 많기 때문에 유한 상태 기계(finite state machine)를 기반으로 하는 모델 체킹으로는 소프트웨어를 분석하기가 쉽지 않다. 왜냐하면, 탐색해야 할 실행 경로가 너무나 크기 때문이다[2].

소프트웨어를 모델 체킹하기 위해서 먼저 소프트웨어를 유한 상태 기계로 변환해야 한다. 이때 검증에 불필

· 본 연구는 국토해양부 첨단도시기술개발사업-지능형국토정보기술혁신사업과제의 연구비지원(07국토정보C03)에 의해 수행되었습니다.
· 본 연구는 경기도의 경기도지역협력연구센터사업의 일환으로 수행하였음

† 학생회원 : 경기대학교 정보과학부
dhlee@adams.kw.ac.kr
†† 종신회원 : 경기대학교 정보과학부 교수
kchung@kw.ac.kr

논문접수 : 2007년 9월 19일
심사완료 : 2008년 8월 27일

정보과학회논문지: 소프트웨어 및 응용 제35권 제9호(2008.9)

요한 부분들은 제거되고, 속성이 보존되도록 추상화된다. 속성이 보존된다는 것은 추상화 된 유한 상태 기계에서 속성이 만족되면, 원래의 프로그램에서도 속성이 만족된다는 의미이다. 그러나 그 반대는 성립하지 않는다. 이렇게 추상화된 모델이라 할지라도 원래의 소프트웨어의 크기가 크거나 병행 동작하는 컴포넌트들이 많을 경우는 모델 체킹이 어렵다. 특히 병행 컴포넌트들은 검색해야 할 상태공간을 지수적으로 증가시켜 상태 폭발 문제를 발생시킨다[3].

바운드 모델 체킹(Bounded Model Checking)은 기존의 모델 체킹 대안으로 제안되었다. 여기서 바운드의 의미는 탐색 깊이를 주어진 바운드로 제한한다는 것이다. 즉, 검사할 시스템의 가능한 모든 상태 공간을 탐색하는 것이 아니고, 에러를 발견할 때까지 상태공간을 점차로 늘려가면서 검사하겠다는 것이다. 이러한 바운드 모델 체킹으로는 모든 상태공간을 탐색하는 것이 아니므로, 불변식(invariant)을 검사하기가 매우 어렵다. 따라서 이러한 기법을 검증(verification)이라고 하지 않고, 반증(falsification)이라고 한다[4,5].

BMC는 모델 체킹을 만족성(Satisfiability)의 문제로 간주하는 방식이다. 이러한 연구들은 최근 개발되고 있는 SAT 처리기들[6-8]의 높은 성능에 의존하고 있다. 모델 체킹이 만족성 문제로 해결되기 위해서는 검사할 대상과 속성을 SAT 처리기이 입력부인 CNF(Conjunctive Normal Form) 식 형태로 변환해 주어야 하는데 이것을 인코딩이라고 부른다. 인코딩은 SAT 처리기의 처리시간을 단축시키는 측면뿐 아니라, 속성을 검사하기 유리한 측면에서도 고려되어야 한다.

본 논문에서는 병행 프로그램을 모델링 할 때 많이 사용되는 LTS(Labeled Transition System)에 대한 바운드 모델 체킹 도구 LTS-BMC를 소개한다. LTS는 비동기화된 시스템을 모델링하기에 적합한 간단하고 일반적인 정형언어이다. 본 논문이 기여하는 바는 크게 두 가지이다. 첫째 우리가 개발한 LTS-BMC는 NuSMV 등에서 사용하는 크립키 구조와는 달리 액션을 기반으로 하는 LTS를 모델 언어로 하기 때문에 이에 적합한 CNF 인코딩 방법이 필요하다. 우리는 인코딩의 효율성을 위해서 [9]에서 소개된 이진 기수조건에 대한 인코딩을 활용했다. 둘째, LTS는 액션을 기반으로 하는 모델이기 때문에 속성 또한 액션을 기반으로 기술 되어야 한다. 따라서 본 논문에서는 액션을 사용해서 속성을 표현하는Fluent LTL을 소개하고 이에 대한 인코딩 기법을 설명한다.

[1]에서 병행 프로그램을 LTS로 모델링하는 방법이 소개되어 있다. 또한 그에 대한 분석도구인 LTSA(LTS Analyzer)[10]를 제공하고 있는데, 그래프 탐색 방식으로 속성을 검사하고 있어서 큰 규모를 검사하기에 적합하지 않다. 또한 최근에 LTSA 도구에서 fluent LTL 모델 체킹하는 연구가 있었다[11]. 여기서는 오토마타 기반의 모델 체킹 방식을 적용해서 속성을 LTS로 변환하여 모델의 LTS와 속성의 LTS를 병행 결합하는 방식을 취한다. 우리는 fluent LTL 바운드 모델 체킹을 위해서 fluent의 인코딩 방법을 기술한다.

한편 LTS 바운드 모델체킹을 시도한 논문[12-14]이 최근에 발표되었는데, 이 논문에서는 모델 체킹 성능 향상을 위해서 세 가지의 기법을 사용한다. 첫째, 반 순서 의미를 사용해서 바운드의 횟수를 줄였고, 둘째 비 결정성을 제거해서 실행 경로를 줄이고, 셋째 경로압축하는 인코딩 방법을 제안했다. 실험 결과 이러한 방법들은 LTS 바운드 모델 체킹의 성능을 향상시킬 수 있었다. 그러나 여기서는 LTS의 도달성 속성만을 다룰 뿐 시제논리를 속성으로 다루고 있지 않다. 우리는 LTS를 바운드 모델 체킹할 때 시제논리를 사용할 수 있도록 액션을 기반한 LTS 인코딩 방법과 fluent LTL 인코딩 방법을 제안한다. 또한 SAT 처리 향상을 위해서 LTS의 인코딩에 이진 기수조건에 대한 인코딩을 적용했다.

논문의 구성을 다음과 같다. 2장에서는 배경지식으로 LTS를 정의하고, BMC를 간략히 설명한다. 3장에서는 제안하는 인코딩 방법과 FLTL(Fluent LTL)[1,15] 속성에 대한 인코딩 방법을 기술한다. 4장에서는 기존 도구들과의 비교 실험을 통해서 LTS-BMC의 장단점을 분석하고 5장에서 결론 및 향후 연구로 맺는다.

2. 배경 지식

2.1 LTS

정의 1. LTS $L = \langle S, q, \Sigma, \Delta \rangle$는 상태집합 S와 초기상태 $q \in S$, 액션집합 Σ, 그리고 전이집합 $\Delta \subseteq S \times \Sigma \times S$로 구성되는데, 각 전이들은 특정 액션에 의해서 레이블 된다.

$t = (s, a, s')$을 LTS L의 전이라고 하면 S는 전이 t의 시작상태, s'은 목적상태라고 한다. $\tau \in \Sigma$는 외부에서 관찰되지 않는 내부 액션이다. 전이 t가 실행된다는 것은 어떤 상태 s에서 액션 a가 실행 가능하다는 것과 같다. LTS는 일반적으로 그래프로도 표현되는데, 상태는 정점으로 전이는 정점간의 간선로 표현된다. 그리고 액션은 각 간선에 레이블 된다.

LTS의 의미는 실행(executions)이라는 단어로 정의되는데, 하나의 전이가 실행된다는 것은 그에 해당하는 액션이 실행된다는 의미이다. 따라서 LTS의 실행은 전이들의 $t_1, t_2, \ldots$ 연속이다. 여기서 t_1은 LTS의 초기상태이다.

정의 2. LTS $L = \langle S, q, \Sigma, \Delta \rangle$ 의 실행 $\sigma = (s_1, a_1, s_2)$,

(s_2, a_2, s_3), ..., (s_k, a_k, s_{k+1})는 모든 $1 \le i \le k$에 대해서 $(s_i, a_i, s_{i+1}) \in \Delta$인 전이들의 연속이다. 여기서 $s_1 = q$이고 $|\sigma| = k$이다.

BMC는 유한 길이의 실행 상에서 속성을 검사하게 된다. 실행상의 어떤 상태 s_i에서 더 이상 액션이 발생하지 않는다면, 그 상태를 deadlock 상태라고 하고, 그 실행 경로를 deadlock 경로라고 한다. 따라서 deadlock은 어떠한 액션도 발생하지 않는 경우라고 하겠다.

이러한 LTS는 하나의 프로세스를 나타내게 되는데, 여러 개의 프로세스의 동작을 모델할 때, LTS들의 병행 결합으로 나타낸다.

정의 3. LTS $L_1, \cdots\cdots, L_n$이 있다고 하자. 여기서 $L_i = <S_i, q_i, \Sigma_i, \Delta_i>$이다. 이들의 병행 결합은 $L_1 \| \cdots\cdots \| L_n$로 나타내고 병행 결합된 LTS $L = <S, q, \Sigma, \Delta>$는 아래와 같이 정의된다.

- $S = S_1 \times ... \times S_n$,
- $q = <q_1, ..., q_n>$,
- $\Sigma = \Sigma_1 \cup ... \cup \Sigma_n$, 그리고
- $\Delta = \{(<s_1, ..., s_n>, l, <s'_1, ..., s'_n>) \in S \times \Sigma / \{\tau\} \times S |$
  for all $1 \le i \le n$ *if* $l \in$ *then* $(s_i, l, s'_i) \in \Delta_i$
  otherwise $s'_i = s_i \}$
  $\cup \{(<s_1, ..., s_n>, \tau, <s'_1, ..., s'_n>) \in S \times \Sigma / \{\tau\} \times S |$
  there is $1 \le i \le n$ *such that* $(s_i, \tau, s'_i) \in \Delta_i$
  and for all L_j, *if* $i \ne j$, *then* $s'_j = s_j \}$

2.2 바운드 모델 체킹

본 절에서는 [5]의 내용을 기반으로 바운드 모델 체킹에 대해 요약 설명한다. 바운드 모델 체킹은 시스템에서 가능한 길이 k의 경로 중에 주어진 속성을 만족하는 경로가 존재하는지 찾는 기술이다. 속성은 LTL(Linear Temporal Logic)로 기술된다. LTL은 고전 논리를 확장한 것인데, 명제 논리에 사용되는 $\wedge$, $\neg$ 등의 연산자 외에 경로에서 다음 상태를 나타내는 **X**, 경로의 모든 상태를 뜻하는 **G**, 경로에서 존재하는 어떤 상태를 의미하는 **F**, 그리고 특정 구간을 나타낼 때 사용하는 **U** 등의 연산자가 추가된다. $p \cup q$는 "주어진 경로 상에서 q가 만족할 때까지 p가 만족한다."라는 의미이다. 보통 LTL식은 크립키 구조(Kripke Structure)에서 해석된다.

정의 4. 크립키 구조 $M = <S, I, T, L>$은 네 개의 튜플로 구성되는데, 상태집합 S와 초기상태 집합 $I \subseteq S$, 전이집합 $T \subseteq S \times S$, 그리고 레이블 함수 $L : S \to 2^A$로 구성되는데, 여기서 A는 원소명제들의 집합이다. 따라서 크립키 구조는 LTS와는 달리 상태에 단위 속성들이 레이블 된다.

크립키 구조 M의 경로 $\pi = s_1, s_2, ...$는 상태들의 연속으로 표현된다. LTS에서와 같이 크립키 구조의 경로 또한 전이 관계를 기초로 하고 초기상태로부터 시작한다. π_i는 경로 π의 i번째 상태를 의미하는데 LTL 식은 이러한 경로 상에서 해석된다.

정의 5. π를 크립키 구조 M의 무한 경로라고 하자. f와 g는 LTL 식이고 식 f가 경로 π에서 만족한다는 것을 $\pi \models f$로 나타내면 LTL 식의 의미는 아래와 같이 귀납적으로 정의된다.

$\pi \models p$	iff	$p \in L(\pi_1)$
$\pi \models \neg f$	iff	$\pi \not\models f$
$\pi \models f \wedge g$	iff	$\pi \models f$ and $\pi \models g$
$\pi \models \mathbf{X} f$	iff	$\pi_1 \models f$
$\pi \models \mathbf{G} f$	iff	$\pi_i \models f$ for all $i \ge 1$
$\pi \models \mathbf{F} f$	iff	$\pi_i \models f$ for some $i \ge 1$
$\pi \models f \mathbf{U} g$	iff	$\pi_i \models g$ for some $i \ge 1$ and $\pi_j \models f$ for all $i > j \ge 1$

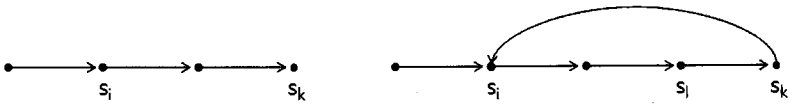
위의 정의를 바탕으로 바운드 k가 주어졌을 때, LTL 식의 의미를 정의해 보자.

정의 6. 바운드 $k \ge 1$가 주어졌을 때, 루프가 존재하지 않는 경로 π에서 LTL 식 f를 만족한다는 것은 $\pi \models^1_k f$와 같고 아래와 같이 귀납적으로 정의된다.

$\pi \models^i_k p$	iff	$p \in L(\pi_i)$
$\pi \models^i_k \neg p$	iff	$p \notin L(\pi_i)$
$\pi \models^i_k f \wedge g$	iff	$\pi \models^i_k f$ and $\pi \models^i_k g$
$\pi \models^i_k f \vee g$	iff	$\pi \models^i_k f$ or $\pi \models^i_k g$
$\pi \models^i_k \mathbf{X} f$	iff	$\pi_1 \models^{i+1}_k f$ and $i < k$
$\pi \models^i_k \mathbf{G} f$	is always false	
$\pi \models^i_k \mathbf{F} f$	iff	$\pi_i \models^i_k f$ for some $i < 1$
$\pi \models^i_k f \mathbf{F} g$	iff	$\pi_i \models^i_k f$ for some $k < j < i$ and $\pi_j \models^i_k f$ for all $j < n < i$

SAT 처리기에 의해서 검사되기 위에서 정의된 LTL 식은 모두 CNF 형태로 변환되어야 한다. LTL식의 변환은 경로상에서 이루어지므로 먼저 경로상에 루프가 존재할 경우와 그렇지 않은 경우로 나누어서 생각해 볼 수 있다. 그림 1의 (b)와 같이 경로 π상에 루프가 존재한다면 경로의 마지막 상태인 s_k와 루프가 시작되는 상태 s_l 사이의 전이 $(s_k, s_l) \in T$가 존재하게 된다. 이러한 전이가 존재할 때, 명제 변수 ${}_iL_k$는 참이 되고, 경로 내에 루프가 존재한다는 술어 L_k는 $V^k_{l=1}({}_iL_k)$로 인코딩 된다. ${}_l[\![\cdot]\!]^i_k$는 LTL식과 i, k, l의 입력을 받아서 명제논리식을 돌려주는 함수 기호이다. 정의 7은 경로의 각 상태에서 LTL식이 CNF로 변환되는 규칙을 설명한다.

정의 7. LTL식 f가 $k \ge l, i$인 $l, i, k \ge 1$와 함께 주어졌을 때, 아래와 같이 반복적으로 변환된다.



(a) no loop (b) (k, l)-loop

그림 1 바운드 경로에 두 가지 경우

$$
\begin{aligned}
&{}_l[\![p]\!]_k^i := p(s_i)\\
&{}_l[\![f \vee g]\!]_k^i := {}_l[\![f]\!]_k^i \vee {}_l[\![g]\!]_k^i\\
&{}_l[\![\boldsymbol{G}f]\!]_k^i := {}_l[\![f]\!]_k^i \wedge {}_l[\![\boldsymbol{G}f]\!]_k^{succ(i)}\\
&{}_l[\![\boldsymbol{X}f]\!]_k^i := {}_l[\![f]\!]_k^{succ(i)}
\end{aligned}
$$

$$
\begin{aligned}
&{}_l[\![\neg p]\!]_k^i := \neg p(s_i)\\
&{}_l[\![f \vee g]\!]_k^i := {}_l[\![f]\!]_k^i \vee {}_l[\![g]\!]_k^i\\
&{}_l[\![\boldsymbol{F}f]\!]_k^i := {}_l[\![f]\!]_k^i \vee {}_l[\![\boldsymbol{F}f]\!]_k^{succ(i)}\\
&{}_l[\![g\boldsymbol{U}f]\!]_k^i := {}_l[\![g]\!]_k^i \vee \left({}_l[\![f]\!]_k^i \wedge {}_l[\![g\boldsymbol{U}f]\!]_k^{succ(i)}\right)
\end{aligned}
$$

앞의 정의 7에서 $p(s_i)$는 상태 s_i에 명제 p가 레이블 되어 있을 때 참이 되는 술어이다. 만일 루프가 존재하지 않는 경로라면, l이 사용되지 않으므로 ${}_l[\![f]\!]_k^i$는 $[\![f]\!]_k^i$로 표기하고 $succ(i)$는 $i+1$이 된다. 그리고 $[\![f]\!]_k^{k+1}=0$으로 인코딩 된다. 만일 루프가 존재한다면, $i<k$인 i에 대해서는 $succ(i)=i+1$이고 $i=k$라면 $succ(i)=l$이다. 이제 모델과 속성이 주어졌을 때, 생성되는 전체 CNF 식을 정의해 보자.

정의 8. LTL식 f와 크립키 구조 M, 그리고 바운드 $k \geq 1$가 주어졌을 때, 변환된 CNF 식은 아래와 같다.

$$[\![M,f]\!]_k := [\![M]\!]_k \wedge \left((\neg L_k \wedge [\![f]\!]_k^1) \vee \bigvee_{l=1}^{k} \left({}_lL_k \wedge {}_l[\![f]\!]_k^1\right)\right)$$

위의 정의 8에서 모델 M은 크립키 구조였지만 본 논문에서는 LTS를 모델로 사용하므로, 다음 장에서는 LTS에 대한 인코딩을 설명한다. 먼저 [12]의 인코딩 방법을 기술하고, 이 인코딩이 왜 LTL 식과 함께 인코딩 하기가 어려운지 설명한 후 우리가 제안하는 인코딩 방법을 설명한다. 또한 정의 6에서 원소 명제에 대한 만족성을 기술한 $\pi \vdash_k^i p$ 와 그것의 인코딩에 해당하는 정의 7의 ${}_l[\![p]\!]_k^i := p(s_i)$ 식은 LTS에서 적용될 수 없기 때문에, LTS와 LTL식을 연결하는 fluent LTL 개념을 도입한다.

3. LTS-BMC의 인코딩

3.1 BMC에서 이진 기수조건 인코딩

이진 기수조건은 "n개의 이진 변수 중에 (적어도, 기껏해야, 혹은 정확하게) k개가 참이 된다"이다. 이러한 조건은 주어진 문제를 SAT으로 변환하는 다양한 분야에서 사용되는데, 바운드 모델 체킹에도 적용할 수 있다. 예를 들어서 "각 LTS에서 상태들 중 정확히 하나만 참이 된다", "전역 액션은 정확히 하나만 참이 된다." 혹은 "각 LTS에서 기껏해야 하나의 전이만 참이 된다" 등 LTS를 인코딩 하는데 빈번하게 사용된다. 따라서 이에 대한 효율적인 인코딩은 전체 검증 시간에 영향을 미친다. 본 논문에서는 아래와 같은 표기법을 사용한다.

- $card_1^1\{x_1,\ldots,x_n\}$: 집합의 원소 중 정확히 하나의 변수만 참이 되도록 하는 CNF 식
- $card_0^1\{x_1,\ldots,x_n\}$: 집합의 원소 중 기껏해야 하나의 변수만 참이 되도록 하는 CNF 식

$card_1^1\{\ \}$는 적어도 하나와 기껏해야 하나의 조건을 기술한 식의 논리곱으로 표현된다. 적어도 하나와 기껏해야 하나에 대한 일반적인 식이 아래와 같다.

$$\left(\bigvee_{1\le i\le n} x_i\right) \wedge \left(\bigwedge_{1\le i<n}\ \bigwedge_{i<j\le n} \neg x_i \vee \neg x_j\right)$$

위와 같은 일반적인 방식으로는 SAT 처리기의 빠른 수행을 기대하기 어렵다. 우리는 $card_0^1\{x_1,\ldots,x_n\}$을 인코딩하기 위해서 [10]에서 제안한 아래의 CNF 식을 사용했다. 여기서 이진 변수 s_i는 보조변수이다. 아래 식에서 사용된 보조변수의 개수는 $n-1$개이다.

$$(\neg x_1 \vee s_1) \wedge (\neg x_n \vee \neg s_{n-1}) \wedge \bigwedge_{1\le i<n} \left((\neg x_i \vee s_i) \wedge (\neg s_{i-1} \vee s_i) \wedge (\neg x_i \vee \neg s_{i-1})\right)$$

LTS의 인코딩을 간결하게 표현하기 위해서 몇 가지 술어를 사용한다. $in(s,\theta)$가 참이라는 것은 스텝 θ에서 상태 s에 도달했다는 의미이다. $ex(t,\theta)$이 참이라는 것은 스텝 θ에서 전이 t가 실행된다는 의미이다. 리터럴 $ex(a,j,\theta)$의 의미는 LTS L_j에서 액션 a가 스텝 θ에서 실행된다는 것을 의미한다.

3.2 LTS 인코딩 방법

이제 LTS에 대한 인코딩 기법을 소개하고자 한다. 내부 액션 τ는 모델이 컴파일 될 때, 모델 축소(minimize)에 의해 제거된다고 가정한다. 먼저 전체 인코딩을 간략하게 설명하면, 병행 결합에 참여하는 각 로컬 LTS들에서 초기상태와 상태 집합, 액션 집합 그리고 발생 가능한 액션들에 대한 조건들이 인코딩 된 후 글로벌 액션 집합에 대한 조건과 글로벌 액션과 로컬 액션의

연관관계가 인코딩 된다. 마지막으로 각 LTS의 전이를 인코딩 한다. 정의 3의 의미를 따르는 병행 LTS들의 인코딩은 다음 식들과 같다.

(1) 각 LTS의 초기 상태는 스텝 1에서 모두 참이다. LTS 마다 초기 상태가 하나이므로 q_i 가 L_i 의 초기 상태라고 할 때, 아래와 같이 인코딩 된다.

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

(2) 각 LTS의 상태들은 스텝 θ 에서 정확히 하나의 상태만이 선택 된다.

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

(3) 로컬 LTS에서 액션들은 최대 한 개 까지만 발생할 수 있다.

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

(4) 로컬 상태마다 가능한 액션들을 인코딩 해 준다. LTS L_i 의 전이 집합 Δ_i 중에서 상태 s 를 시작상태로 하는 전이들의 집합을 $\Delta_i^s = \{t \mid t = (s, a, s') \square \wedge t \in \Delta_i\}$로 정의한다. 그리고 $act(t)$ 는 전이 t 의 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 인코딩 한다. 글로벌 액션 또한 한 스텝에서 기껏해야 하나의 액션만 실행된다.

$$card_0^1\{ex(a,\theta) \mid a \in \Sigma\}$$

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

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

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

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

(8) 마지막으로 로컬 LTS의 전이들은 아래와 같이 인코딩 된다.

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

(9) 따라서 LTS 모델 L 에 대한 인코딩 $\|L\|_k$ 은 아래와 같이 (1)식부터 (8)식까지 논리곱으로 묶은 식이 된다.

$$[\![L]\!]_k \coloneqq (1)\wedge\bigwedge_{1\le\theta\le k}\bigwedge_{2\le\alpha\le 8}(\alpha)$$

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)은 동기화에 참여하지 않는 로컬 액션들 중 하나가 발생할 때와 동기화 액션들이 동기화 되어서 발생할 때, 글로벌 액션은 반드시 발생되어야 한다는 것을 인코딩 한다. $C(a)$는 동기화에 참여하지 않은 액션에만 적용되는 함수이다. 즉, $a \in \Sigma_i \Rightarrow \wedge_{1\le i\ne j\le n} a \notin \Sigma_j$ 이다. 따라서 $a \in \Sigma_i$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. 이제 다음 절에서는 LTL식이 어떻게 LTS와 함께 인코딩 되는지 기술한다.

3.3 FLTL(Fluent LTL) 인코딩

LTS는 액션 기반의 모델언어이기 때문에, LTL 식을 표현하고 해석하는 방법이 상태에 원소 명제들의 집합을 레이블하는 크립키 구조와는 다르다. 변량(fluent) Fl 은 액션의 발생여부에 의해서 진위 값이 변하는 명제 변수이다. 아래 정의 9는 변량에 대한 정의이다.

정의 9. 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 액션 집합 Σ에 포함된다. $b_{Fl} \in \{true, false\}$ 는 Fl 의 초기값이다.

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

Fl 은 LTS에서 시작액션 집합에 해당하는 액션이 발생할 때 참이 되고, 종료액션집합에 해당하는 액션이 발생할 때, 거짓이 된다. $I_{Fl} \cap T_{Fl} = \varnothing$ 이고, $E_{Fl} = \Sigma/(I_{Fl} \cup T_{Fl})$ 인 액션이 LTS에서 발생하면 Fl 의 값은 변하지 않는다. 이제 정의 6의 처음 두 개의 정의를 다시 쓰면 아래와 같다.

$$\pi \vDash_k^i Fl \quad \text{iff} \quad \mathsf{V}_{a\in I_{Fl}}(ex(a,i))$$
$$\pi \vDash_k^i \neg Fl \quad \text{iff} \quad \mathsf{V}_{a\in T_{Fl}}(ex(a,i))$$

정의 10. 정의 7의 처음 두 개의 인코딩을 아래와 같이 변량에 대한 인코딩으로 바꾸어서 FLTL의 인코딩을 정의 한다. 나머지 LTL식에 대해서는 동일하게 인코딩

된다. 여기서 $Fl_0 = b_{Fl}$ 이다.

$$
\begin{aligned}
{}_l[\![Fl]\!]^i_k := \quad & b_{Fl} \\
\wedge \quad & \bigvee_{a \in I_{Fl}} (ex(a,i) \Rightarrow Fl_i) \\
\wedge \quad & \bigvee_{a \in I_{Fl}} (ex(a,i) \Rightarrow \neg Fl_i) \\
\wedge \quad & \bigvee_{a \in I_{Fl}} (ex(a,i) \wedge Fl_{i-1} \Rightarrow Fl_i) \\
\wedge \quad & \bigvee_{a \in I_{Fl}} (ex(a,i) \wedge \neg Fl_{i-1} \Rightarrow \neg Fl_i)
\end{aligned}
$$

$${}_l[\![\neg Fl]\!]^i_k := \neg\ {}_l[\![Fl]\!]^i_k$$

정의 8의 $[\![M]\!]_k$을 식 (9)로 바꾸고, 정의 10을 적용하면, 바운드 $k \geq 1$와 주어진 LTS L, 그리고 FLTL 식 f에 대한 CNF 식을 생성할 수 있다.

4. 구현 및 실험

본 장에서는 앞장에서 설명한 인코딩 방법을 적용해서 만든 LTS-BMC 도구에 대해 개략적으로 설명한다. 그리고 몇몇 실험을 통해서 기존의 모델 체킹 도구들과 비교분석한다. 이 도구에서 사용하는 모델링 언어는 FSP(Finite State Process)이다. FSP는 일종의 프로세스 계산식(process calculi)으로써 영국 임페리얼 컬리지에서 자바 언어의 쓰레드를 모델하고 분석하기 위해 고안한 모델링 언어이다[1]. FSP는 LTS를 보다 편리하게 기술할 수 있게 해 준다.

구현된 도구의 구조는 아래 그림 2와 같이 파싱, 인코딩, SAT 처리기, 디코딩으로 크게 4단계로 이루어진다. 파싱 단계에서는 FSP를 입력 받아서 각각의 컴포넌트에 대한 LTS를 얻어낸다. 그렇게 얻어낸 LTS들은 인코더를 통해서 SAT 처리기의 입력인 CNF 형태로 변환된다. SAT 처리기는 기존에 개발된 것 중 성능이 우수한 Minisat[8]을 사용했다. 사용한 Minisat의 버전은 1.14이다. SAT 처리기의 결과가 만일 불만족인 경우 바운드를 증가하거나 에러를 찾지 못하고 종료하게 되고, 만족하는 경우는 에러를 발견한 것이므로 SAT 처리기의 결과가 디코더를 통해서 해석 된다.

우리는 LTS-BMC에 대해서 두 가지 실험을 하였다. 먼저[1]에 제시된 예제를 가지고, FSP 언어를 입력 받

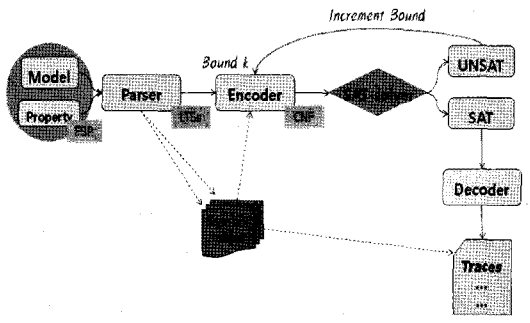


그림 2 LTS-BMC 구조

는 명시적 모델 체커인 LTSA와 성능을 비교했다. 사용된 LTSA의 버전은 2.3이다. 그 다음 [17]에 의해서 제기된 병행 프로그램의 극단적 인터리빙의 예를 가지고 SPIN, Cadence SMV, NuSMV와 비교 실험하였다. 실험은 1Gb 메모리와 1.86GHz CPU에서 동작하는 윈도우 XP에서 수행되었고 timeout은 1000초였다. 실험에 사용한 SPIN의 버전은 5.1.4이고, NuSMV는 버전 2.4.3을 사용했다. SPIN은 비 동기적으로 동작하는 병행 모델을 검증하는 명시적 모델 체커이므로 비교에 의의가 있고, BDD(Binary Decision Diagram) 기반의 Cadence SMV와 NuSMV의 BMC와 비교하였다.

4.1 LTSA와의 비교

LTS-BMC 도구를 이용해서 모델 체킹한 결과가 표 1과 같다. 표에서 k는 바운드를 의미한다. 각각의 예제는 식사하는 철학자(DiningPhil5~10)와 SingleLane, 그리고 Semaphore에 대한 예제로 [1]에서 FSP로 기술된 예제들을 참조하였다. 식사하는 철학자와 SingleLane 예제에서는 데드락 속성을 검사하였고, Semaphore 예제에서는 도달성 검사를 수행했다. 표 1의 결과를 보면 기존의 FSP 분석도구인 LTSA에 비해서 LTS-BMC가 월등한 성능을 보였다. 특히 LTSA에서는 식사하는 철학자가 9명 이상일 경우 timeout과 더불어 상태폭발이 발생했다.

표 1 LTSA와 LTS-BMC의 비교

	LTSA	LTS-BMC			
	time	k	#var	#clause	time
Semaphore	0.063	4	205	873	0.004
SingleLane	0.094	2	264	975	0.004
DiningPhil5	0.140	10	1,826	13,123	0.008
DiningPhil6	0.625	12	2,587	20,538	0.016
DiningPhil7	7.406	14	3,480	30,236	0.020
DiningPhil8	77.563	16	4,505	42,511	0.028
DiningPhil9	timeout	18	5,662	57,657	0.104
DiningPhil10	timeout	20	6,951	75,968	0.112

4.2 다른 모델 체커와 비교

병행적으로 동작하는 프로세스의 행위의 복잡성을 보여주는 극적인 예가 [17]에서 기술한 극단적인 인터리빙이다. 그림 3은 두 개의 프로세스가 공유변수 n을 증가시키는 프로그램의 일부이다. 여기서 n을 증가시키는 부분인 'n:=n+1'은 실제로 값을 읽고(*load*), 증가(*increment*)시킨 후, 저장(*store*)하는 3단계로 수행된다.

위의 프로그램이 종료한 후의 공유 변수 n의 값은 k보다 크거나 같고 $2k$보다 작거나 같을 것으로 예상 된다. 만일 두 개의 프로세스가 순차적으로 실행된다면 공유변수의 값은 $2k$가 된다. 그리고 다음과 같이 완벽한 인터리빙으로 수행된다면 n의 값은 k가 된다.

```
process P1;
var I: Integer;
begin
  for I := 1 to k do
  n := n + 1;
end;
```

```
process P2;
var I: Integer;
begin
  for I := 1 to k do
  n := n + 1;
end;
```

그림 3 단순한 병행 프로그램의 예

1. process P1 : *load n*
2. process P2 : *load n*
3. process P1 : *increment*
4. process P2 : *increment*
5. process P1 : *store n*
6. process P2 : *store n*

그런데 실제 공유 변수의 값은 k 보다 작은 값을 가질 때도 있고 심지어는 공유 변수 n의 값이 2가 되는 극단적인 경우도 있다. 이러한 경우는 시뮬레이션으로 찾기가 어렵기 때문에, 반드시 철저한 검증이 요구된다. 우리는 그림 3과 같은 간단한 프로그램에 대해서 Promela와 SMV 입력언어 그리고 FSP로 모델링 하였다. 그리고 프로세스의 수를 2개에서 6개까지 늘리면서 처리 시간과 검증에 사용된 메모리의 양을 비교하였다. 서로 다른 모델링 언어를 가지는 도구들을 비교하는 것은 매우 어려운 일이다. 그래서 최대한 동일한 구조를 가지도록 모델링 하였다. 실제로 바운드 모델 체킹에 사용된 NuSMV의 바운드수가 LTS-BMC의 바운드 보다 하나 더 적게 모델링 되었다. 그림 4는 검증 시간(왼쪽)과 사용된 메모리(오른쪽)에 대한 결과이다. 세로 축은 각각 검증에 사용된 시간과 메모리이고 가로 축은 프로세스의 수이다. 실험 결과 4가지 도구에서 모두 n=2 가 되는 경로를 찾을 수 있었다. SPIN에서는 `assert(n>2)` 구문을 속성으로 사용해서 도달성 검사를 했고, BDD를 사용한 Cadence SMV에서는 CTL 속성 `AG(p1.control=end & p2.control=end -> (n>2))`를 사용했다. 바운드 모델 검사를 위해서 NuSMV에서는 LTL 속성 `G(p1.control=end & p2.control=end -> (n>2))`를 썼고, LTS-BMC에서는Fluent LTL 속성을 다음과 같이 기술했다. SPIN을 제외한 나머지 도구들은 프로세스가 2개인 경우에 해당하는 속성이다.

```
Test = (end -> {a, b}.print[2] -> STOP).
||C_Test = (a:Proc || b:Proc || Test).
fluent F = <{{a,b}.print[2]}>
assert A = <>F
```

위에서 기술된 `Test`는 검증을 위해 사용된 LTS이다. `Test`가 `a:Proc`(첫 번째 프로세스)와 `b:Proc`(두 번째 프로세스)와 결합되면 오직 end 액션이 발생한 후에 print[2]액션을 수행 할 수 있게 된다. `F`는 print[2] 액션이 수행될 때 참이 되는 변량이고 `A`는 "언젠가 F이다."라는 LTL 속성을 기술한 `assert` 구문이다. NuSMV에서 바운드 모델 체킹을 수행하기 위해서 우리는 아래와 같이 NuSMV의 interaction 모드를 사용했다.

```
NuSMV > set input_file p2.smv
NuSMV > set bmc_length 19
NuSMV > set bmc_loopback 18
NuSMV > set sat_solver MiniSat
NuSMV > go_bmc
NuSMV > check_ltlspec_bmc
```

그림 4의 그래프에서 보이는 바와 같이 이 예제에 대해서는 NuSMV의 BMC와 LTS-BMC가 가장 좋은 결과를 보였다. SPIN과 BDD를 사용한 Cadence SMV는 프로세스를 증가시킴에 따라 메모리의 사용량이 지수적으로 증가해서 프로세스가 5개인 경우 1Gbyte의 메모리를 모두 소모하고도 해결되지 않았다. NuSMV의 경우 프로세스가 5일 경우무려 46분 27초가 지나서야 해결되었다. 반면, LTS-BMC의 경우 모두 1000초 이내에서 반례를 구할 수 있었고, 메모리도 100Mbyte 이내에서 해결되었다.

그러나 모델 체킹의 결과는 해결하고자 하는 문제에 따라서 크게 달라질 수 있다. LTS-BMC는 다수의 프로세스가 인터리빙으로 동작하는 상황을 모델링하고 검증하는데 좋은 성능을 보였다. 그러나 표 2에서 드러난 것과 같이 LTS-BMC는 같은 문제를 풀이하는데 NuSMV

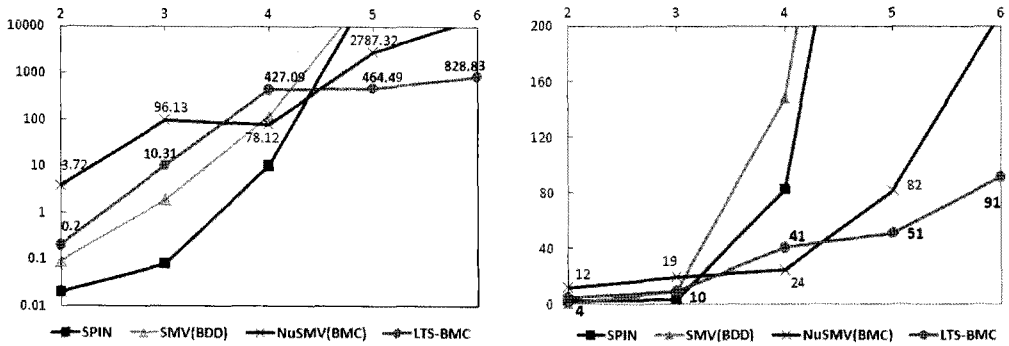


그림 4 극단적인 인터리빙 검증 결과: 왼쪽-시간(초), 오른쪽-메모리(Mb)

표 2 LTS-BMC와 NuSMV가 만들어낸 CNF 식에 대한 비교

# Proc	LTS-BMC			NuSMV(BMC)		
	k	#variables	#clauses	k	#variables	#clauses
2	20	5,523	16,882	19	1,439	4,366
3	29	11,520	35,771	28	2,945	9,572
4	38	19,695	61,662	37	4,959	16,722
5	47	30,048	94,555	46	7,443	25,986
6	56	42,579	134,450	55	10,412	37,284

보다 더 많은 변수와 절이 필요했다. 이것은 LTS-BMC에서는 정수를 표현할 수 없기 때문에 이를 대치하기 위해서 변수를 프로세스로 모델링했기 때문에 발생한 현상이지만, 보다 적은 변수와 절을 갖는 CNF 식을 만들도록 개선할 필요하다.

위 표에서 #Proc는 프로세스의 개수이고, k는 바운드 수, #variables는 변수의 수 #clauses는 절의 수를 나타낸다. 비록 LTS-BMC가 NuSMV에 의해서 생성된 CNF 식보다는 큰 식을 만들어 내지만, 메모리를 더 적게 사용하면서 더 빨리 해결되었다. 이에 대한 정확한 분석은 SAT-solver 내부의 동작 방식을 정확히 알 때 가능하겠지만, CNF 시각화 도구 DPvis[18]를 통해서 두 모델 체커가 만들어내는 CNF 식에 대한 직관을 얻을 수 있다. LTS-BMC는 그림 5의 왼쪽과 같이 원의 형태 구조를 만들고 NuSMV(오른쪽)가 만들어내는 식의 구조는 애벌레 모양이다. 그림에서 노드 변수 아크는 절을 나타낸다.

프로세스의 수가 늘어날수록 LTS-BMC에 의해서 만들어진 식은 더 꽉 찬 원의 형태가 되며, NuSMV에 의해서 만들어진 식은 더 길쭉한 벌레모양이 유지된다. 이것으로 미루어 볼 때, 비록 NuSMV가 만들어 내는 식의 크기가 더 작지만, 그 구조는 SAT-solver로 하여금 더 많은 처리 시간을 요구한다고 생각할 수 있다. 그리고 많은 절수가 오히려 SAT-solver에게 더 많은 정보를 주어서 처리 속도를 향상시킬 수 있다. 그러나 여전히 LTS- BMC의 인코딩 방식은 개선이 필요한 것으로 보인다.

5. 결론 및 향후 연구

모델 체킹의 한계인 상태 폭발의 문제를 극복하려는 노력의 일환으로 BMC가 연구되고 있다[4,5]. 본 논문은 병행 시스템을 모델링 할 때 많이 사용되는 LTS에 대한 BMC 도구를 개발하고 몇몇 예제로 실험하였다. LTS-BMC는 LTL식을 검사하기 위해서 액션을 기반으로 하는 FLTL[1,11]의 인코딩 방법을 정의했다. 또한 LTL식을 해석하기 위해서 새로운 LTS 인코딩 기법을 제안했고, SAT 처리 속도의 개선을 위해서 LTS 인코딩에 최적화된 이진 기수조건[10]을 사용했다. 실험을 통해서 기존의 분석 도구인 LTSA[9]가 검사할 수 없었던 규모의 모델을 검사할 수 있었고, 극단적인 인터리빙 예제를 통해서 LTS-BMC가 기존의 모델 체킹 도구보다 좋은 성능을 낼 수 있음을 보였다.

향후에는 LTS에서 LTL식을 빠르게 검사하는 인코딩 기법[14,15]과 함께 LTS에 대한 바운드 없는 바운드 모델 체킹에 대한 연구도 진행 되어야 한다. 그리고 [12-14]에서 제안했던 효율적인 LTS 검증 방법들을 LTS-BMC 도구에 구현하는 것과 정수 및 산술 연산식을 포함하도록 모델을 확장하고 인코딩 방법을 고안하는 연구들이 남아있다.

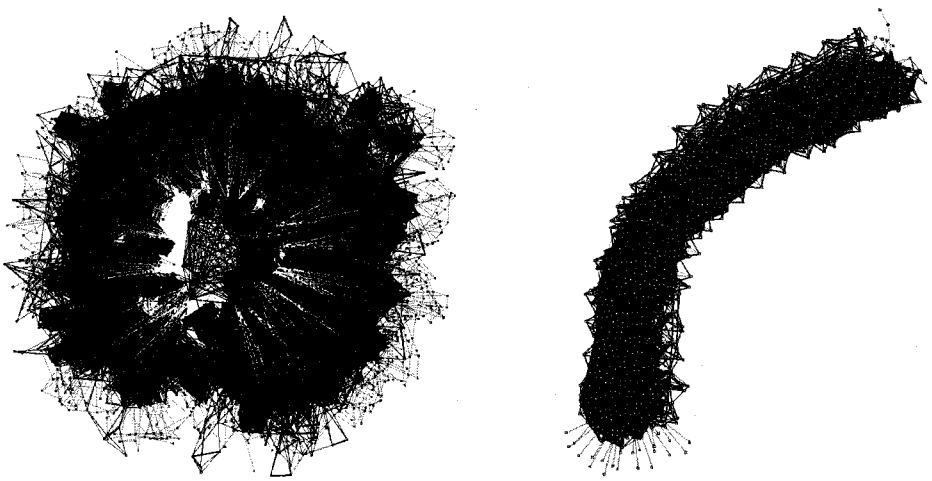


그림 5 LTS-BMC와 NuSMV에 의해서 생성된 CNF 식의 시각화(DPvis)

참 고 문 헌

[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] J. P. Quielle and J. Sifakis, "Specification and verification of concurrents systems in CESAR," In Proceedings of the 5th International Symposium of Programming, pages 337-350, 1981.

[4] 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.

[5] 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).

[6] Marques-Silva, J. P., and Sakallah, K. A., "GRASP: A Search Algorithm for Propositional Satisfiability," IEEE Transactions on Computers, Vol.48, 506-521, 1999.

[7] 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.

[8] http://www.cs.chalmers.se/Cs/Research/For-mal-Methods/MiniSat/

[9] http://www.doc.ic.ac.uk/ltsa/

[10] C. Sinz, "Towards an optimal CNF encoding of Boolean cardinality constraints," In the Proceedings of CP 2005, pp. 827-831, Vol.3709, LNCS, 2005.

[11] D. Gannakopoulou and J. Magee, "Fluent Model Checking for Event-based Systems," In Proceedings of ESEC/FSE03, 2003.

[12] 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.

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

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

[15] A. Cimatti, M. Pistore, M. Roveri, and R. Sebastiani, "Improving the Encoding of LTL Model Checking into SAT," In Proceedings of the 3rd VMCAI, Vol.2294 of LNCS, Springer-Verlag, 2002.

[16] T. Latvala, A. Bere, K. Heljanko, and T. junttila, "Simple Bounded LTL Model Checking," In Proceedings of the 5thVMCAI, Vol.2937 of LNCS, Springer-Verlag, 2004.

[17] M. Ben-Ari and A.Burns, Extreme Interleavings, IEEE Concurrency, Vol.6, No.3, pp. 90-91, July 1998.

[18] C. Sinz and E.-M. Dieringer, "DPvis - a tool to visualize structured SAT instances," In Proceedings of SAT 2005, pp. 257-268, Vol.3569, LNCS, 2005.

박 사 천

2001년 경기대학교 전산학과(학사). 2003년 경기대학교 전산학과(석사). 2004년~현재 경기대학교 전산학과 박사과정. 관심분야는 모델 체킹, 정형 기법, 소프트웨어 공학

권 기 현

정보과학회논문지 : 소프트웨어 및 응용 제 35 권 제 2 호 참조