Programming Assignment 2
AAA528, Spring 2025

Hakjoo Oh

Problem 1 *Z3를 이용해서 프로그램 자동 검증기(Automatic Program Verifier)를 구현해보자. 모델 체킹(Bounded Model Checking)이라 불리우는 프로그램 검증 기법을 이용하여 동시 프로그래밍(Concurrent Programming)에서 상호 배제(Mutual Exclusion)가 정확히 이루어지는지 여부를 검증하는 것이 목표이다.*

예를 들어, 아래 프로그램을 생각해보자.

Thread A	*Thread B*
A0. Maybe go to A1.	*B0. Maybe go to B1.*
A1. If l go to A1, else to A2.	*B1. If l go to B1, else to B2.*
A2. Set $l \leftarrow 1$, go to A3.	*B2. Set $l \leftarrow 1$, go to B3.*
A3. Critical, go to A4.	*B3. Critical, go to B4.*
A4. Set $l \leftarrow 0$, go to A0.	*B4. Set $l \leftarrow 0$, go to B0.*

위 프로그램은 스레드 A와 B로 이루어져 있고, 각 스레드는 다섯개의 상태(State)로 구성되어 있다. 스레드 A의 다섯 상태를 A0, A1, A2, A3, A4, 스레드 B의 상태를 B0, B1, B2, B3, B4로 표시하였다. 각 상태는 명령문을 포함하고 있는데, 프로그램에서 사용할 수 있는 명령문을 아래 네 가지 종류로 제한하자:

- *"Maybe go to s": 상태 s로 이동할 수 있다는 뜻이다. 단, 이동하지 않고 현재 상태에 머물러 있어도 된다.*
- *"If v go to s_1, else to s_0": 공유 변수(shared variable) v의 값이 true이면 s_1으로 이동하고, false이면 s_0로 이동한다. v는 항상 부울 변수(boolean variable)이다.*
- *"Set $v \leftarrow b$, go to s: 공유 변수 v의 값을 b로 변경하고 상태 s로 이동한다.*
- *"Critical, go to s": 임계 구역(Critical section)을 뜻한다. 상태 s로 이동한다.*

초기 상태 (A0, B0)에서 시작하여 두 스레드가 동시에 임계 구역에 진입할 수 있는지 여부를 검증하는 것이 목표이다. 두 스레드가 임계 구역에 진입할 수 있는 경우가 존재한다면 가장 간단한 시나리오를 반례(Counterexample)로 생성한다. 위 프로그램의 경우 상호 배제가 정확히 이루어지지 않는 오류가 있는 프로그램이며 따라서 다음과 같이 오류 발생 시나리오(error trace)를 출력해야 한다.

$$(0,0) \rightarrow (0,1) \rightarrow (1,1) \rightarrow (1,2) \rightarrow (2,2) \rightarrow (3,2) \rightarrow (3,3)$$

위 프로그램의 경우 총 6단계를 거쳐야 해당 오류가 발생한다. 5단계 이하의 실행에서는 오류가 발생하지 않는다.

Examples

1. *위에서 예로 든 프로그램의 경우 다음과 같은 텍스트 파일로 입력 받는다.*

```
30
5
1
```

```
--thread1--
0 maybe 1
1 if 0 1 2
2 set 0 1 3
3 critical 4
4 set 0 0 0
--thread2--
0 maybe 1
1 if 0 1 2
2 set 0 1 3
3 critical 4
4 set 0 0 0
```

가장 첫 줄의 숫자는 검증에 고려할 최대 단계 수이다. 위의 경우 30단계 이하의 프로그램 실행에서 상호 배제가 이루어지는지 검증하라는 뜻이다. 이와 같이 검증할 프로그램의 실행 단계를 제한하는 프로그램 검증 기법을 제한된 모델 체킹(Bounded Model Checking)이라 부른다. 두 번째 숫자(5)는 두 스레드가 가지는 상태의 개수이다. 두 스레드는 항상 동일한 개수의 상태로 구성된다고 가정한다. 세 번째 숫자(1)은 공유 변수의 개수이다. 위 예제의 경우 공유 변수를 하나만 사용하고 있다. 명령문 "Maybe go to A1"은 `maybe 1`*으로 나타내었다. 공유 변수는 0, 1, 2와 같이 숫자로 표시한다. 위의 경우 사용할 수 있는 공유 변수는 1개 뿐이므로 0으로 나타낸다. 명령문* `if 0 1 2`*는 공유 변수 0의 값이 true이면 상태 1로 이동하고 아니면 2로 이동하라는 뜻이다. 비슷하게* `set 0 1 3`*은 공유 변수 0의 값을 1(true)로 변경하고 상태 3으로 이동하라는 뜻이다.* `critical 4`*는 임계 구역을 뜻하고 상태 4로 이동하라는 뜻이다.*

위의 입력을 받으면 다음과 같이 오류를 검출하고 반례를 생성해야 한다:

```
(0, 0, False)
(0, 1, False)
(1, 1, False)
(1, 2, False)
(2, 2, False)
(3, 2, True)
(3, 3, True)
```

두 스레드의 상태와 더불어 공유 변수의 값을 함께 출력하였다.

2. *아래 프로그램을 생각하자.*

Thread A	*Thread B*
A0. Maybe go to A1.	*B0. Maybe go to B1.*
A1. If b go to A1, else to A2.	*B1. If a go to B1, else to B2.*
A2. Set $a \leftarrow 1$, go to A3.	*B2. Set $b \leftarrow 1$, go to B3.*
A3. Critical, go to A4.	*B3. Critical, go to B4.*
A4. Set $a \leftarrow 0$, go to A0.	*B4. Set $b \leftarrow 0$, go to B0.*

공유 변수를 두 개 사용하여 각 스레드가 임계 구역에 진입하는 경우를 나타내었다. 위 알고리즘은 상호 배제를 보장할까? 그렇지 않다. 위 프로그램은 다음과 같이 입력 파일로 표현할 수 있다.

```
30
5
2
--thread1--
0 maybe 1
1 if 1 1 2
2 set 0 1 3
```

```
3 critical 4
4 set 0 0 0
--thread2--
0 maybe 1
1 if 0 1 2
2 set 1 1 3
3 critical 4
4 set 1 0 0
```

검증기는 아래와 같이 오류를 찾아낼 수 있어야 한다:

```
(0, 0, False, False)
(1, 0, False, False)
(1, 1, False, False)
(2, 1, False, False)
(2, 2, False, False)
(3, 2, True, False)
(3, 3, True, True)
```

3. *위 프로그램에서 단계 1과 2를 서로 바꾸어 보자.*

Thread A	*Thread B*
A0. Maybe go to A1.	*B0. Maybe go to B1.*
A1. Set $a \leftarrow 1$*, go to A2.*	*B1. Set* $b \leftarrow 1$*, go to B2.*
A2. If b go to A2, else to A3.	*B2. If a go to B2, else to B3.*
A3. Critical, go to A4.	*B3. Critical, go to B4.*
A4. Set $a \leftarrow 0$*, go to A0.*	*B4. Set* $b \leftarrow 0$*, go to B0.*

위 프로그램에서 스레드 A와 B는 동시에 임계 구역에 진입할 수 없다. 즉, 상호 배제 측면에서 올바른 프로그램이다. 아래와 같이 입력 파일을 구성해서

```
30
5
2
--thread1--
0 maybe 1
1 set 0 1 2
2 if 1 2 3
3 critical 4
4 set 0 0 0
--thread2--
0 maybe 1
1 set 1 1 2
2 if 0 2 3
3 critical 4
4 set 1 0 0
```

검증기를 실행시키면 다음과 같이 검증에 성공해야 한다:

```
Mutual exclusion is proved.
```

(위 프로그램은 상호 배제 측면에서는 오류가 없지만 두 스레드가 교착상태(deadlock)에 빠질 수 있다는 문제가 있다. A와 B가 모두 상태 2로 진입할 수 있고 서로가 서로를 무한히 기다릴 수 있다. 이번 문제에서는 교착 상태는 고려하지 않겠지만, 검증기를 확장하여 교착 상태를 검출하는 것이 크게 어렵지 않다.)

4. *아래 프로그램은 정확할까?*

Thread A	*Thread B*
A0. Maybe go to A1.	*B0. Maybe go to B1.*
A1. Set $a \leftarrow 1$, go to A2.	*B1. Set $b \leftarrow 1$, go to B2.*
A2. If l go to A3, else to A5.	*B2. If l go to B5, else to B3.*
A3. If b go to A3, else to A4.	*B3. If a go to B3, else to B4.*
A4. Set $l \leftarrow 0$, go to A2.	*B4. Set $l \leftarrow 1$, go to B2.*
A5. Critical, go to A6.	*B5. Critical, go to B6.*
A6. Set $a \leftarrow 0$, go to A0.	*B6. Set $b \leftarrow 0$, go to B0.*

다음과 같이 입력 파일을 만들어서 테스트 해보자.

```
30
7
3
--thread1--
0 maybe 1
1 set 0 1 2
2 if 2 3 5
3 if 1 3 4
4 set 2 0 2
5 critical 6
6 set 0 0 0
--thread2--
0 maybe 1
1 set 1 1 2
2 if 2 5 3
3 if 0 3 4
4 set 2 1 2
5 critical 6
6 set 1 0 0
```

검증기를 실행하면 다음과 같이 반례를 발견할 수 있어야 한다.

```
(0, 0, False, False, False)
(0, 1, False, False, False)
(0, 2, False, True, False)
(1, 2, False, True, False)
(1, 3, False, True, False)
(1, 4, False, True, False)
(2, 4, True, True, False)
(5, 4, True, True, False)
(5, 2, True, True, True)
(5, 5, True, True, True)
```

5. *아래는 상호 배제 문제를 교착 상태(deadlock)나 기아 상태(starvation)없이 올바르게 해결하는 알고리즘의 예이다(G. L. Peterson, 1981).*

Thread A	*Thread B*
A0. Maybe go to A1.	*B0. Maybe go to B1.*
A1. Set $a \leftarrow 1$, go to A2.	*B1. Set $b \leftarrow 1$, go to B2.*
A2. Set $l \leftarrow 0$, go to A3.	*B2. Set $l \leftarrow 1$, go to B3.*
A3. If b go to A4, else to A5.	*B3. If a go to B4, else to B5.*
A4. If l go to A5, else to A3.	*B4. If l go to B3, else to B5.*
A5. Critical, go to A6.	*B5. Critical, go to B6.*
A6. Set $a \leftarrow 0$, go to A0.	*B6. Set $b \leftarrow 0$, go to B0.*

다음과 같이 입력 파일을 만들고

```
30
7
3
--thread1--
0 maybe 1
1 set 0 1 2
2 set 2 0 3
3 if 1 4 5
4 if 2 5 3
5 critical 6
6 set 0 0 0
--thread2--
0 maybe 1
1 set 1 1 2
2 set 2 1 3
3 if 0 4 5
4 if 2 3 5
5 critical 6
6 set 1 0 0
```

검증기를 실행시키면 성공해야 한다.

```
Mutual exclusion is proved.
```

SAT Encoding *첫 번째 프로그램을 생각해보자.*

Thread A	*Thread B*
A0. Maybe go to A1.	*B0. Maybe go to B1.*
A1. If l go to A1, else to A2.	*B1. If l go to B1, else to B2.*
A2. Set $l \leftarrow 1$, go to A3.	*B2. Set $l \leftarrow 1$, go to B3.*
A3. Critical, go to A4.	*B3. Critical, go to B4.*
A4. Set $l \leftarrow 0$, go to A0.	*B4. Set $l \leftarrow 0$, go to B0.*

고려하는 프로그램 실행 단계를 r이라고 할 때, r보다 작은 각 단계 $t(0 \leq t < r)$ 마다 다음의 부울 변수를 생성한다.

$$A0_t, A1_t, A2_t, A3_t, A4_t, B0_t, B1_t B2_t, B3_t, B4_t, l_t$$

예를 들어, $A0_t$가 true이면 단계 t에서 스레드 A가 상태 0에 있음을 뜻한다. l_t가 true이면 공유 변수의 값이 단계 t에서 1임을 뜻한다. 가장 먼저 다음이 성립해야 한다.

$$(\neg A0_t \vee \neg A1_t), (\neg A0_t \vee \neg A2_t), \ldots, (\neg A3_t \vee \neg A4_t)$$

$$(\neg B0_t \vee \neg B1_t), (\neg B0_t \vee \neg B2_t), \ldots, (\neg B3_t \vee \neg B4_t)$$

변수 $@_t$를 도입해서 단계 t에서 실행되는 스레드를 나타낸다고 하자. $@_t$가 true이면 A가 실행되며, $@_t$가 false이면 B가 실행된다는 뜻이다.

프로그램의 초기 상태는 다음과 같이 표현된다.

$$A0_0 \wedge \neg A1_0 \wedge \neg A2_0 \wedge \neg A3_0 \wedge \neg A4_0 \wedge B0_0 \wedge \neg B1_0 \wedge \neg B2_0 \wedge \neg B3_0 \wedge \neg B4_0 \wedge \neg l_0$$

주어진 프로그램의 실행 의미는 다음과 같이 표현할 수 있다($0 \leq t < r$):

$(@_t \vee \neg A0_t \vee A0_{t+1})$	$(\neg @_t \vee \neg A0_t \vee A0_{t+1} \vee A1_{t+1})$	$(@_t \vee \neg B0_t \vee B0_{t+1} \vee B1_{t+1})$
$(@_t \vee \neg A1_t \vee A1_{t+1})$	$(\neg @_t \vee \neg A1_t \vee \neg l_t \vee A1_{t+1})$	$(@_t \vee \neg B1_t \vee \neg l_t \vee B1_{t+1})$
$(@_t \vee \neg A2_t \vee A2_{t+1})$	$(\neg @_t \vee \neg A1_t \vee l_t \vee A2_{t+1})$	$(@_t \vee \neg B1_t \vee l_t \vee B2_{t+1})$
$(@_t \vee \neg A3_t \vee A3_{t+1})$	$(\neg @_t \vee \neg A2_t \vee A3_{t+1})$	$(@_t \vee \neg B2_t \vee B3_{t+1})$
$(@_t \vee \neg A4_t \vee A4_{t+1})$	$(\neg @_t \vee \neg A2_t \vee l_{t+1})$	$(@_t \vee \neg B2_t \vee l_{t+1})$
$(\neg @_t \vee \neg B0_t \vee B0_{t+1})$	$(\neg @_t \vee \neg A3_t \vee A4_{t+1})$	$(@_t \vee \neg B3_t \vee B4_{t+1})$
$(\neg @_t \vee \neg B1_t \vee B1_{t+1})$	$(\neg @_t \vee \neg A4_t \vee A0_{t+1})$	$(@_t \vee \neg B4_t \vee B0_{t+1})$
$(\neg @_t \vee \neg B2_t \vee B2_{t+1})$	$(\neg @_t \vee \neg A4_t \vee \neg l_{t+1})$	$(@_t \vee \neg B4_t \vee \neg l_{t+1})$
$(\neg @_t \vee \neg B3_t \vee B3_{t+1})$	$(\neg @_t \vee l_t \vee A2_t \vee A4_t \vee \neg l_{t+1})$	$(@_t \vee l_t \vee B2_t \vee B4_t \vee \neg l_{t+1})$
$(\neg @_t \vee \neg B4_t \vee B4_{t+1})$	$(\neg @_t \vee \neg l_t \vee A2_t \vee A4_t \vee l_{t+1})$	$(@_t \vee \neg l_t \vee B2_t \vee B4_t \vee l_{t+1})$

마지막으로 검증해야 하는 조건을 추가한다. 스레드 A와 B가 동시에 임계 구역에 있을 수 있는지 여부를 검증하는 것이므로 다음을 추가하면 된다.

$$A3_r \wedge B3_r$$

이와 같이 생성한 논리식이 UNSAT이면 프로그램에 오류가 없다는 뜻이다(r 단계 내에서). SAT이면 오류가 존재한다는 뜻이며 해(satisfying assignment)로부터 오류 경로를 생성할 수 있다.