병행 프로그램에서 극단적 인터리빙의 정형 분석

박사천, 권기현
경기대학교 컴퓨터과학과
{sachem, khkwon}@kgu.ac.kr

Formal Analysis of the Extreme Interleaving on Concurrent Program

Sachoun Park, Gihwon Kwon
Department of Computer Science, Kyonggi University

요 약

프로그램을 작성할 때 시스템의 성능을 높이기 위해서, 여러 개의 프로세스가 동시에 동작하고 상호작용하는 병행 프로그램을 사용한다. 병행 프로그램에서 프로세스들은 인터리빙 방식으로 동작하는 경우가 많다. 그런데 인터리빙을 잘 못 이해할 경우 프로그램에 오류가 생길 가능성이 높고 이러한 오류는 직관적으로 납득하지 못할 때도 많다. 본 논문에서는 인터리빙이 발생하는 최악의 경우를 소개하고 이를 SMV 와 Spin 그리고 LTS-BMC 로 정형 분석하는 방법에 대해서 설명한다. 또한 실험을 통해서 우리가 만든 LTS-BMC 도구가 병행 오류 검증에 효과적임을 알 수 있었다.

1. 서론

여러 프로세스로 구성된 병행 프로그램은, 외부적으로 볼 때 프로세스들이 동시에 수행되는 것처럼 보이지만 내부적으로는 인터리빙(interleaving) 방식으로 수행된다. 그런데 프로세스들이 인터리빙으로 실행된다면 가능한 행위 수가 크게 증가한다. 만일 길이가 n, m 인 두 프로세스가 인터리빙으로 동작하는 경우 서로 다른 행위 수는 $(m+n)!/(m!*n!)$ 이다. 예를 들어 두 프로세스로 구성된 (그림 1)의 병행 프로그램은 공유변수 n 을 증가시킨다. 여기서 n 을 증가하는 문장인 'n:=n+1'은 값을 읽고(load), 증가(increment)하고, 저장(store)하는 3 단계로 수행된다고 가정한다.

```
process P1;                    process P2;
var I: Integer;                var I: Integer;
begin                          begin
  for I := 1 to k do             for I := 1 to k do
  n := n + 1;                    n := n + 1;
end;                           end;
```

(그림 1) 병행 프로그램의 예

프로그램이 종료한 후의 공유 변수 값을 예측하면 $k \le n \le 2k$ 이다. 만일 두 프로세스가 순차적으로 실행된다면 $n = 2k$ 이다. 그리고 다음과 같이 완벽한 인터리빙으로 수행된다면 $n = k$ 이다.

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*

이 논문은 경기도지역협력연구센터(콘텐츠융합소프트웨어 연구센터)의 지원을 받아 수행된 연구임

그런데 실제 $n < k$ 인 경우도 있고 심지어는 $n = 2$ 인 극단적인 경우도 있다[1]. 이와 같은 극단적인 경우를 자세히 살펴보면 다음과 같다. 맨 처음 프로세스 P1 이 공유변수 n 의 값을 읽은 후 P2 가 n 의 값을 k-1 번 증가한다. 다시 P1 이 n 값을 1 로 저장한 후 P2 가 이 값을 읽어온다. 계속해서 P1 이 n 의 값을 k 까지 증가시킨다. 이때까지도 P2 는 n 의 값을 1 로 알고 있기 때문에 마지막으로 P2 의 증가 및 저장이 수행 될 때 n 값은 2 가 된다.

(그림 1)에서 $n = 2$ 인 극단적인 경우는 테스팅 또는 시뮬레이션을 수백 번 반복한다 해도 쉽게 발견할 수 없다. 이들 기법은 시스템이 갖는 전체 행위 중에서 일부분만을 관찰하기 때문에 이들로는 시스템 전체 행위를 조사할 수 없다. 따라서 인터리빙으로 동작되는 병행 프로그램을 분석하기 위해서는 일부분의 행위가 아닌 전체 행위를 조사할 수 있는 분석 기법이 요구된다. 테스팅 또는 시뮬레이션과는 달리 모델체킹은 시스템의 가능한 모든 행위를 조사한다. 모델체킹은 조사할 시스템의 모델과 속성을 입력 받아 모델이 속성을 만족하는지 여부를 자동으로 분석해 주는 도구로서 SMV[2], Spin[3] 등이 널리 사용된다.

본 논문에서는 병행 프로그램 분석에서 중요한 인터리빙의 이해를 돕기 위해서 (그림 1)의 병행 프로그램을 SMV 와 Spin 으로 각각 모델링하고 프로세스의 수를 증가시키면서 어떠한 성능을 보이는지 실험했다. 프로세스 수가 아무리 증가하더라도 위와 같은 공유변수의 값이 2 가 되는 극단적인 인터리빙이 존재함을 알 수 있었다. 그런데 앞의 두 도구는 검증을 위해서 각각의 프로세스들을 병렬 결합하기 때문에 프로세스의 수가 증가할수록 탐색해야 하는 상태공간의 수가 지수적으로 증가하게 된다. 이러한 문제를 해결

하기 위해서 우리가 만든 도구인 LTS-BMC 로도 이를 모델링 하여 앞의 두 도구와 비교 실험했다[4]. 그 결과, 앞의 두 도구와는 달리 인터리빙으로 나타날 수 있는 모든 경우를 병렬 결합 없이 나타낼 수 있었다. 따라서 여러 프로세스가 결합되는 모델에서 기존 도구보다 더 나은 결과를 보였다.

논문 구성은 다음과 같다. 2 장에서는 모델에 대해 살펴보고, 3 장에서는 극단적인 인터리빙 예를 각 모델링 언어로 기술한다. 그리고 4 장에서는 실험 결과를 기술하고, 마지막으로 5 장에서 결론을 맺는다.

2. 모델 체킹과 모델 체킹 도구

모델 체킹은 앞서 언급한 바와 같이 시스템을 철저하게 검사한다. 모델 체킹의 입력은 모델과 속성인데, 모델은 유한 상태 기계로 작성되고, 시스템에서 만족되어야 하는 속성은 시제논리로 기술된다. 그런데 사용되는 시제논리가 CTL (Computation Tree Logic)인지 LTL(Linear Temporal Logic) 인지에 따라 모델 체킹 기술은 크게 두 가지로 분류된다. CTL 을 사용하는 대표적인 도구는 SMV 인데 이 도구는 모델을 BDD(Binary Decision Diagram)로 표현하고 고정점 계산으로 속성의 만족성 여부를 판정한다. 주로 하드웨어 시스템을 검증하는데 많이 적용되었고, 현재는 소프트웨어 모델 체킹을 위해서 SAT 기술을 사용하는 바운드 모델 체킹 방식이 적용되고 있다[5].

LTL 을 사용하는 대표적인 도구는 Spin 이고 주로 프로토콜이나 멀티 스레드 모델과 같이 비동기적인 시스템을 검증하는데 많이 적용되고 있다. LTL 모델 체킹은 모델과 속성의 부정을 모두 오토마타로 변환하고 이 둘에 대한 곱 오토마타를 생성해서 인식되는 언어가 있는 검사한다. 만일 인식되는 언어가 있다면, 속성이 위배되는 경우를 발견한 것이 된다. 인식되는 언어의 존재 여부는 곱 오토마타에서 도달 가능한 사이클을 찾는 방식으로 판정한다. 특히 Spin 의 경우 오토마타에 대한 C 코드를 만들어 검사함으로써 C 컴파일러의 효율성을 이용했다.

최근에 모델 체킹의 기술의 흐름은 소프트웨어를 검증 대상으로 다루기 위해서 탐색하는 상태 공간에 제약을 가하는 바운드 모델 체킹이 많이 연구되고 있다. 바운드 모델 체킹은 SAT 기술을 사용해서 검사하려는 길이를 미리 정해 놓고 그에 해당하는 상태와 전이 정보를 CNF 형태의 식으로 펼쳐놓게 된다. 이렇게 만들어진 CNF 식은 SAT 의 입력이 되어서 해당 식을 만족시키는 모델의 존재여부를 판정 받게 된다. 만일 모델이 존재한다면, 식을 구성하는 각각의 변수들이 어떤 값을 갖는지 알려준다. CBMC[6]를 비롯하여 많은 바운드 모델 체커들이 있는데 우리는 FSP(Finite State Process)를 입력 받아서 바운드 모델 체킹 하는 LTS-BMC 를 만들었다.

3. 인터리빙 모델

실제 1 장에서 살펴보았던 극단적인 인터리빙의 동작을 Spin 모델 체커의 입력언어인 Promela 로 기술하면 (그림 2)와 같다. P()라는 프로세스는 1 장에서 설명한 하나의 프로세스를 의미하고, 전역변수 n 을 3 번 증가시키는 역할을 한다. init 구문은 두 개의 동일한 프로세스를 동작시키는 역할을 하고, _nr_pr 은 실행되는 프로세스의 개수를 의미하는 변수이다. 따라서 (_nr_pr == 1)은 main 프로세스만이 실행되고 있는 상황을 의미하고 이것은 곧 다른 프로세스들이 모두 종료된다는 조건을 표현한다.

```
#define N 3
byte n = 0;

proctype P(){
        byte temp;
        byte i=1;
        do
        :: i > N -> break
        :: else ->
                temp = n;
                n = temp + 1;
                i++
        od
}

init{
        atomic{
                run P(); run P()
        }
        (_nr_pr == 1);
        assert(n > 2)
}
```

(그림 2) Spin 모델

SMV 에서는 지역변수에서 값을 보관하기 위해 Spin 과 달리 프로세스 P()에서 전역변수의 값을 증가시키는 부분을 load, inc, store 로 나누어서 기술해 주어야 한다. 사실상 SMV 는 상태기계를 표현한 것이 되기 때문에 어떤 면에서 SMV 의 코딩은 마치 어셈블리 언어를 작성하는 것과 비슷한 면이 있다.

```
MODULE main
  VAR
        p1 : process P (n);
        p2 : process P (n);
        n  : 0..6;
  ASSIGN
        init(n) := 0;
  SPEC
  AG (p1.control=end & p2.control=end -> (n>2))

MODULE P(n)
  VAR
        i : 1..9;
        control : {load, inc, store, end};
        temp : 0..6;

  ASSIGN
        init(i) := 1;
        next(i) := case
                i<9 : i+1;
                1 : i;
        esac;
        init(control) := load;
        next(control) := case
           control = load & i < 9: inc;
           control = inc & i < 9: store;
           control = store & i < 9: load;
           1 : end;
        esac;
        init(temp) := 0;
        next(temp) := case
           control = load : n;
           control = inc & temp < 6: temp + 1;
           1 : temp;
        esac;
        next(n) := case
           control = store : temp;
           1 : n;
        esac;
```

(그림 3) SMV 모델

마지막으로 LTS-BMC 의 입력언어는 액션을 기반으로 하기 때문에 어떠한 액션이 발생하는지를 중심으로 모델을 기술한다. 그리고 같은 이름의 액션으로 동기화 하는 수단을 삼는다. 따라서 프로세스 P 는 read, inc, 그리고 write 의 액션을 차례대로 수행하는데, read 혹은 write 액션을 수행하면, 전역변수를 모델링 한 프로세스 VAR 에서도 해당하는 액션이 동시에 발생하게 된다. LTS-BMC 모델이 SMV 와 비슷한 이유는 두 모델 다 상태 기계를 표현하기 때문이다. 아래 그림에서 bound 20 은 사용자에 의해서 입력된 바운드 모델 체킹의 탐색 깊이이다.

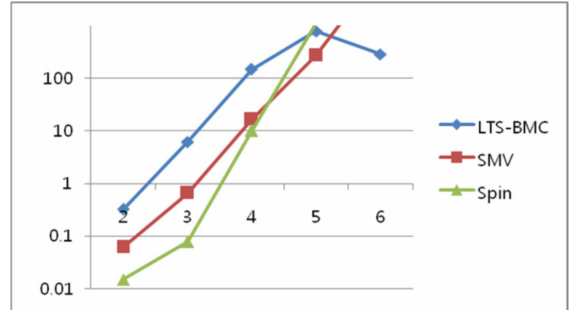
```
bound 20
const Imax = 3
range Int = 0..Imax
set VarAlpha = {read[Int], write[Int]}

VAR(Init = 0) = VAR[Init],
VAR[v:Int] =      (read[v] ->VAR[v]
                  |print[v]->VAR[v]
                  |write[c:Int]->VAR[c]).

P = P[0],
P[i:Int] = ( when(i<Imax)
                read[j:0..Imax-1] ->
                inc ->
                write[j+1] -> P[i+1]
           | when (i == Imax)
                end -> END)+{write[0]}.

||C = ({a,b}::VAR||a:P||b:P)/{end/{a,b}.end}.

Test = (end -> {a}.print[2] ->STOP).

||C_Test = (C || Test).

fluent F = <{{a,b}.print[2]}>
assert A = <> F
```

(그림 4) LTS 모델

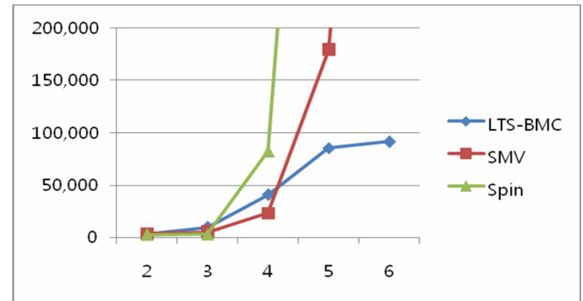
SMV 에서 속성은 두 프로세스가 모두 종료했을 때, “전역변수의 값은 항상 2 보다 크다”라는 의미를 갖는 CTL 논리식으로 기술했다. 이것은 Spin 에서 프로세스의 수가 1 개가 되었을 때 assert(n>2)라고 한 것과 동일하다. LTS-BMC 에서는 프로세스의 수행이 종료되었다는 것을 Test 프로세스와 C 프로세스를 결합한 ||C_test 에서 end 액션이 발생하는 것으로 나타내고, 플루언트 LTL 논리식을 사용해서 종료 후 전역변수의 값이 2 가 될 수 있다는 것을 표현했다.

4. 실험 및 분석

앞 장에서 설명한 각 모델들을 프로세스의 숫자를 증가시키면서 실험해 보았다. 실험은 1.66GHz 듀얼 CPU, 1GB 메모리에서 수행되었다. 앞서 언급한 바와 같이 프로세스의 수를 늘리더라도 세 모델에서 모두 전역변수의 값은 2 가 될 수 있다. 그림 5 는 수행시간을 나타낸다. 초반에는 Spin 이 가장 빨리 끝났으나, 점점 SMV 가 더 좋은 성능을 보였다. 그러나 그림 6 에서 볼 수 있는 것과 같이 Spin 과 SMV 는 프로세스의 수가 증가함에 따라서 메모리 사용량이 지수적으로 증가함으로 인해서 각각 4 개와 5 개의 프로세스들이 결합되는 경우까지만 해결할 수 있었다. 이에 비해서 LTS-BMC 는 프로세스를 결합하지 않고 발생되는 전연 액션에 의해서 각각의 프로세스의 지역 액션들이 발생하도록 인코딩 했기 때문에 프로세스의 수가 증가되어도 비교적 적은 메모리에서 검증을 수행할 수 있었다. 그러나 프로세스의 수가 증감함에 따라서 바운드도 증가하기 때문에 6 개의 프로세스까지만 해결할 수 있었다.



(그림 5) 검증 시간(초)



(그림 6) 메모리 사용량(KB)

5. 결론 및 향후 연구

시스템의 성능을 생각한다면 프로그램을 병행적으로 작성할 수 밖에 없다. 그런데 병행 프로그램은 에러를 발생시킬 가능성이 너무나 많다. 길이가 각각 *n* 와 *m* 인 두 개의 프로세스가 인터리빙으로 동작하는 경우 $(m+n)!/(m!*n!)$의 서로 다른 행위들이 존재한다. 따라서 프로세스들의 병행 결합은 검증을 매우 어렵게 한다. 이를 잘 처리하기 위해서 우리는 모델을 결합하지 않고 주어진 바운드 내에서 각각의 프로세스들이 동기를 맞추어 동작하게 하는 LTS-BMC 를 개발했다. 극단적인 인터리빙의 예를 가지고 프로세스를 늘리면서 검증한 실험에서 LTS-BMC 는 SMV 나 Spin 보다 좋은 성능을 보였다.

참고문헌

[1] M. Ben-Ari and A. Burns, “Extreme Interleavings,” IEEE Concurrency, vol.6, no.3, pp.90-91, July 1998.

[2] K. L. McMillan, Symbolic Model Checking, PhD thesis, Carnegie Mellon University, 1993.

[3] G. Holzmann, The SPIN Model Checker, Addison-Wesley, 2003.

[4] S. Park and G. Kwon, “Bounded model checking for LTS” In the Proceedings of KSEJW, vol. 5, no. 1, 2007.

[5] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu, “Symbolic model checking without BDDs,” In the Proceedings of the DAC, pp. 317-320, ACM press, 1999.

[6] E. Clarke, D. Kroening, and F. Lerda, “A tool for checking ANSI-C programs,” In the Proceedings of the TACAS, pp. 168-176, Springer, 2004.