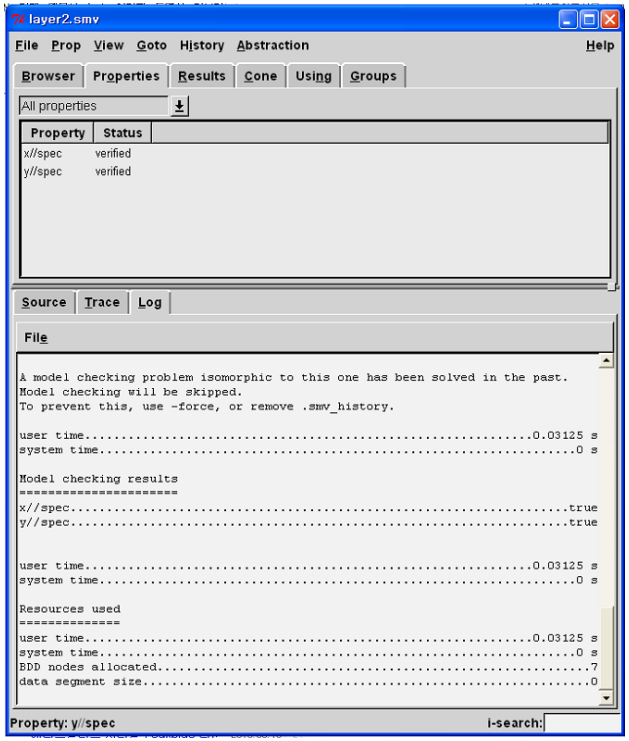
SMV 소개

Konkuk UNIV MBC lab
Park Chulwoo

목차

시스템 소개

- SMV(Symbolic Model Verifier)는 유한 상태 시스템(finite state system)이 CTL(Computation Tree Logic)이라는 논리와 BDD(Binary Decision Diagram)를 이용하여 요구 명세를 만족하는지를 자동적으로 검증하는 정형 검증 도구.
- 심볼릭 모델 체킹 알고리즘을 사용함.
- 모델을 BDD(Binary Decision Diagram)로 표현하고 고정점 계산으로 속성의 만족성 여부를 판정.



CTL(Computation Tree Logic)

- CTL의 연산자들은 주어진 유한 모델(finite model)의 임의의 한 상태(state)가 주어진 논리식(formula)을 만족하는지를 효율적으로 알아 볼 수 있는 fixed point 특성을 가짐.
- CTL에서 시제 연산자는 A또는 E 다음의 *F, G, U, X*의 쌍으로 이루어짐.
- 또는 *E*로 시작하기 때문에 논리식의 참, 거짓은 특정 branch에 해당하는 것이 아니라 특정 상태에 해당함.

CTL

- CTL(Computational tree logic)는 Linear temporal logic에서 사용되는 시제 연산자 G, F, X, U 앞에 path quantifier E, A가 선행되어 논리식을 표현한다.

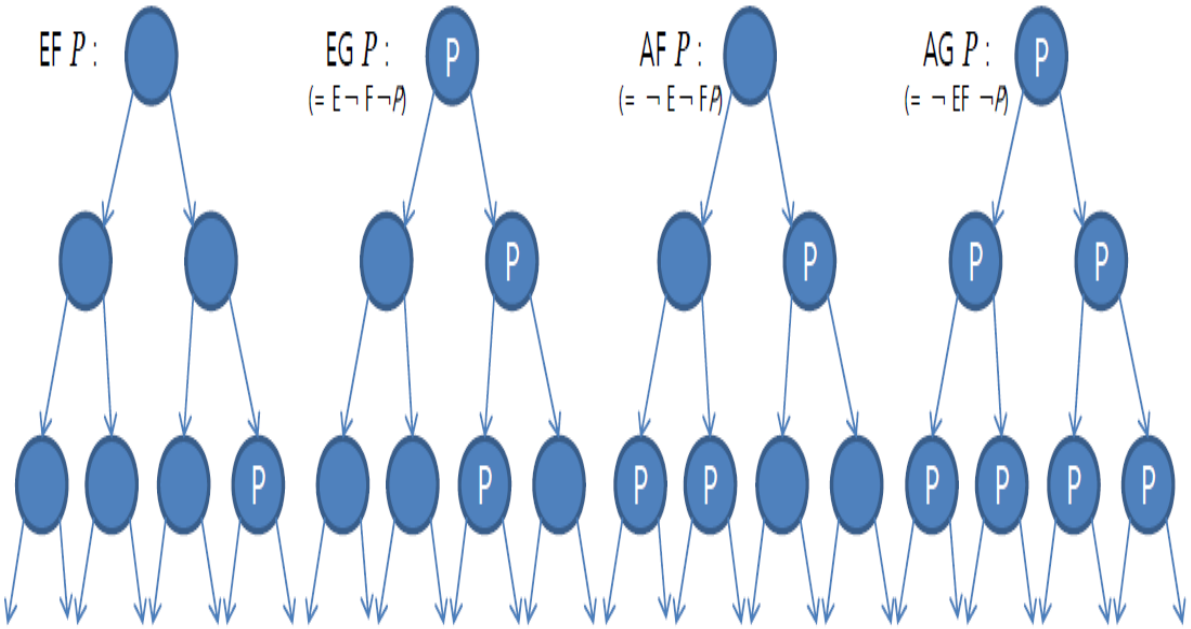
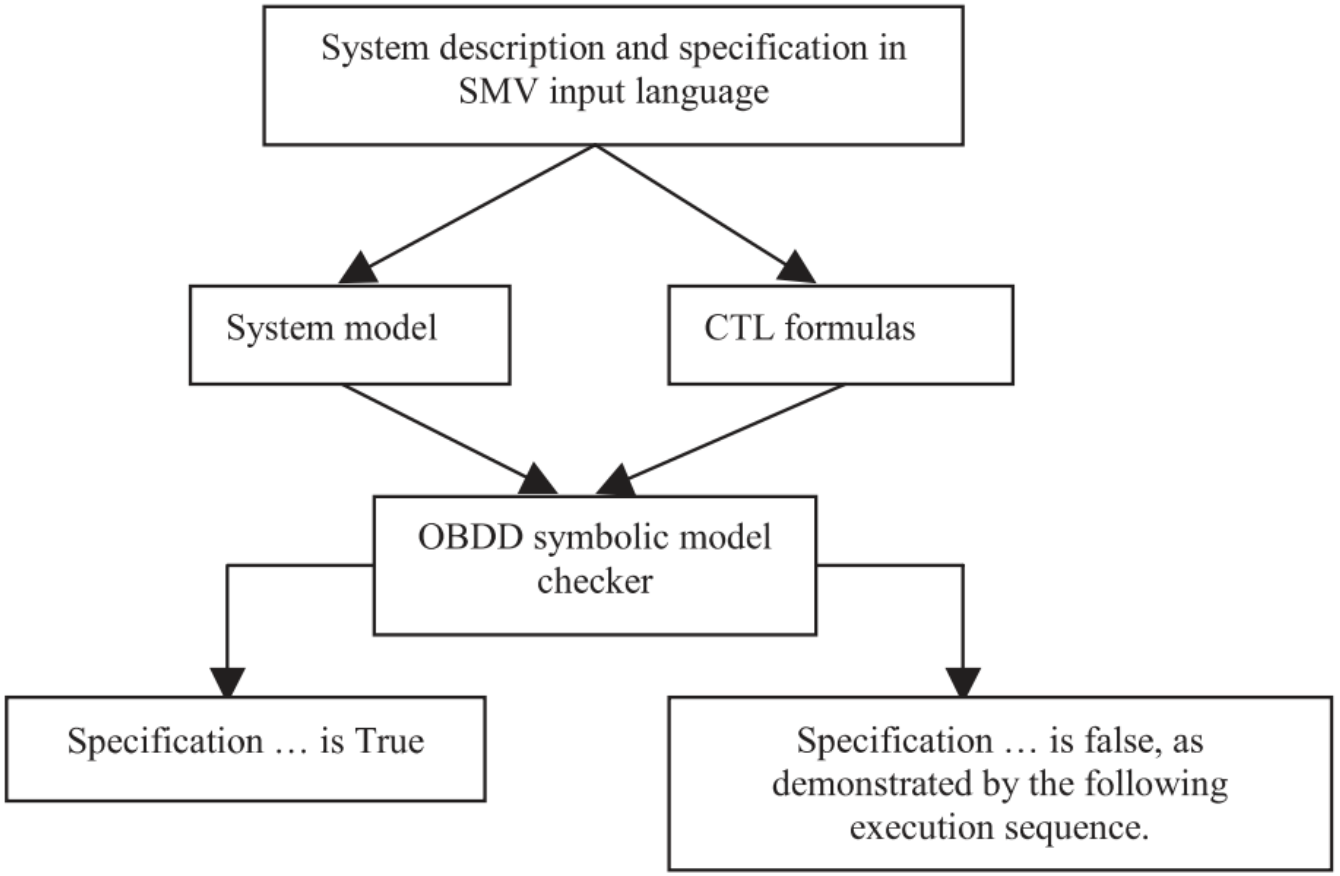


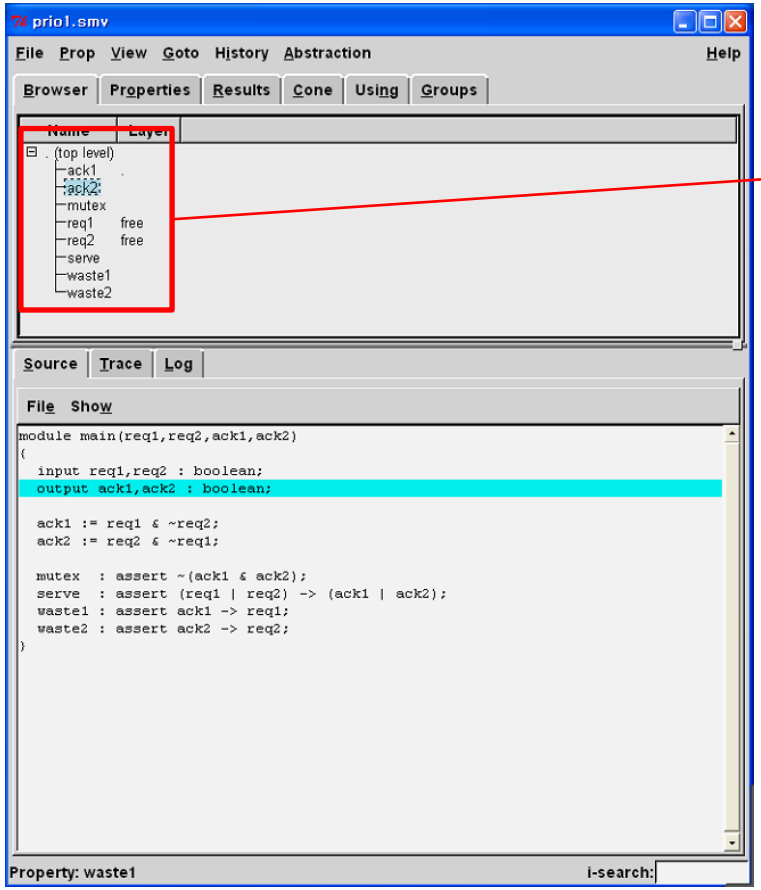
Table 1: Some temporal connectives in CTL

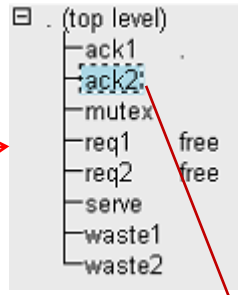
EX φ	**true** in current state if formula φ is **true** in at least one of the next states
EF φ	**true** in current state if there exists some state in some path beginning in current state that satisfies the formula φ
EG φ	**true** in current state if every state in some path beginning in current state satisfies the formula φ
AX φ	**true** in current state if formula φ is **true** in every one of the next states
AF φ	**true** in current state if there exists some state in every path beginning in current state that satisfies the formula φ
AG φ	**true** in current state if every state in every path beginning in current state satisfies the formula φ

SMV 시스템 구조


System description and specification in SMV input language
System model
CTL formulas
OBDD symbolic model checker
Specification … is True
Specification … is false, as demonstrated by the following execution sequence.

SMV



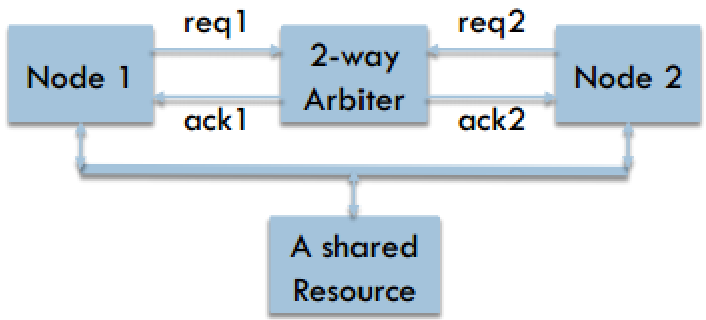


marked

```
module main(req1,req2,ack1,ack2)
(
  input req1,req2 : boolean;
  output ack1,ack2 : boolean;

  ack1 := req1 & ~req2;
  ack2 := req2 & ~req1;

  mutex  : assert ~(ack1 & ack2);
  serve  : assert (req1 | req2) -> (ack1 | ack2);
  waste1 : assert ack1 -> req1;
  waste2 : assert ack2 -> req2;
)
```

Example1 : 2-way Arbiter

```
module main(req1,req2,ack1,ack2)
{
  input req1,req2 : boolean;
  output ack1,ack2 : boolean;

  ack1 := req1 & ~req2;
  ack2 := req2 & ~req1;

  mutex  : assert ~(ack1 & ack2);
  serve  : assert (req1 | req2) -> (ack1 | ack2);
  waste1 : assert ack1 -> req1;
  waste2 : assert ack2 -> req2;
}
```



parameters
req1,req2, ack1,ack2

Mutex : output ack1, ack2가 동시에 만족하지 않는다
Serve : req1,req2가 true이면 output인 ack1또는 ack2도 true여야한다

Property	Result	Time
mutex	true	Fri May 03 10:15:33 ëα î럴誘쉴덩 î묘以 î뫷 2013
serve	false	Fri May 03 10:15:33 ëα î럴誘쉴덩 î묘以 î뫷 2013

Property	Status
mutex	verified
serve	unverified
waste1	unverified
waste2	unverified

SMV STOP

Not all the properties checked

counterexample

	1
ack1	0
ack2	0
req1	1
req2	1

Both input **true**
Both output **true**

↓

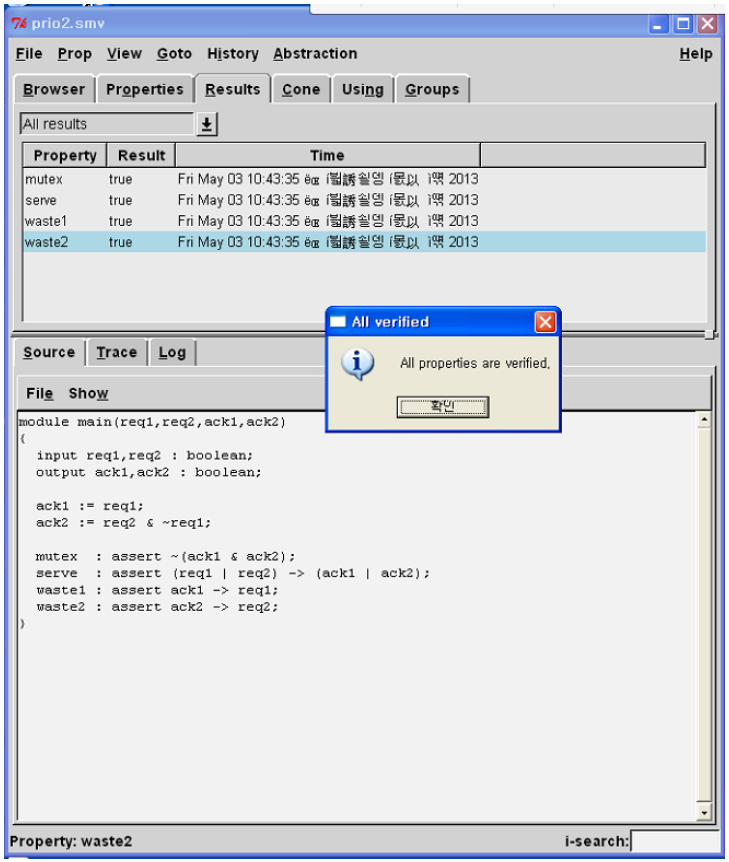
false

Example1 : 2-way Arbiter

```
module main(req1,req2,ack1,ack2)
{
  input req1,req2 : boolean;
  output ack1,ack2 : boolean;

  ack1 := req1 & ~req2;
  ack2 := req2 & ~req1;

  mutex  : assert ~(ack1 & ack2);
  serve  : assert (req1 | req2) -> (ack1 | ack2);
  waste1 : assert ack1 -> req1;
  waste2 : assert ack2 -> req2;
}
```

change

```
module main(req1,req2,ack1,ack2)
{
  input req1,req2 : boolean;
  output ack1,ack2 : boolean;

  ack1 := req1;
  ack2 := req2 & ~req1;

  mutex  : assert ~(ack1 & ack2);
  serve  : assert (req1 | req2) -> (ack1 | ack2);
  waste1 : assert ack1 -> req1;
  waste2 : assert ack2 -> req2;
}
```



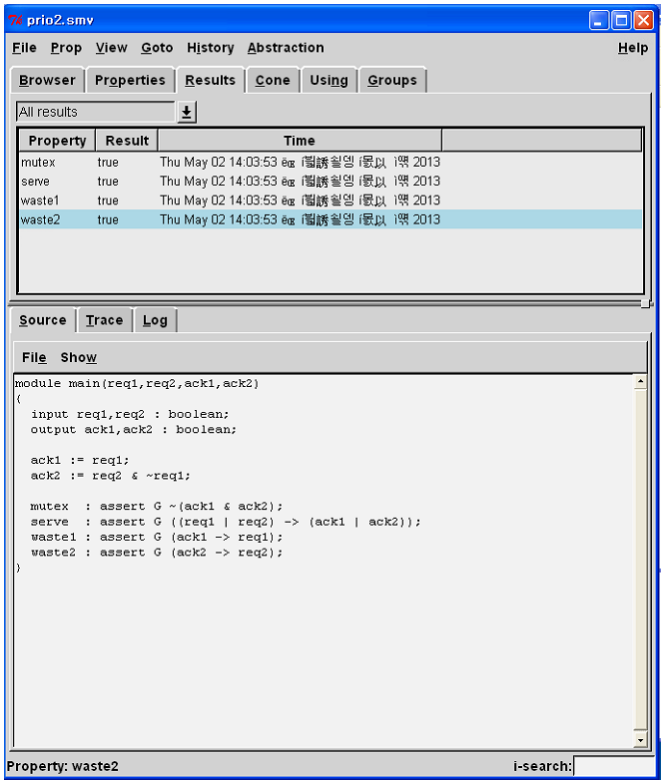
Example1 : 2-way Arbiter

```
module main(req1,req2,ack1,ack2) {
input req1,req2 : boolean;
 output ack1,ack2 : boolean;
 ack1 := req1;
 ack2 := req2 & ~req1;
 mutex : assert G ~(ack1 & ack2);
 serve : assert G ((req1 | req2) -> (ack1 | ack2));
 waste1 : assert G (ack1 -> req1);
 waste2 : assert G (ack2 -> req2);
}
```

Two "grant" signals : ack1, ack2

Signals never assert same time : G ~(ack1 & ack2)

One of req1, req2 is true, then one of ack1, ack2 must be true : G ((req1 | req2) -> (ack1 | ack2))



Example1 : 2-way Arbiter : Starvation Property

```
module main(req1,req2,ack1,ack2)
{
  input req1,req2 : boolean;
  output ack1,ack2 : boolean;

  ack1 := req1;
  ack2 := req2 & ~req1;

  mutex  : assert G ~(ack1 & ack2);
  serve  : assert G ((req1 | req2) -> (ack1 | ack2));
  waste1 : assert G (ack1 -> req1);
  waste2 : assert G (ack2 -> req2);
  no_starve : assert G F (~req2 | ack2);
}
```

Two “grant” signals : ack1, ack2

Signals never assert same time : G ~(ack1 & ack2)

One of req1, req2 is true, then one of ack1, ack2 must be true : G ((req1 | req2) -> (ack1 | ack2))

Property	Result	Time	
no_starve	false	Thu May 02 15:38:21 [illegible] 2013	

	|: 1 :|			
ack2	0			
req1	1			
req2	1			

|: 1 :| first state repeats forever

False

Because : one state should be appear

When both request lines asserted, always prioritize ack1, so Node 2 starves

Property	Status	
mutex	verified	
no_starve	unverified	
serve	unverified	
waste1	unverified	
waste2	unverified	

no_starve : assert G F (~req2 | ack2);

we don't want req2 to be asserted forever while ack2 is never asserted. Put another way, we want it to always eventually be true that either req2 is negated or ack2 is asserted. In temporal logic, we write ``always eventually'' by combining G and F. In this case we assert: G F (~req2 | ack2). Therefore, add the following specification to the program:

2-way Arbiter: Starvation Avoidance

```
module main(req1,req2,ack1,ack2)
{
  input req1,req2 : boolean;
  output ack1,ack2 : boolean;
  bit : boolean;

  next(bit) := ack1;

  if (bit) {
    ack1 := req1 & ~req2;
    ack2 := req2;
  }
  else {
    ack1 := req1;
    ack2 := req2 & ~req1;
  }


  mutex  : assert G ~(ack1 & ack2);
  serve  : assert G ((req1 | req2) -> (ack1 | ack2));
  waste1 : assert G (ack1 -> req1);
  waste2 : assert G (ack2 -> req2);
  no_starve : assert G F (~req2 | ack2);
}
```

to prevent this starvation case

let's add a latch to the circuit that remembers whether ack1 was asserted on the previous cycle. In this case we'll give priority to requester 2 instead. To do this, add the following code to the program:

reverse the priority order

Property	Status
mutex	verified
no_starve	verified
serve	verified
waste1	verified
waste2	verified

Example 2: Traffic Light Controller


```
module main(N_Sense,S_Sense,E_Sense,N_Go,S_Go,E_Go){
  input N_Sense,S_Sense,E_Sense : boolean;
  output N_Go,S_Go,E_Go : boolean;

  NS_Lock, N_Req, S_Req, E_Req : boolean;

  init(N_Go) := 0;
  init(S_Go) := 0;
  init(E_Go) := 0;
  init(NS_Lock) := 0;
  init(N_Req) := 0;
  init(S_Req) := 0;
  init(E_Req) := 0;


  default{
    if(N_Sense) next(N_Req) := 1;
    if(S_Sense) next(S_Req) := 1;
    if(E_Sense) next(E_Req) := 1;
  }
  in default case{
    N_Req & ~N_Go & ~E_Req : {
      next(NS_Lock) := 1;
      next(N_Go) := 1;
    }
    N_Go & ~N_Sense : {
      next(N_Go) := 0;
      next(N_Req) := 0;
      if(~S_Go) next(NS_Lock) := 0; /* unlock if South-going is off */
    }
  }
```

```
  in default case{
    S_Req & ~S_Go & ~E_Req : {
      next(NS_Lock) := 1;
      next(S_Go) := 1;
    }
  S_Go & ~S_Sense : {
      next(S_Go) := 0;
      next(S_Req) := 0;
      if(~N_Go) next(NS_Lock) := 0;
    }
  }
  in case{
    E_Req & ~NS_Lock & ~E_Go : next(E_Go) := 1;
    E_Go & ~E_Sense : {
      next(E_Go) := 0;
      next(E_Req) := 0;
    }
  }

  safety: assert G ~(E_Go & (N_Go | S_Go));

  N_live : assert G (N_Sense -> F N_Go);
  S_live : assert G (S_Sense -> F S_Go);
  E_live : assert G (E_Sense -> F E_Go);

  N_fair : assert G F ~(N_Sense & N_Go);
  S_fair : assert G F ~(S_Sense & S_Go);
  E_fair : assert G F ~(E_Sense & E_Go);

  using N_fair, S_fair, E_fair prove N_live, S_live,
E_live;
  assume N_fair, S_fair, E_fair;
}
```

- The fix: give North light controller responsibility to turn off the lock when both lights are going off

```
if( ~S_Go | /* South already off */
~S_Sense) /* South will go off */
next(NS_Lock) := 0;
```

E_fair	assumed
N_fair	assumed
S_fair	assumed

Example 2: Traffic Light Controller

Safety: light in cross directions never on at same time
safety: assert G ~(E_Go & (N_Go | S_Go));

Liveness: no traffic waits forever
N_live : assert G (N_Sense -> F N_Go);
S_live : assert G (S_Sense -> F S_Go);
E_live : assert G (E_Sense -> F E_Go);

North & South go off same time

Fairness assumptions: car don't wait at green light forever
"always eventually, it is not the case that a car is at a green light"
N_fair : assert G F ~(N_Sense & N_Go);
S_fair : assert G F ~(S_Sense & S_Go);
E_fair : assert G F ~(E_Sense & E_Go);

Setting up the proof
using N_fair, S_fair, E_fair prove N_live, S_live, E_live;
assume N_fair, S_fair, E_fair;

	1	2	3	|: 4 :|
E_Go	0	0	0	0
E_Req	0	0	1	1
E_Sense	0	1	0	0
NS_Lock	0	0	1	1
N_Go	0	0	1	0
N_Req	0	1	1	0
N_Sense	1	0	0	0
S_Go	0	0	1	0
S_Req	0	1	1	0
S_Sense	1	0	0	0

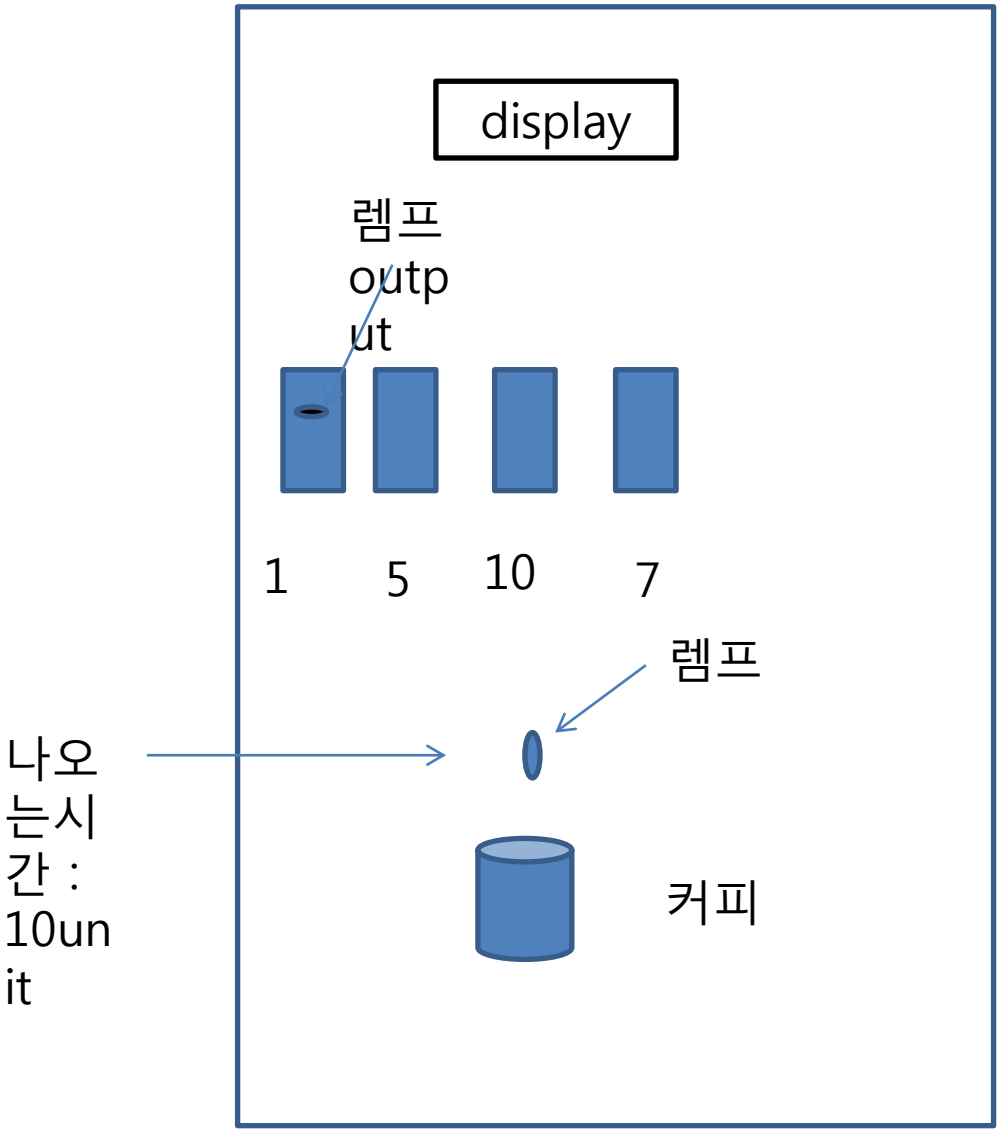
Thank you

동전단위 :I
투입금액 :1~100

Button= 0,1,2,3

Button0_ramp ={0,1}

Display : 코인이들어올 때마다 올라감.
Coin : {1,5,10}
총합은 0~100
Refund 바이너리
Change : 잔돈 0~100




HOT
COLD