Note: content of these slides are from “Getting started with SMV” by K. L. McMillan, 1999. Refer to this document for more details.

SMV TUTORIAL – Part I

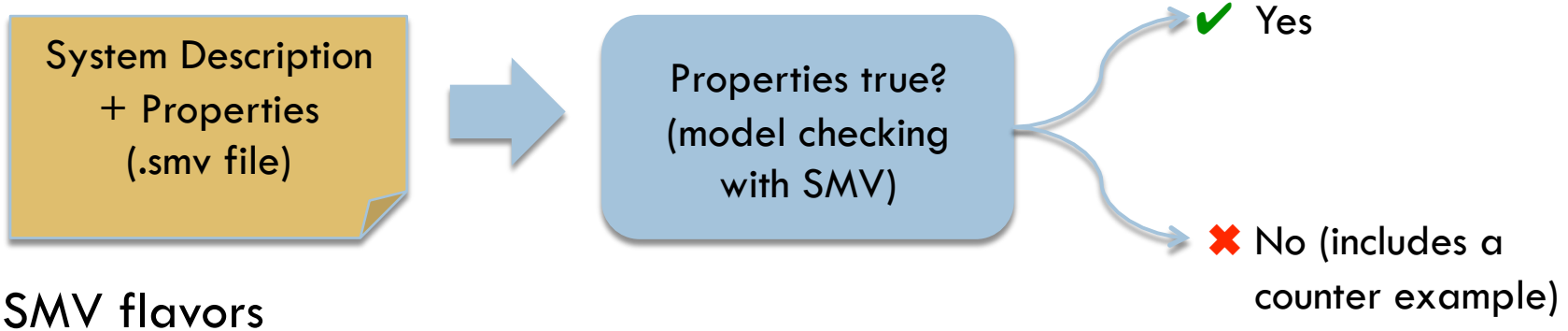
Eriko Nurvitadhi

Agenda

- Part I – SMV Basics (*this talk*)
  - About SMV
  - Example 1: a simple 2-way arbiter
    - Creating an SMV Description
    - Correctness specification with temporal logic operators
    - Running SMV
    - Debugging example
  - Example 2: traffic light controller
    - Using case and default constructs
    - Specifying assumptions
    - Debugging examples
  - Example 3: buffer allocation controller
    - Using iterator and chain constructs
    - BDD limit
- Part II – Compositional Verification

About SMV

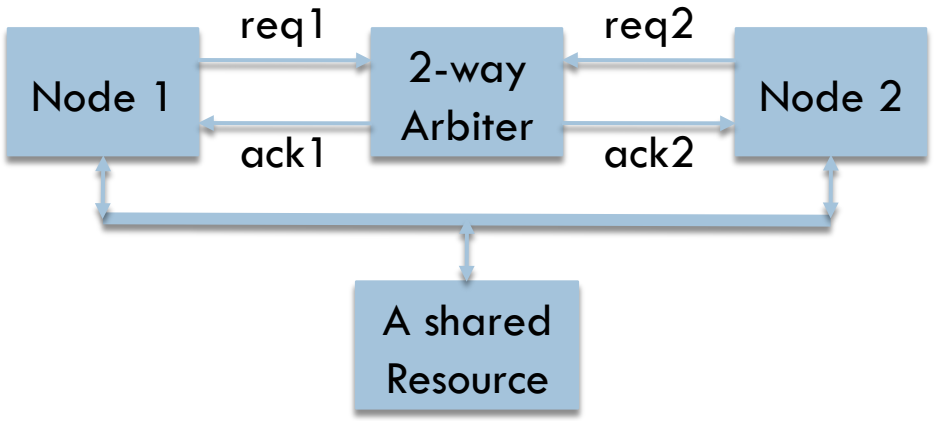
- SMV: Symbolic Model Verifier



- SMV flavors
  - Cadence SMV (*this tutorial*)
    - based on Ken McMillan's work at CMU
    - can do compositional verification
    - http://www.kenmcmil.com/smv.html
  - NuSMV
    - open source
    - http://nusmv.irst.itc.it/

Example 1: 2-way Arbiter

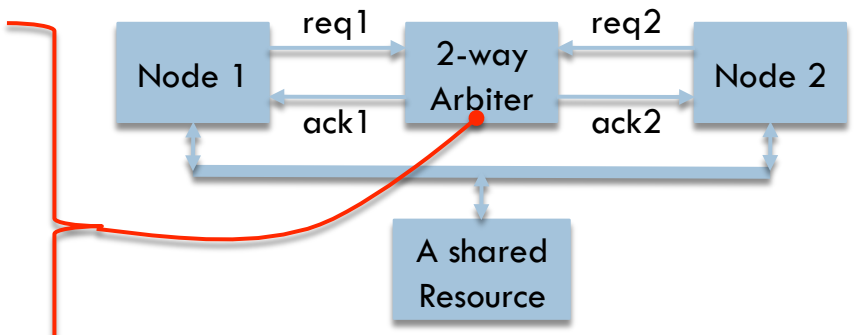
- Arbitrate 2 nodes from accessing a shared resource



- Specification
  - Node request(s) should result in a grant
  - Both grant signals (ack1, 2) never asserted at same time

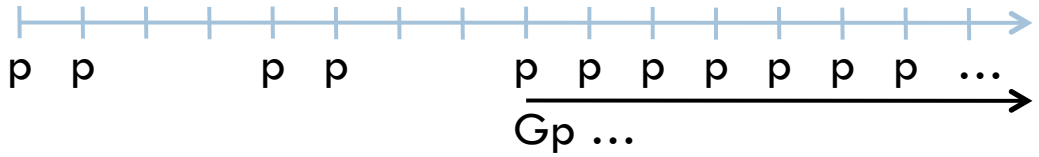
2-way Arbiter: SMV Description


```
module main(req1,req2,ack1,ack2){

   input req1,req2 : boolean;
   output ack1,ack2 : boolean;

   ack1 := req1;  /* priority */
   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);

}
```

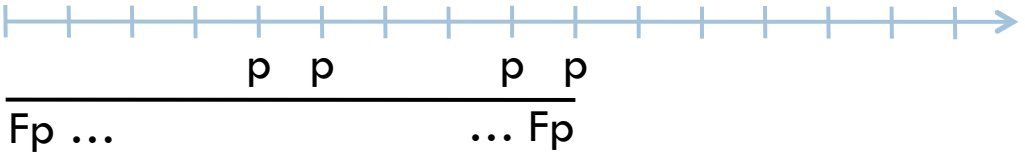


Specifying Correctness Properties with Temporal Logic Operators

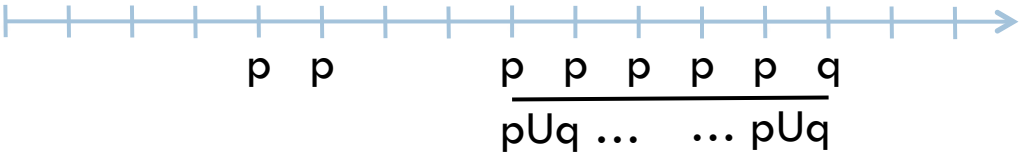
- The "global" operator (e.g., **G** p)



- The "future" operator (e.g., **F** p)



- The "until" operator (e.g., p **U** q)



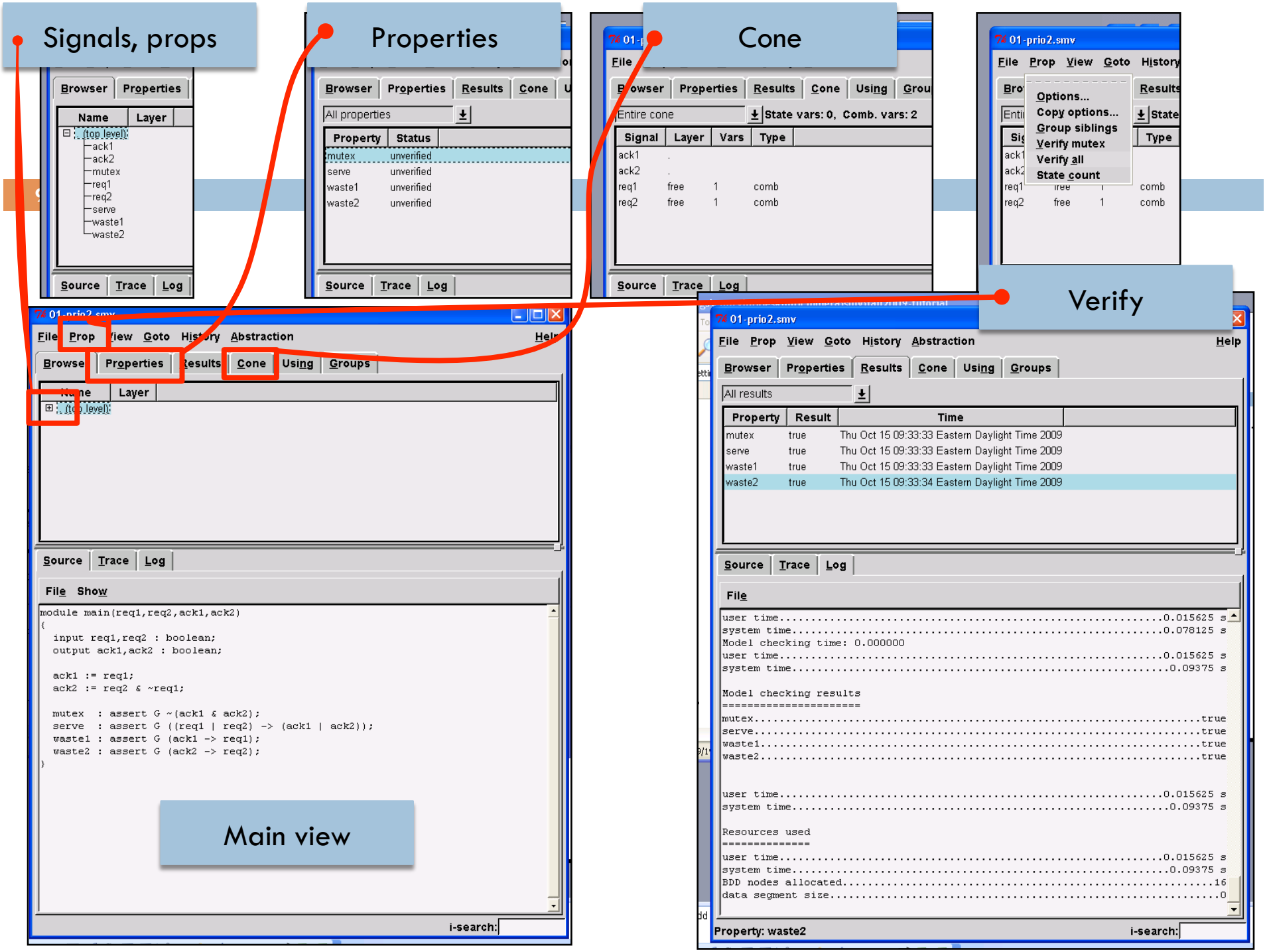
- The "next time" operator (e.g., **X** p)

2-way Arbiter: Correctness Properties


```
module main(req1,req2,ack1,

   input req1,req2 : boolea
   output ack1,ack2 : boole

   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);

}
```

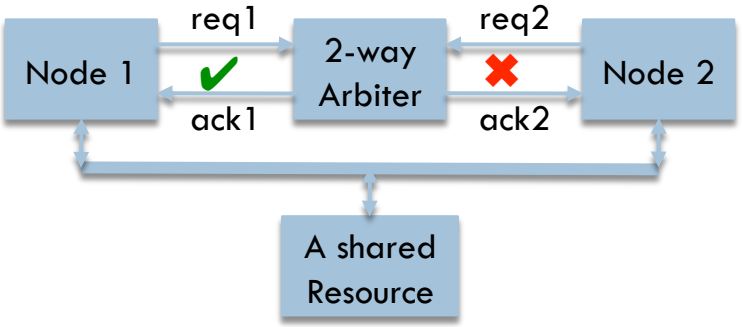
- Specification
  - Node request(s) should result in a grant
  - Both grant signals (ack1, 2) never asserted at same time

Verification with Cadence SMV


Signals, props
Properties
Cone
Verify
Main view
Browser
Properties
Results
Cone
Using
Groups
Name
Layer
(top level)
ack1
ack2
mutex
req1
req2
serve
waste1
waste2
Source
Trace
Log
File
Show
All properties
Property
Status
unverified
Entire cone
State vars: 0, Comb. vars: 2
Signal
Vars
Type
free
comb
Options...
Copy options...
Group siblings
Verify mutex
Verify all
State count
01-prio2.smv
File
Prop
View
Goto
History
Abstraction
Help
All results
Result
Time
true
Thu Oct 15 09:33:33 Eastern Daylight Time 2009
Thu Oct 15 09:33:34 Eastern Daylight Time 2009
Property: waste2
i-search:
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);
}
user time.....0.015625 s
system time.....0.078125 s
Model checking time: 0.000000
user time.....0.015625 s
system time.....0.09375 s
Model checking results
mutex.....true
serve.....true
waste1.....true
waste2.....true
user time.....0.015625 s
system time.....0.09375 s
Resources used
user time.....0.015625 s
system time.....0.09375 s
BDD nodes allocated.....16
data segment size.....0

2-way Arbiter: Starvation Property

- Suppose that the arbiter should avoid starvation
  - i.e., we don't want (lower priority) Node 2 requesting, and never granted access to the shared resource



- For this, we add the following property

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

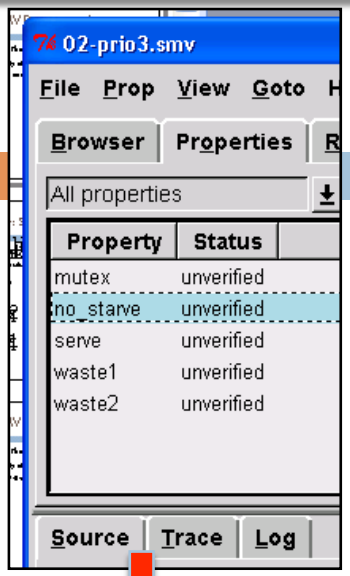
  - i.e., it should "always eventually be true that either `req2` is negated or `ack2` is asserted"

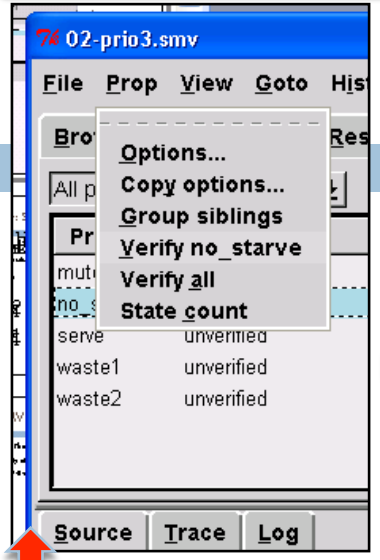
Verifying no_starve

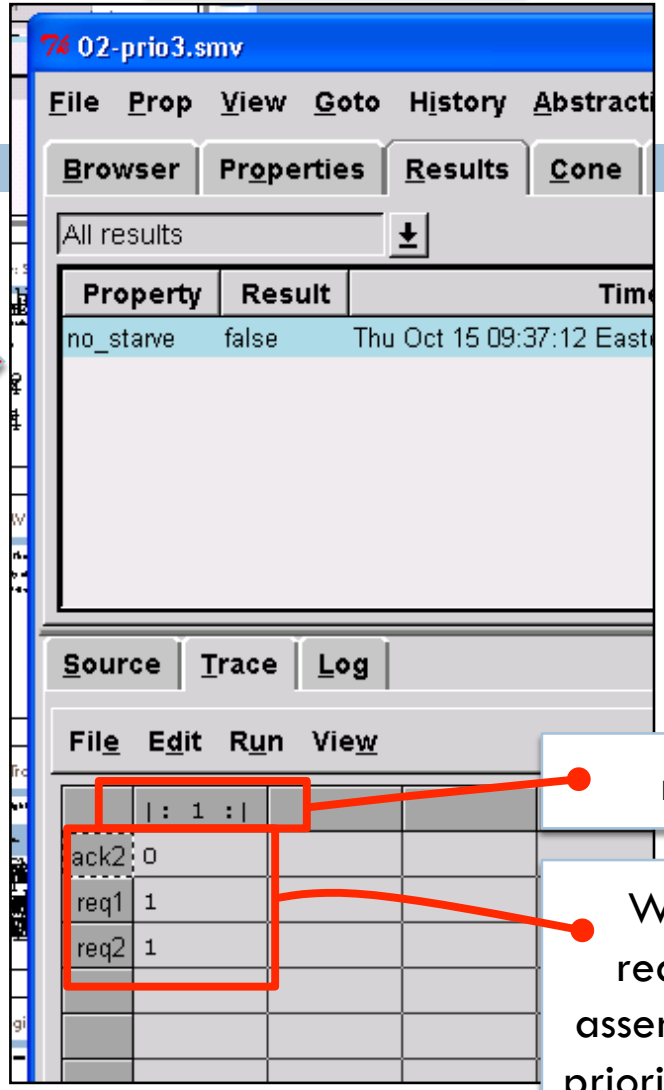
no_starve

verify

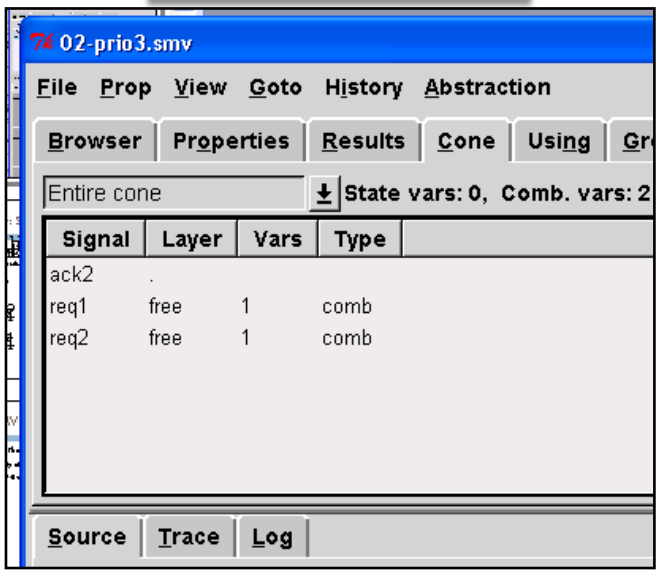
counter example







cone



repeats

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

2-way Arbiter: Starvation Avoidance

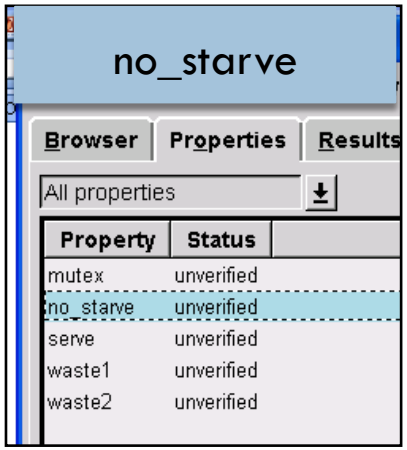
- We can add a state element, `bit`
  - remembers whether `ack1` was asserted previously
  - if so, give priority to Node 2 this time

- The SMV description

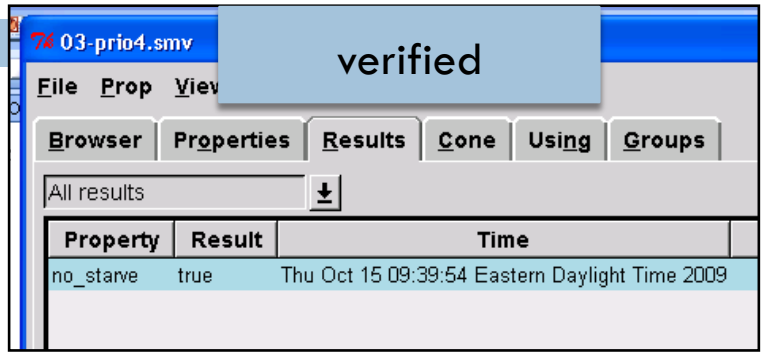
```
bit : boolean;
next(bit) := ack1;

if (bit) {                          /* ack1 asserted previously */
  ack1 := req1 & ~req2;
  ack2 := req2;                     /* prioritize Node 2 */
} else {
  ack1 := req1;                     /* prioritize Node 1 */
  ack2 := req2 & ~req1;
}
```

Re-verifying no_starve

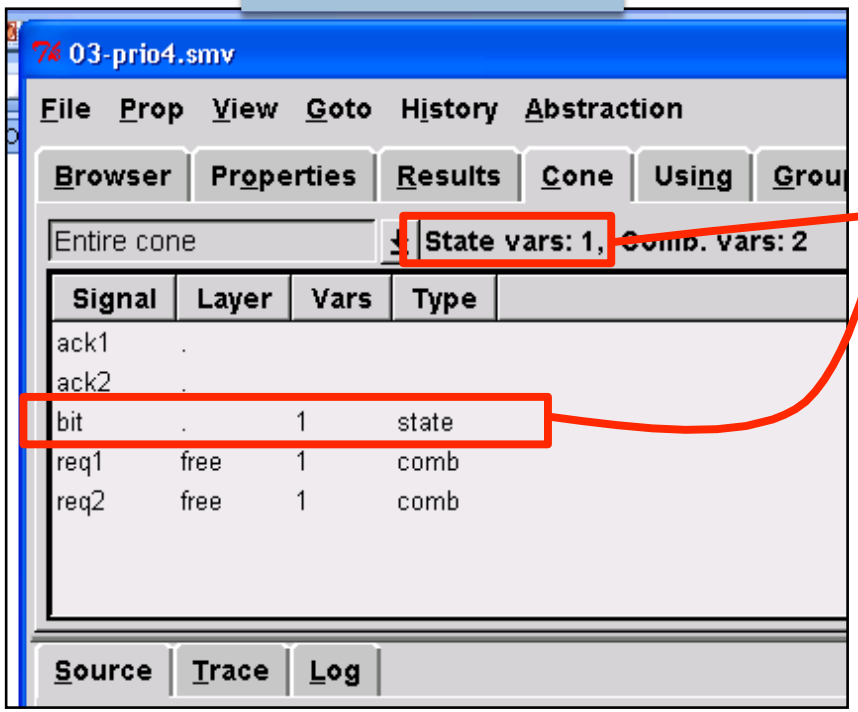
no_starve



verified



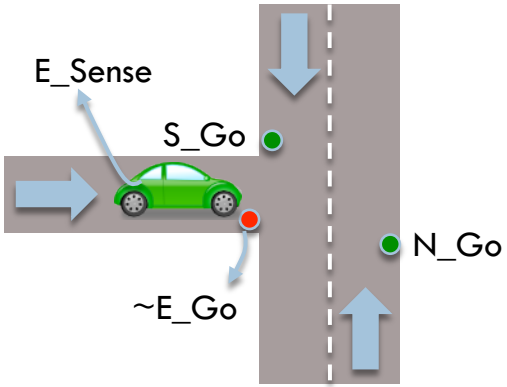
cone



Source of state explosion

Example 2: Traffic Light Controller

- Controls the traffic lights at an intersection




```
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 : boolean;              /* set when north or south traffic enabled */
  N_Req, S_Req, E_Req : boolean; /* remember previous values of traffic sensor inputs */

  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;
```

Controller Logic

Note 1:

```
case{
     cond1 : {block1}
     cond2 : {block2}
     cond3 : {block3}
}
```

Is same as

```
if (cond1) {block1}
else if (cond2) {block2}
else if (cond3) {block3}
```

Note 2:

```
default {block1}
in      {block2}
```

Means `block2` takes precedence over `block1`

- "Remembers" set sense bits

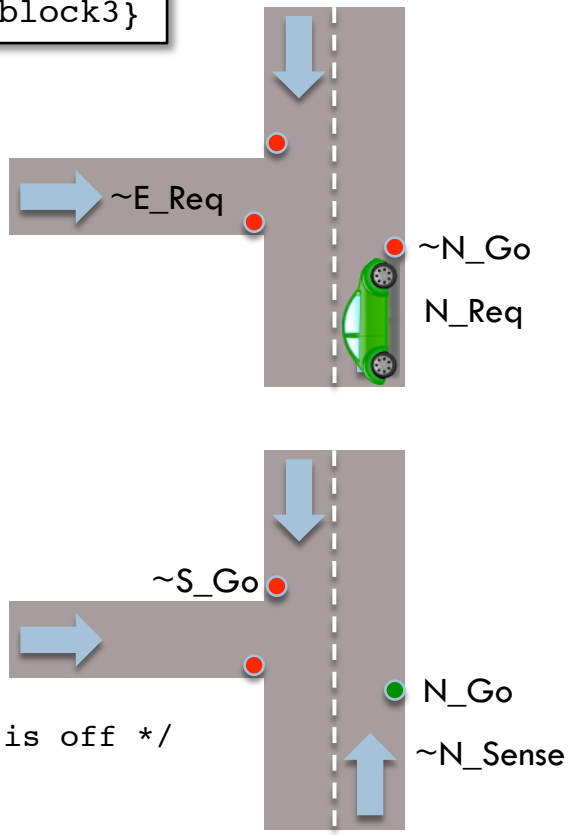
```
default{
   if(N_Sense) next(N_Req) := 1;
   if(S_Sense) next(S_Req) := 1;
   if(E_Sense) next(E_Req) := 1;
 }
```

- North-going light

```
in default case{
   N_Req & ~N_Go & ~E_Req : {
      next(NS_Lock) := 1; /* set lock */
      next(N_Go) := 1;    /* turn on light */
    }
   N_Go & ~N_Sense : {
      next(N_Go) := 0;    /* turn off */
      next(N_Req) := 0;   /* clear history */
      if(~S_Go) next(NS_Lock) := 0;  /* unlock if South-going is off */
    }
 }
```

Prioritize East traffic

Prevent collision with East traffic



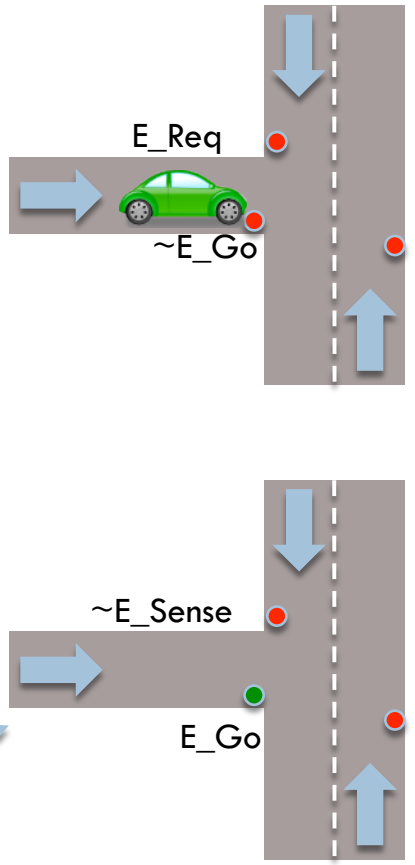
Controller Logic

- South-going light (similar to North-going)

```
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;
  }
}
```

- East-going light

```
in case{
  E_Req & ~NS_Lock & ~E_Go : next(E_Go) := 1; /* turn on */
  E_Go & ~E_Sense : {
    next(E_Go) := 0;   /* turn off */
    next(E_Req) := 0;  /* clear history */
  }
}
```



Specifications and Assumptions

- Safety: light in cross directions never on at same time

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

- Liveness: no traffic waits forever

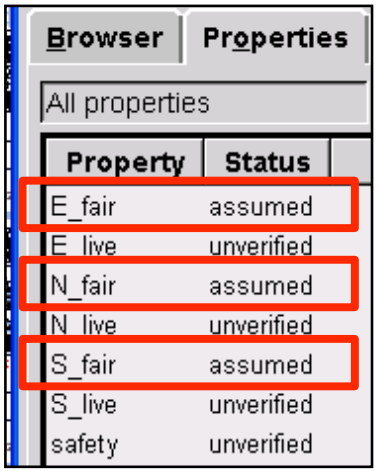
```
N_live : assert G (N_Sense -> F N_Go);
S_live : assert G (S_Sense -> F S_Go);
E_live : assert G (E_Sense -> F E_Go);
```

- 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);
```

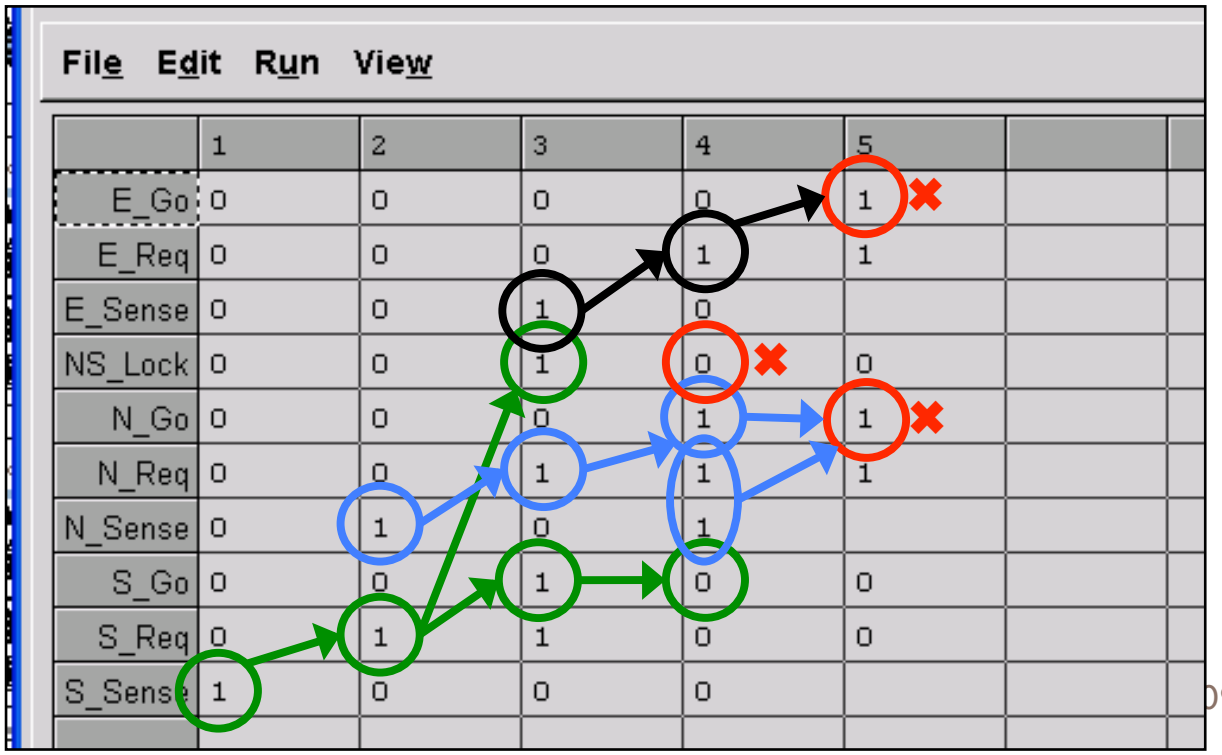
- Setting up the proof

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



Bug 1: Safety

- Cleared lock when south light goes off exactly when north light goes on
  - North light will be on w/o a lock → can cause collision w/ East traffic!
  - Caused by our use of `default`; South code at higher precedence than North



Bug 1: Safety

- Cleared lock when south light goes off exactly when north light goes on
  - North light will be on w/o a lock → can cause collision w/ East traffic!
  - Caused by our use of `default`; South code at higher precedence than North

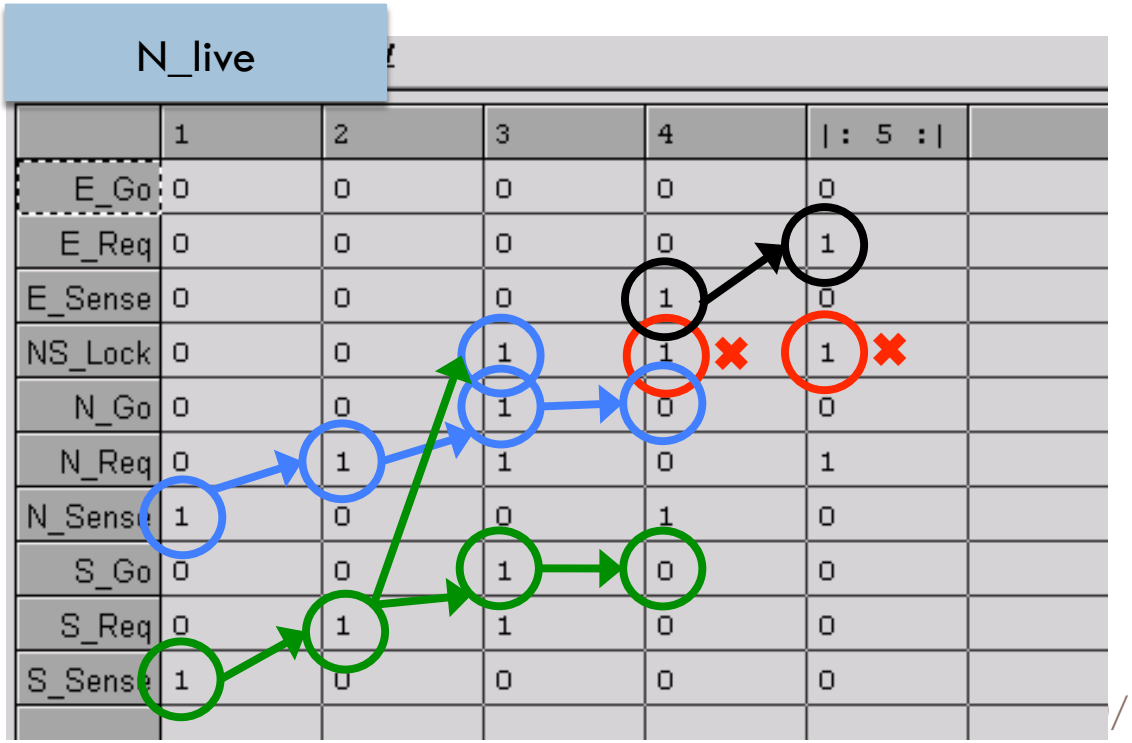
```
/* Original South light controller code */
 in default case{
    S_Req & ~S_Go & ~E_Req : {
      next(NS_Lock) := 1;
      next(S_Go) := 1;
    }
    S_Go & ~S_Sense : {
      next(S_Go) := 0;
      next(S_Req) := 0;
      if(~N_Go)
           next(NS_Lock) := 0;
    }
  }
```

- The fix: have South light code check whether North light ***about to go on*** before unlocking

```
if( ~( N_Go |
      /* North light about to go on? */
      N_Reg & ~N_Go & ~E_Reg) )
        next(NS_Lock) := 0;
```

Bug 2: Liveness

- Un-cleared lock when North & South go off same time
  - Lock stays on → "higher-priority" East light un-served → deadlocked!
  - Each North and South code block think the other light is still on, no one clears lock



/15/09

Bug 2: Liveness

- Un-cleared lock when North & South go off same time
  - Lock stays on → "higher-priority" East light un-served → deadlocked!
  - Each North and South code block think the other light is still on, no one clears lock

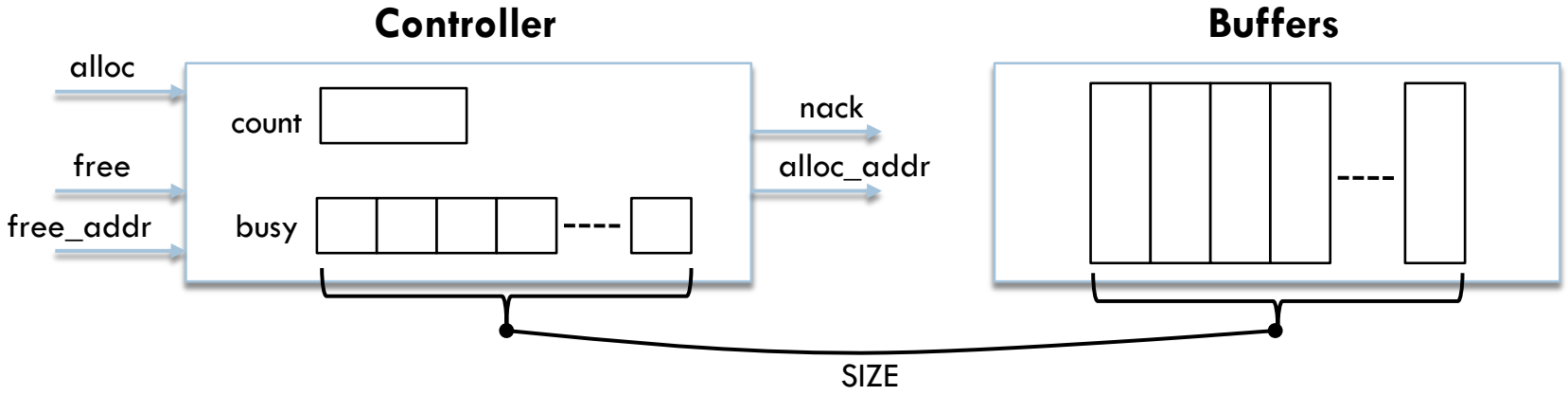
```
/* Original North light controller code */
in default case{
   N_Req & ~N_Go & ~E_Req : {
     next(NS_Lock) := 1;
     next(N_Go) := 1;
   }
   N_Go & ~N_Sense : {
     next(N_Go) := 0;     /* turn off */
     next(N_Req) := 0;    /* clear history */
     if(~S_Go) next(NS_Lock) := 0;  /* unlock if South-going is off */
   }
 }
```

- 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;
```

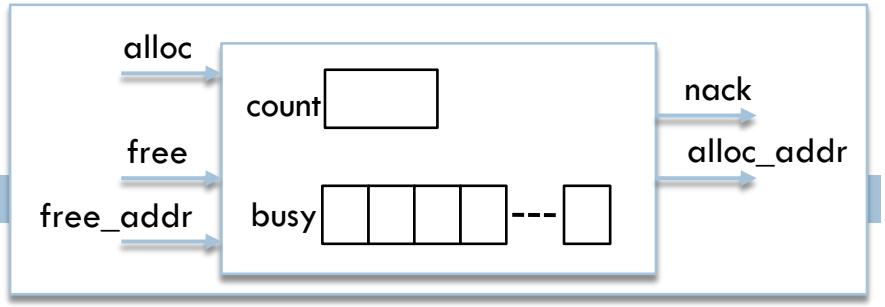
Example 3: Buffer Alloc. Controller

- Controls the allocation and freeing of buffers




```
#define SIZE 32
module main(alloc,nack,alloc_addr,free,free_addr)
{
  input alloc : boolean;
  output nack : boolean;
  output alloc_addr : 0..(SIZE - 1);
  input free : boolean;
  input free_addr : 0..(SIZE - 1);

  busy : array 0..(SIZE - 1) of boolean;
  count : 0..(SIZE);
```

Controller Logic



- Initialization

```
/* bit 0 to (SIZE-1) is init to 0 using iterator expression */
init(busy) := [0 : i = 0..(SIZE-1)];
init(count) := 0;
```

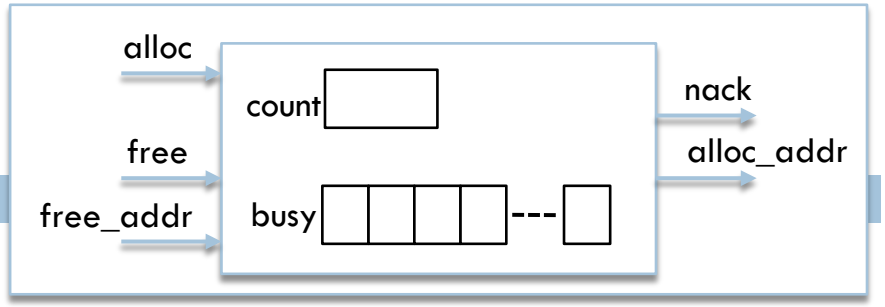
- Nack
  - 1 when allocation is requested, but busy bit count is equal to buffer size
  - i.e., all entries are busy, no free buffer

```
nack := alloc & (count = SIZE);
```

- Count update
  - +1 when allocation requested, and no nack
  - -1 on free request on a busy buffer

```
next(count) := count
             + (alloc & ~nack)
             - (free & busy[free_addr]);
```

Controller Logic



- Setting/clearing busy bits
  - Note: allocation takes precedence if buffer is freed and allocated at same time

```
default{  /* free request */
   if(free) next(busy[free_addr]) := 0;
} in {    /* allocation request AND buffer available (no nack) */
   if(alloc & ~nack) next(busy[alloc_addr]) := 1;
}
```

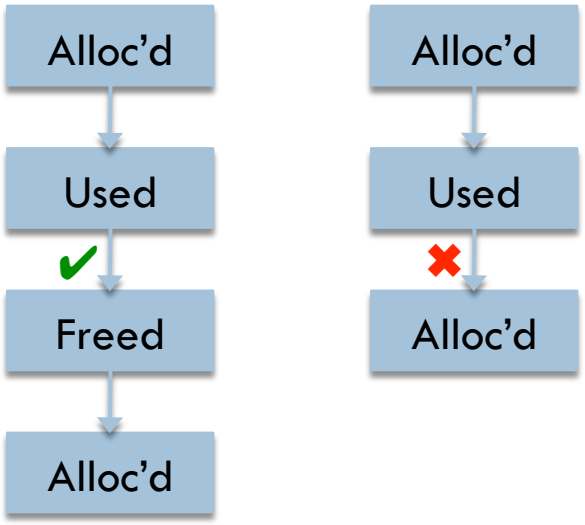
- Choosing a buffer to allocate
  - scan from buffer 0 to SIZE-1, allocate the first one that's free

```
chain(i = (SIZE - 1); i >= 0; i = i - 1){
   if(~busy[i]) alloc_addr := i; /* buffer available, output addr */
}
```

```
default      { if(~busy[SIZE-1]) alloc_addr := SIZE-1 }
in default   { if(~busy[SIZE-2]) alloc_addr := SIZE-2 }
...
in           { if(~busy[0])      alloc_addr := 0      }
```

Specification

- Safety: buffer never allocated twice without being freed
  - note: we'll write the specification for *each* buffer



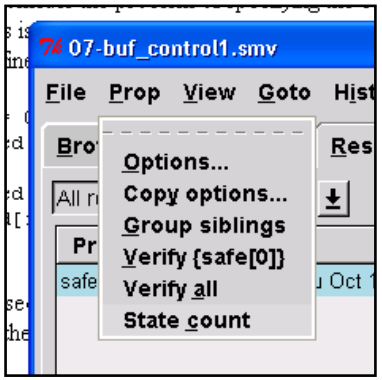

```
/* Helper signals */
for(i = 0; i < SIZE; i = i+1){
   allocd[i], freed[i] : boolean;

   /* buffer i is being allocated */
   allocd[i] := alloc & ~nack & alloc_addr = i;

   /* buffer i is being freed */
   freed[i] := free & free_addr = i;
}

/* The specification */
for(i = 0; i < SIZE; i = i+1){
    safe[i] : assert G (allocd[i] -> ~ X ((~freed[i]) U allocd[i]));
}
```

"if buffer i is allocated, then it is not the case that, starting at the next time, it remains unfreed until it is allocated a second time."

BDD Limit

- SMV uses symbolic model checking with BDD
  - Allows handling more states than explicit enumeration
  - → Nevertheless, does not eliminate state explosion

- For buffer allocation controller example – verifying `safe[0]`
  - Buffer size of 32 → under 1 minute, 10^9 states reached
  - Buffer size of 64 → 10+ minutes, 10^{19} states reached
  - … eventually, state explosion!



How can we scale further?