Computer Lab 1: Model Checking and Logic Synthesis using Spin (lab)

Richard M. Murray

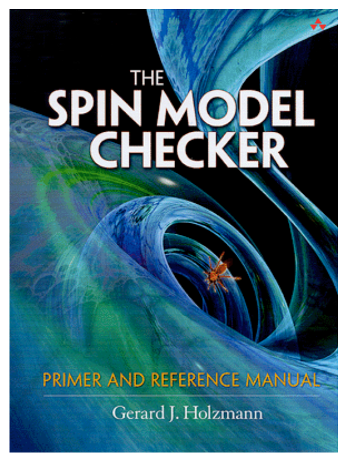
Nok Wongpiromsarn Ufuk Topcu

California Institute of Technology

EECI 15 May 2012

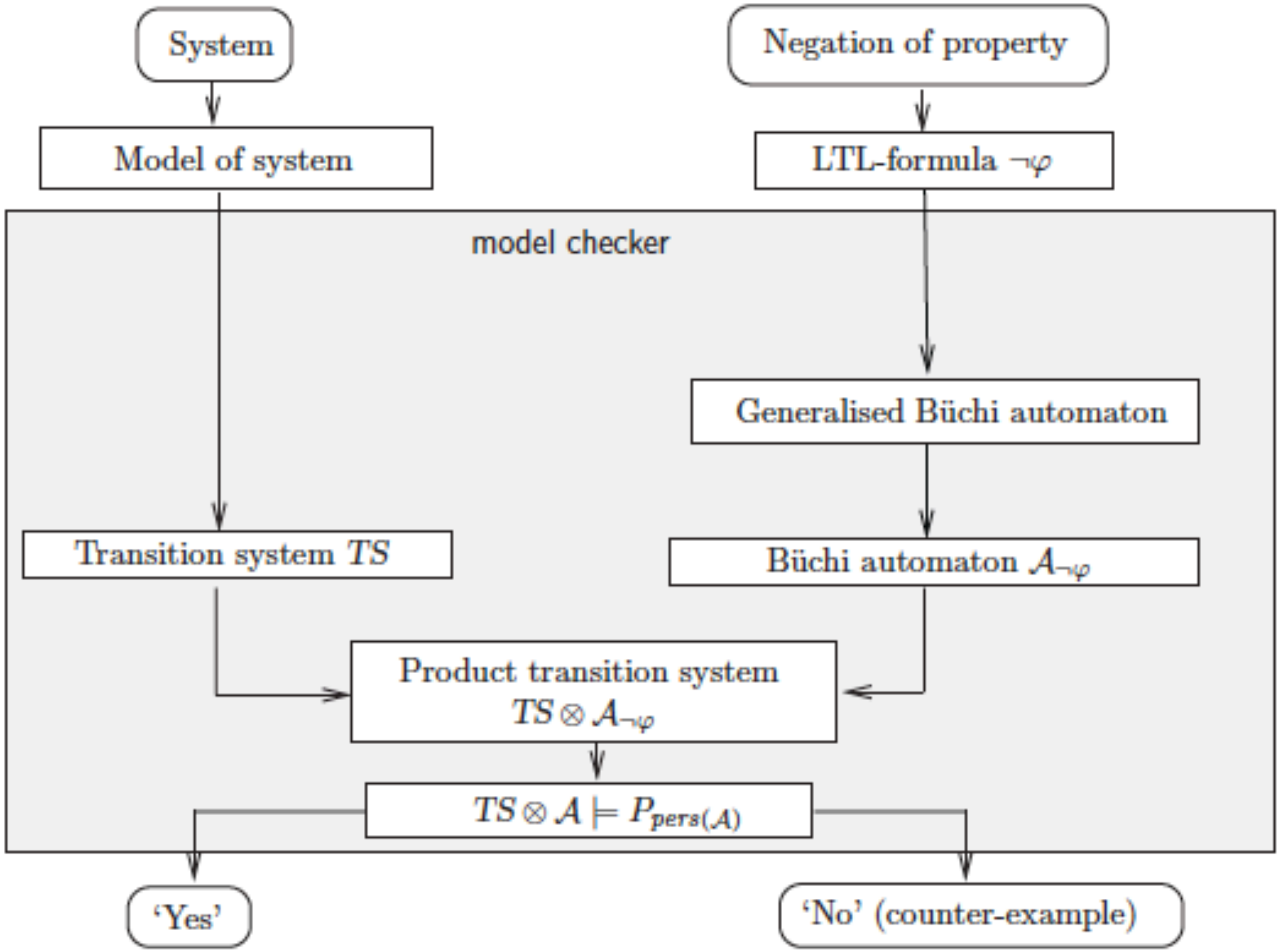
Outline

- Spin model checker: modeling concurrent systems and describing system requirements in Promela
- Model-checking with Spin
- Logic synthesis with Spin



The Spin Model Checker
Gerard J. Holzmann
Addison-Wesley, 2003
http://spinroot.com

The process flow of model checking



Efficient model checking tools automate the process: SPIN, nuSMV, TLC,...

Spin Verification Models

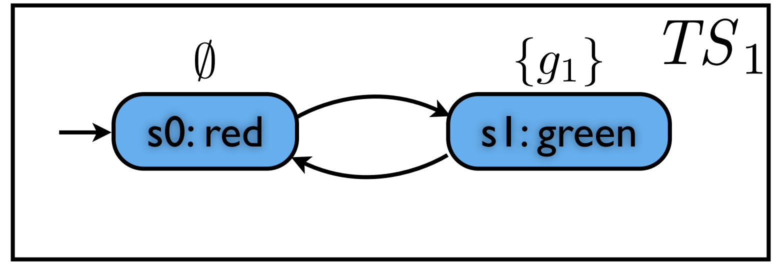
System design (behavior specification)

- *Promela* (Process Meta Language) is a non-deterministic, guarded command language for specifying possible system behavior of a distributed system in Spin
- There are 3 types of objects in Spin verification model
  - asynchronous processes
  - global and local data objects
  - message channels

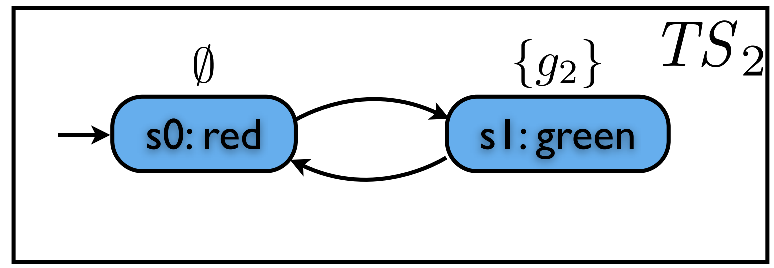
System requirements (correctness claims)

- default properties
  - absence of system deadlock
  - absence of unreachable code
- assertions
- end-state labels
- acceptance
- progress
- fairness
- never claim
- LTL formulas
- trace assertions

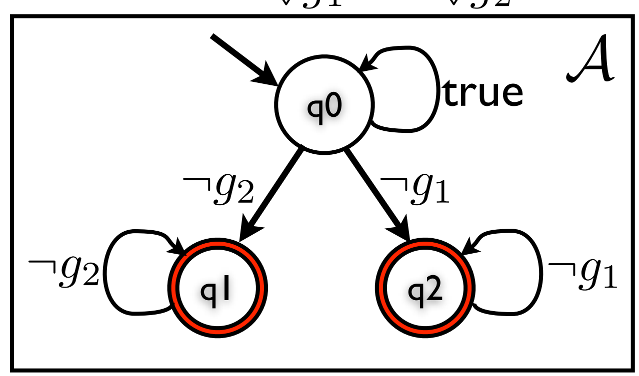
process 1



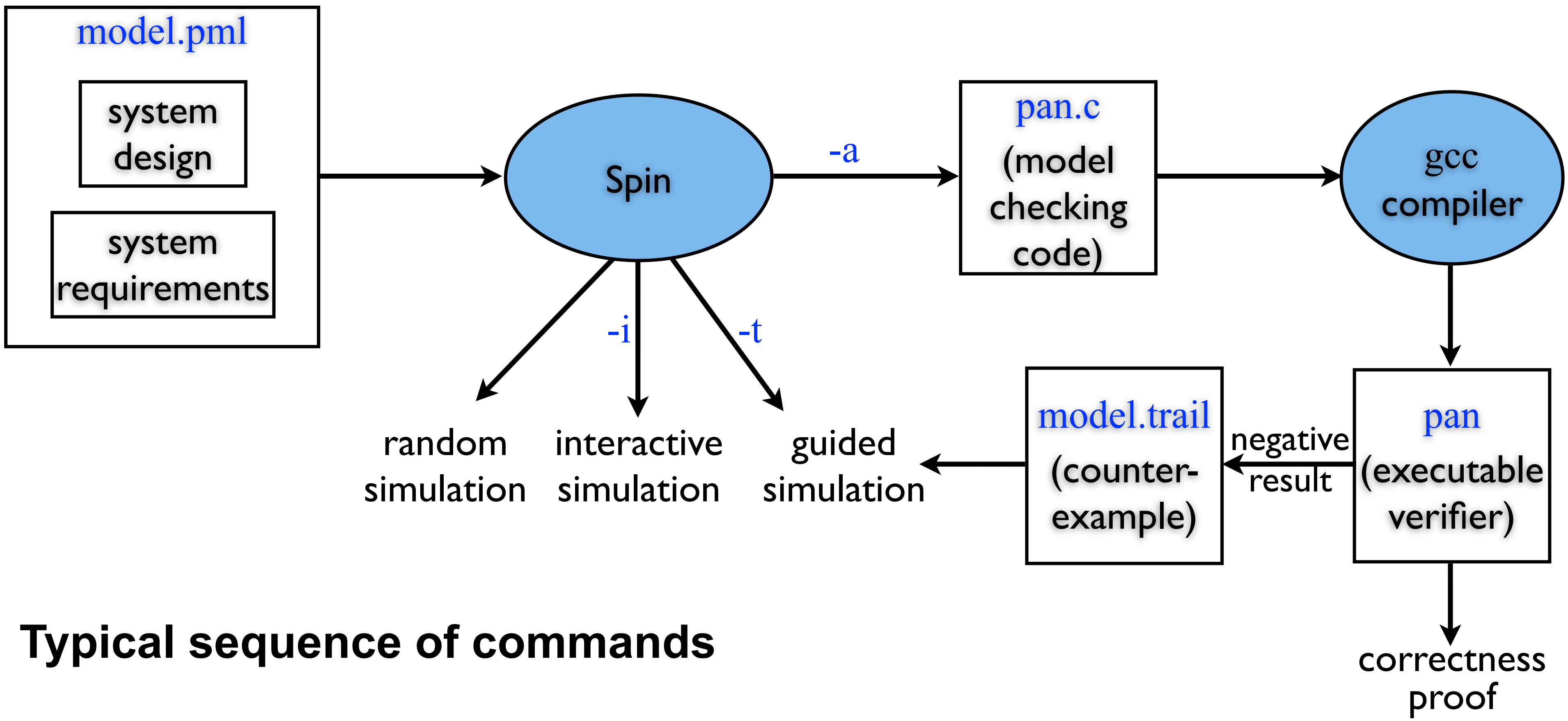
process 2



$$\Phi = \Box\Diamond g_1 \wedge \Box\Diamond g_2$$



Running Spin



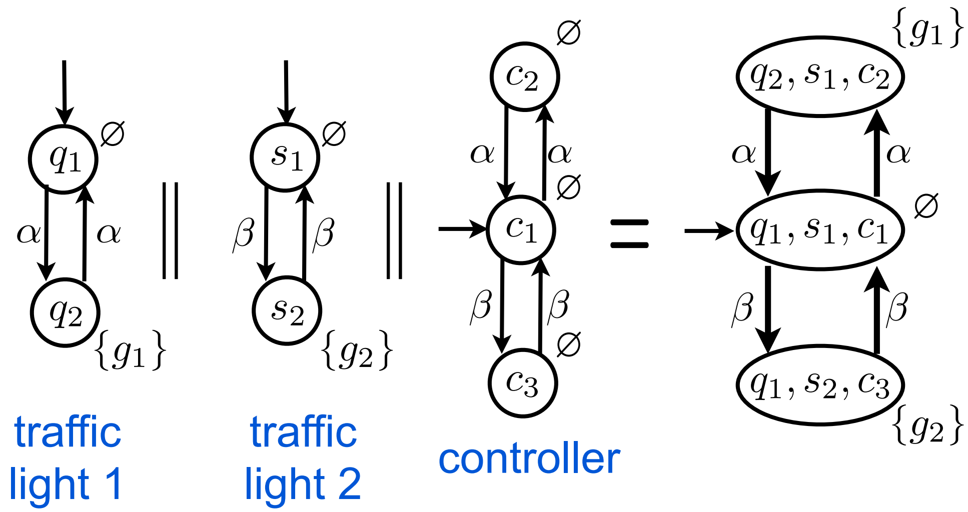
Typical sequence of commands

```
$ spin -u100 model # non-verbose simulation for 100 steps
$ spin -a model    # generate C code for analysis (pan.c)
$ gcc -o pan pan.c # generate executable verifier from pan.c
$ ./pan -a -N P1   # perform verification of specification P1
$ spin -t -p model # show error trail
```

Note: `spin --` and `./pan --` list available command-line and un-time options, resp.

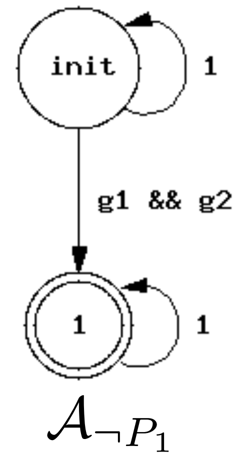
Example 1: traffic lights (property verified)

System *TS*: composition of two traffic lights and a controller



Specification P_1:
"The light are never green simultaneously."

[](!(g1 && g2))



SPIN code:

lights_simple.pml

```
bit g1=0, g2=0;

active proctype P()
{
  do
  :: atomic{ (g1==0 && g2==0) -> g1=1; g2=0 }
  :: atomic{ (g1==0 && g2==0) -> g1=0; g2=1 }
  :: atomic{ (g1==1 && g2==0) -> g1=0; g2=0 }
  :: atomic{ (g1==0 && g2==1) -> g1=0; g2=0 }
  od
}

ltl P1 { [] (! (g1 && g2)) }
ltl P2 { [] <> g1 }
ltl P3 {
  (always (!(g1&&g2))) &&
  (always eventually g1)
}
```

Property verified:

$$TS \models P_1$$

```
spin -a lights_simple.pml
gcc -o pan pan.c
./pan -a -N P1 lights_simple.pml
./pan -a -N P2 lights_simple.pml
spin -t -p lights_simple.pml
```

Promela: Process Modeling Language

```
bit g1, g2;            /* light status */
bit alpha1, alpha2, beta1, beta2;
int c = 1;             /* control state */

active proctype TL1() {
  do
  :: alpha1 -> g1 = 1
  :: beta1 -> g1 = 0
  od
}

active proctype TL2() {
loop2:
  alpha2 -> g2 = 1
  beta2 -> g2 = 0
  goto loop2
}

active proctype control() {
  do
  :: c == 1 -> alpha1=1; beta1=0; c = 2;
  :: c == 2 -> alpha1=0; beta1=1; c = 3;
  :: c == 3 -> alpha2=1; beta2=0; c = 4;
  :: c == 4 -> alpha2=0; beta2=1; c = 1;
  od
}

ltl P1 { [] <> g1 }
ltl P2 { [] ! (g1 && g2) }
```

Declarations

- Declare global variables (+ initialize)
- Types: bit, int, char, etc + arrays

Processes

- Each process runs independently
- 'active' => process starts immediately
  - Otherwise use 'run' command
- Process = sequence of statements
- Statements = guard or assignment

Control flow

- 'do' loops: non-deterministic execution
- 'goto' statements: jump to location
- guarded commands: 'guard -> rule'

Specifications

- LTL statement to be checked
  - These generate 'never' claims internally to spin (will see later)

Promela Objects: Processes

- A keyword `proctype` is used to declare process behavior
- 2 ways to instantiate a process
  - Add the prefix `active` to a `proctype` declaration. The process will be instantiated in the initial system state.
  - Use `run` operator to instantiate a process in any reachable system state

# of processes to be instantiated in the initial system state

```
active [2] proctype main()
{
    prinf("hello world\n")
}
```

keyword for initial process declaration and instantiation

```
proctype you_run(byte x)
{
    printf("x = %d\n", x)
}
init
{
    run you_run(0);
    run you_run(1)
}
```

Extra process `init` needs to be created

Promela Objects: Data Objects

- 2 levels of scope: global and process local
- No intermediate levels of scope
- The default initial value of all data objects is zero
- All objects must be declared before they can first be referenced
- User-defined type can be declared using keyword `typedef`

Type	Typical Range	Sample Declaration
`bit`	0,1	`bit turn = 1`
`bool`	*false*, *true*	`bool flag = true`
`byte`	$0 \ldots 255$	`byte a[12]`
`chan`	$1 \ldots 255$	`chan m`
`mtype`	$1 \ldots 255$	`mtype n`
`pid`	$0 \ldots 255$	`pid p`
`short`	$-2^{15} \ldots 2^{15} - 1$	`short b[4] = 89`
`int`	$-2^{31} \ldots 2^{31} - 1$	`int cnt = 67`
`unsigned`	$0 \ldots 2^n - 1$	`unsigned w : 3 = 5`

all elements initialized to 0 (`byte a[12]`)

all elements initialized to 89 (`short b[4] = 89`)

unsigned stored in 3 bits (range 0...7)

Basic Statements

- Assignment
  - valid assignment: `c++, c--, c = c+1, c = c-1`
  - invalid assignment: `++c, --c`

if the right-hand side yields a value outside the range of c, truncation can result

- Expressions
  - must be side effect free
  - the only exception is the run operator, which can have a side effect
- Print: `printf("x = %d\n", x)`
- Assertion: `assert(x+y == z), assert(x <= y)`
  - always executable and has no effect on the state of the system when executed
  - can be used to check safety property: Spin reports a error if the expression can evaluate to zero (*false*)

```
int n;
active proctype invariant()
{
   assert(n <= 3)
}
```

The assertion statement can be executed at any time. This can be used to check a system invariant condition: it should hold no matter when the assertion is checked.

- send
- receive

} message passing between processes (later, if time)

Rules for Executability

A statement in a Spin model is either *executable* or *blocked*

- A statement is executable iff it evaluates to *true* or non-zero integer value

`2 < 3`	is always executable
`x < 27`	executable iff $x < 27$
`3 + x`	executable iff $x \neq 3$

- print statements and assignments are always unconditionally executable
- If a process reaches a point where there is no executable statements left to execute, it simply blocks

```
while (a != b)
{
   skip;
}
```

do nothing while waiting for a==b

```
a == b;
```

block until a==b

```
do
::  (a == b) -> break
::  else -> skip
od
```

```
L:  if
    ::  (a == b) -> skip
    ::  else -> goto L
    fi
```

Nondeterminism

2 levels of nondeterminism

- System level: processes execute concurrently and asynchronously
  - Process scheduling decisions are non-deterministic
  - Statement executions from different processes are arbitrarily interleaved in time
    - Basic statements execute atomically
- Process level: local choice within processes can also be non-deterministic

```
byte x = 2, y = 2;
active proctype A() {
   do
   ::  x = 3-x
   ::  y = 3-y
   od
}
active proctype B() {
   do
   ::  x = 3-y
   ::  y = 3-x
   od
}
```

At any point in an execution, any of these statements can be executed

Control Flow

- Semicolons, gotos and labels
- Atomic sequences: `atomic{ ... }`
  - Define an indivisible sequence of actions
  - No other process can execute statements from the moment that the first statement of this sequence begins to execute until the last one has completed
- Deterministic steps: `d_step{ ... }`
  - Similar to atomic sequence but more restrictive, e.g., no nondeterminism, goto jumps, or unexecutable statements is allowed
- Nondeterministic selection:

```
if
::  guard1 -> stmnt11; stmnt12; ...
::  guard2 -> stmnt21; stmnt22; ...
::  ...
fi
```

- Nondeterministic repetition:

```
do
::  guard1 -> stmnt11; stmnt12; ...
::  guard2 -> stmnt21; stmnt22; ...
::  ...
do
```

- Escape sequences: `{ P } unless { E }`
- Inline definitions: `inline{ ... }`

swap the values of a and b

```
atomic {
   tmp = b;
   b = a;
   a = tmp
}
```

```
d_step {
   tmp = b;
   b = a;
   a = tmp
}
```

```
if
::  (n % 2 != 0) -> n = 1
::  (n >= 0) -> n = n-2
::  (n % 3 == 0) -> n = 3
::  else /* -> skip */
fi
```

the else guard is executable iff none of the other guards is executable.

without the else clause, the if-statement would block until other guards becomes true.

```
do
::  x++
::  x--
::  break
od
```

transfers control to the end of the loop

Nondeterministic Selection and Repetition

```
if
::   guard1 -> stmnt11; stmnt12; ...
::   guard2 -> stmnt21; stmnt22; ...
::   ...
fi
```

```
do
::   guard1 -> stmnt11; stmnt12; ...
::   guard2 -> stmnt21; stmnt22; ...
::   ...
do
```

- If at least one guard is executable, the if/do statement is executable
- If more than one guard is executable, one is selected non-deterministically
- If none of the guard statements is executable, the if/do statement blocks
- Any type of basic or compound statement can be used as a guard
- ‘if’ statement checks once and continues; ‘do’ statement re-executes code until a break is reached

Defining Correctness Claims

- default properties
  - absence of system deadlock
  - absence of unreachable code
- assertions
  - local process assertions
  - system invariants
- end-state labels
  - define proper termination points of processes

safety

- "nothing bad ever happens"
- properties of reachable states

- accept-state labels
  - when looking for acceptance cycles
- progress-state labels
  - when looking for non-progress cycles
- fairness
- never claims
- LTL formulas
- trace assertions

liveness

- " something good eventually happens"
- properties of infinite sequences of states

Progress and Acceptance

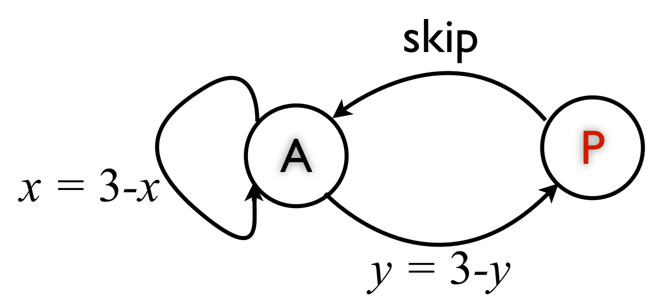
Progress

- Search for reachable non-progress cycles (infinite executions that do *not* pass through any progress state)
- Progress states are specified using `progress` label
- Enforced by `gcc -DNP` and `pan -l`

Acceptance

- Search for acceptance cycles (infinite executions that *do* pass through a specially marked state)
- Acceptance states are specified using `accept` label
- Enforced by `pan -a`

```
byte x = 2, y = 2;
active proctype A()
{
   do
   ::  x = 3-x
   ::  y = 3-y;   progress:  skip
   od
}
```



a non-progress cycle is an infinite execution sequence that does not pass through any progress state

Fairness

Weak fairness

- If a statement is executable infinitely long, it will eventually be executed
- Process-level weak-fairness can be enforced by run-time option `pan -f`
  - if a process contains at least one statement that remains executable infinitely long, that process will eventually execute a step
  - does not apply to non-deterministic transition choices within a process

Strong fairness

- If a statement is executable infinitely often, it will eventually be executed

Enforcing fairness increases the cost of verification

- **Weak fairness**: complexity is linear in the number of active processes
- **Strong fairness**: complexity is quadratic in the number of active processes

```
byte x = 2, y = 2;
active proctype A() {
   do
   ::  x = 3-x
   od
}
active proctype B() {
   do
   ::  y = 3-y;   progress:  skip
   od
}
```

```
$ spin -a progress.pml
$ gcc -DNP -o pan pan.c
$ ./pan -l -f
```

```
(Spin Version 5.2.4 -- 2 December 2009)
        + Partial Order Reduction

Full statespace search for:
        never claim             +
        assertion violations    + (if within scope of claim)
        non-progress cycles     + (fairness enabled)
        invalid end states      - (disabled by never claim)

State-vector 24 byte, depth reached 15, errors: 0
       16 states, stored
       17 states, matched
       33 transitions (= stored+matched)
        0 atomic steps
hash conflicts:         0 (resolved)

    4.653       memory usage (Mbyte)

unreached in proctype A
        line 7, state 5, "-end-"
        (1 of 5 states)
unreached in proctype B
        line 14, state 6, "-end-"
        (1 of 6 states)

pan: elapsed time 0 seconds
```

Never Claims

Define an observer process that executes synchronously with the system

- Intended to monitor system behavior; do not contribute to system behavior
- Can be either deterministic or non-deterministic
- Contain only side-effect free expressions
- Abort when they block
- Reports a violation when
  - closing curly brace of never claim is reached
  - an acceptance cycle is found (infinite execution passing through accept label)

```
spin -f '! []<>g1'
```

```
never {    /* ! []<>g1 */
T0_init:
  if
  :: (! ((g1))) -> goto accept_S4
  :: (1) -> goto T0_init
  fi;
accept_S4:
  if
  :: (! ((g1))) -> goto accept_S4
  fi;
}
```

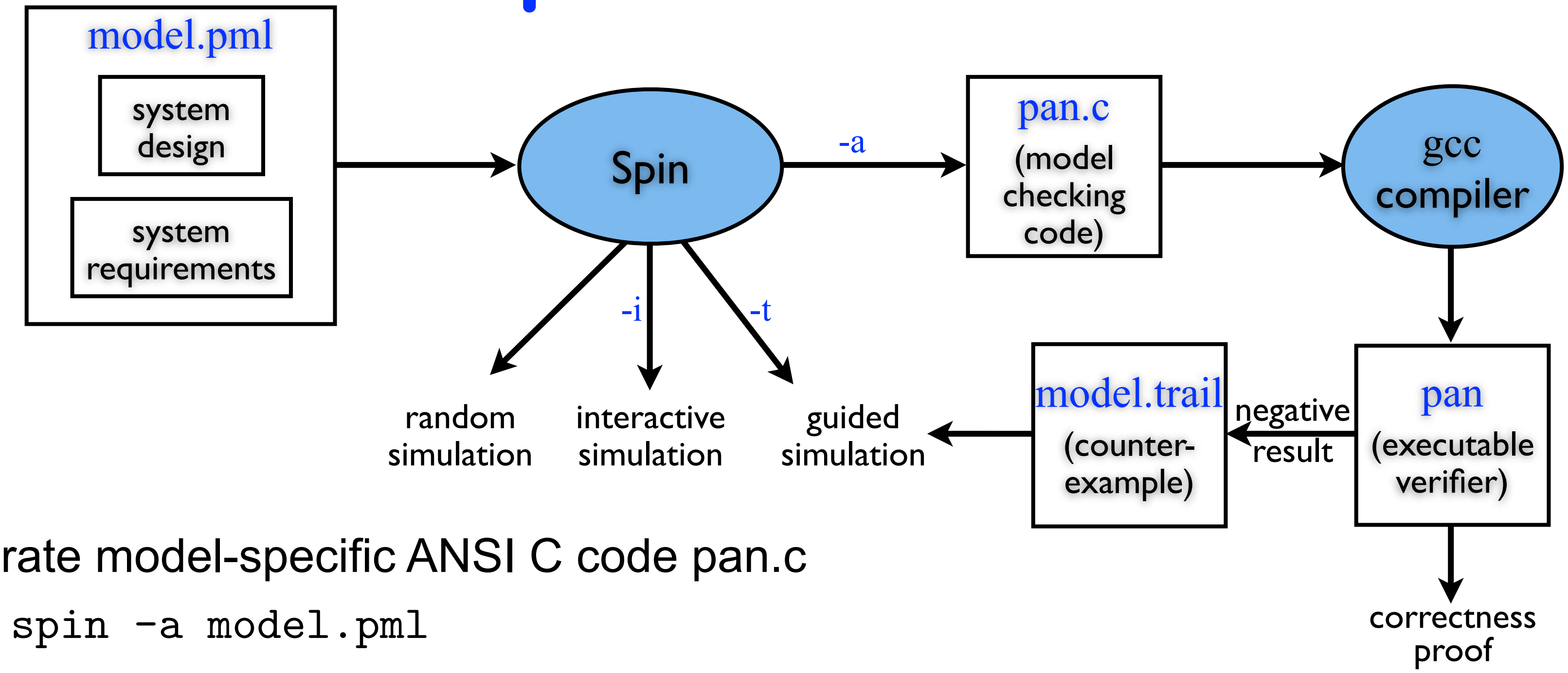
Typically used to enforce LTL property

- Old style: `spin -f '!spec'` generates never claim
- New style: use `ltl label { spec }`
- Make sure to run pan -a when you have never claims

Example: []<>g1

- To make sure this is *always* true, need to make sure that !spec is *never* true (same inversion as usual)

Spin Commands



Generate model-specific ANSI C code pan.c

```
$ spin -a model.pml
```

Generate verifier from pan.c

- Typical command

```
$ gcc -o pan pan.c
```

- Enforcing progress

```
$ gcc -DNP -o pan pan.c
```

Perform verification

- Typical command

```
$ ./pan -a -N P1 model.pml
```

- Enforcing progress: add `-l`
- Enforcing acceptance: add `-a`
- Enforcing fairness: add `-f`

Relay error trail

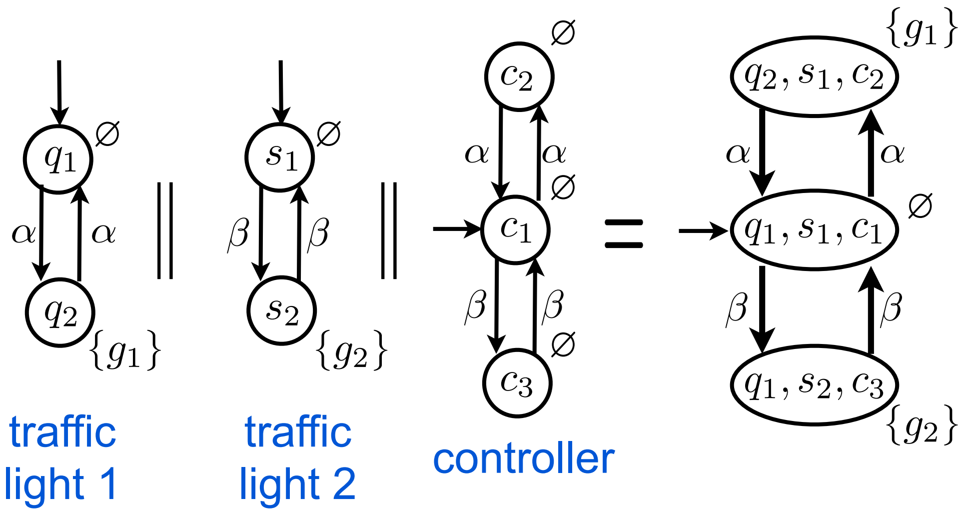
```
$ spin -t -p -g model.pml
```

-t: follow error trail
-p: print all statements
-g: print all global variables

Note: `spin --` and `./pan --` list available command-line and run-time options, resp

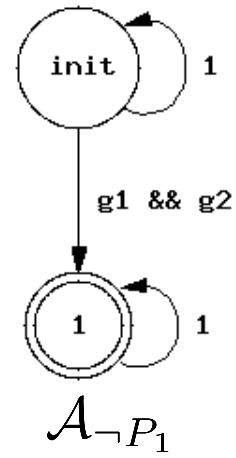
Exercise 1: traffic lights

System *TS*: composition of two traffic lights and a controller



Specification P_1:
"The light are never green simultaneously."

[](!(g1 && g2))



Property verified:

$$TS \models P_1$$

```
spin -a lights_simple.pml
gcc -o pan pan.c
./pan -a -N P1 lights_simple.pml
./pan -a -N P2 lights_simple.pml
spin -t -p lights_simple.pml
```

SPIN code:

lights_simple.pml

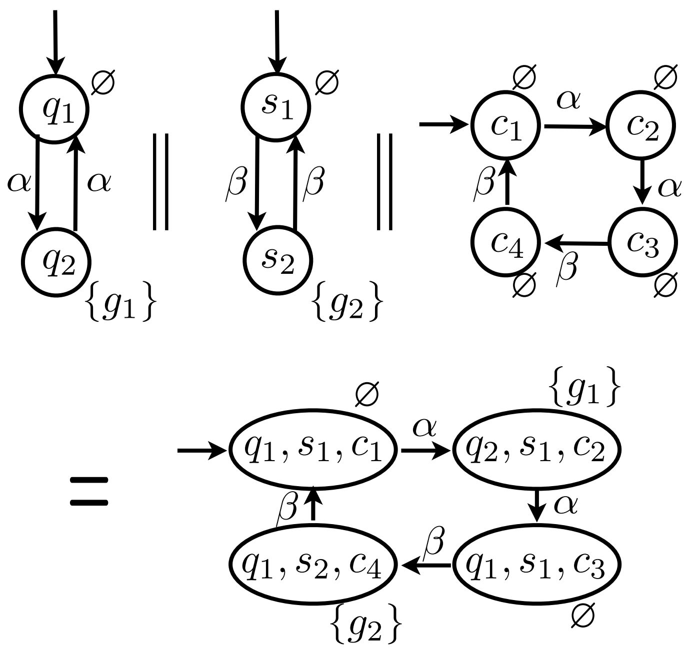
```
bit g1=0, g2=0;

active proctype P()
{
  do
  :: atomic{ (g1==0 && g2==0) -> g1=1; g2=0 }
  :: atomic{ (g1==0 && g2==0) -> g1=0; g2=1 }
  :: atomic{ (g1==1 && g2==0) -> g1=0; g2=0 }
  :: atomic{ (g1==0 && g2==1) -> g1=0; g2=0 }
  od
}

ltl P1 { [] (! (g1 && g2)) }
ltl P2 { [] <> g1 }
ltl P3 {
  (always (!(g1&&g2))) &&
  (always eventually g1)
}
```

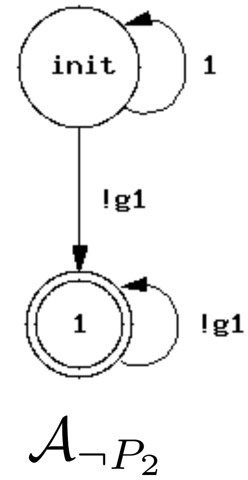
Exercise 2: modified traffic lights

System *TS*: composition of two traffic lights and a modified controller




```
ltl P1 { [] (! (g1 && g2)) }
ltl P2 { [] <> g1 }
ltl P3 {
  (always (!(g1&&g2))) &&
  (always eventually g1)
}
```

Specification P_2:
"The first light is infinitely often green."

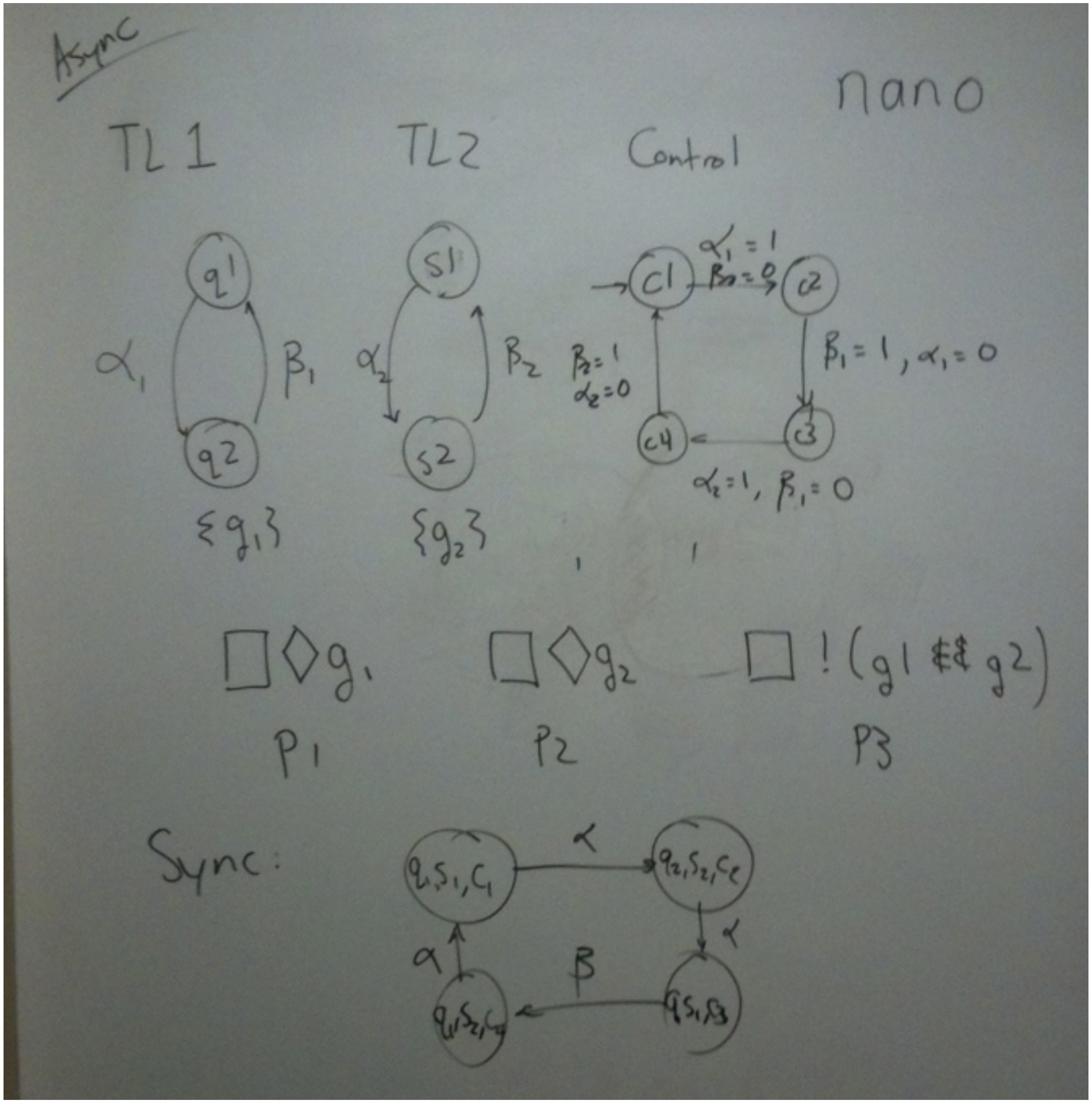


Property verified:

$$TS \models P_2$$

Construct a new Promela model and verify P1, P2, P3

Exercise 3: Traffic Light Controller



Distributed traffic controller

- TL1: traffic light one, accepts on/off commands
- TL2: same for second light
- Control: send a sequence of commands

Approach

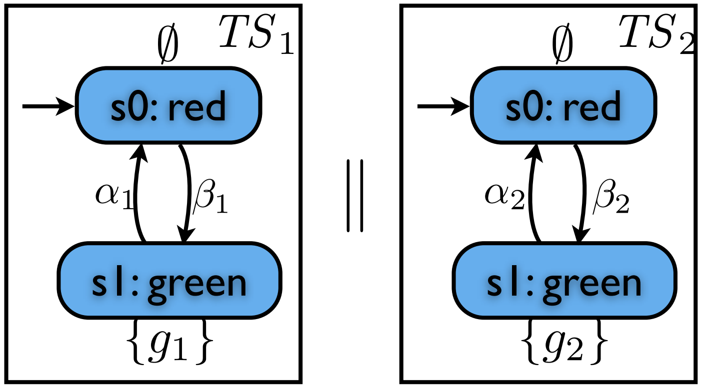
- Model commands to lights using global variables
- Use a finite state machine to implement controller

Check multiple properties

- Both lights turn green infinitely often
- It is never true that both lights are green at the same time

Exercise 4: Controller Synthesis

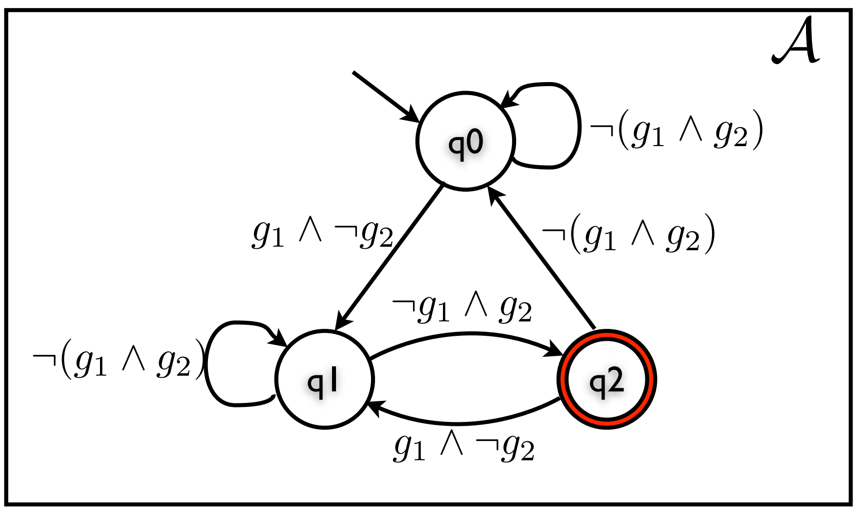
$$P = TS_1 \parallel TS_2$$




```
bool g1 = 0, g2 = 0;
active proctype TL1() {
   do
   ::  atomic{ g1 == 0 -> g1 = 1}
   ::  atomic{ g1 == 1 -> g1 = 0 }
   od
}
active proctype TL2() {
   do
   ::  atomic{ g2 == 0 -> g2 = 1}
   ::  atomic{ g2 == 1 -> g2 = 0 }
   od
}
```

$$\Phi = \Box\neg(g_1 \wedge g_2) \wedge \Box\Diamond g_1 \wedge \Box\Diamond g_2$$

$$\mathcal{L}_\omega(\mathcal{A}) = Words(\Phi)$$




```
never {
T0_init:
   if
   ::  (!g1) || (!g2) -> goto T0_init
   ::  (g1 && !g2) -> goto T1_S1
   fi;
T1_S1:
   if
   ::  (!g1) || (!g2) -> goto T1_S1
   ::  (!g1 && g2) -> goto accept_S1
   fi;
accept_S1:
   if
   ::  (!g1) || (!g2) -> goto T0_init
   ::  (g1 && !g2) -> goto T1_S1
   fi;
}
```

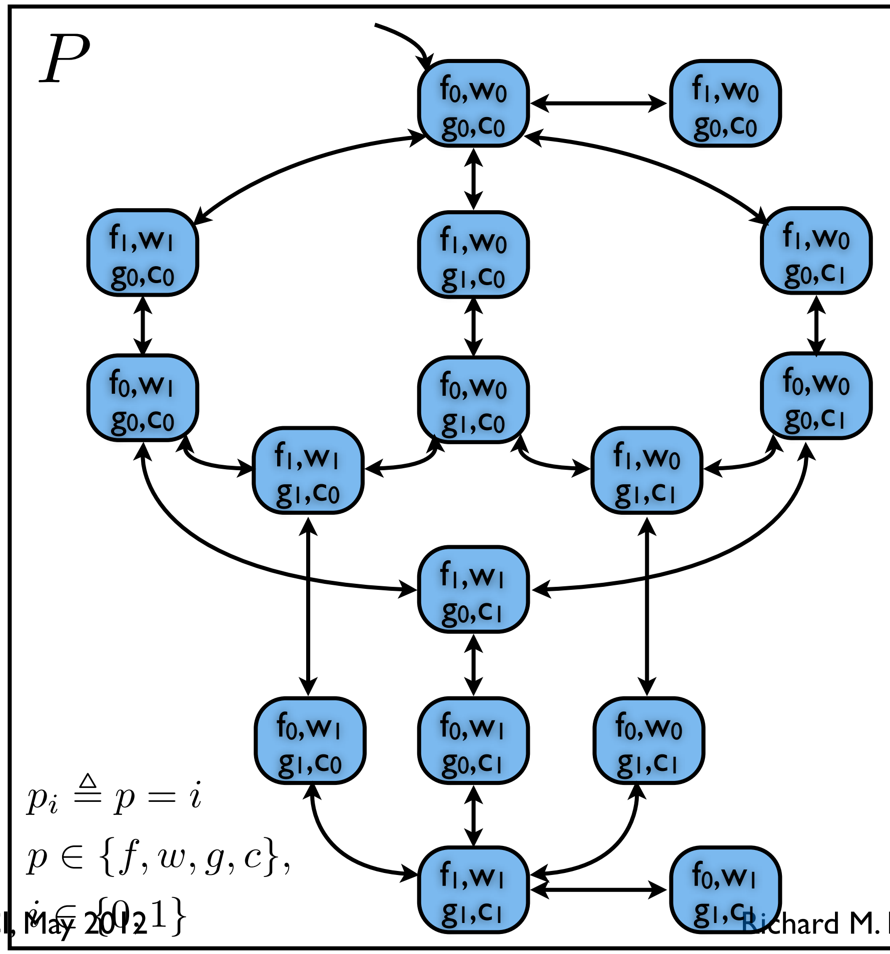
Exercise 5: Farmer Puzzle

A farmer wants to cross a river in a little boat with a wolf, a goat and a cabbage.

Constraints:

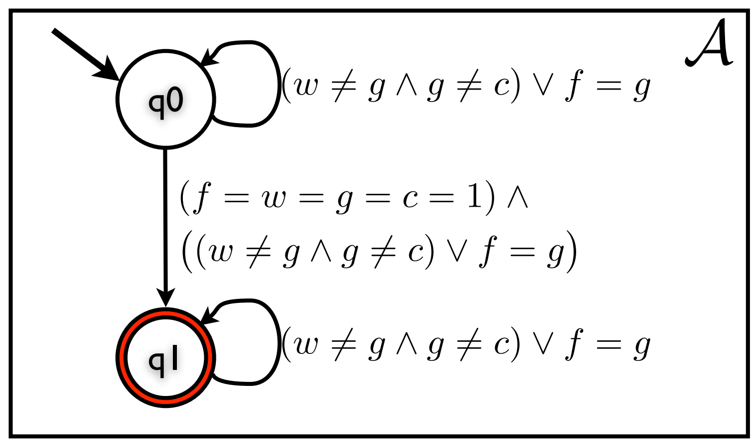
- The boat is only big enough to carry the farmer plus one other animal or object.
- The wolf will eat the goat if the farmer is not present.
- The goat will eat the cabbage if the farmer is not present.

How can the farmer get all both animals and the cabbage safely across the river?

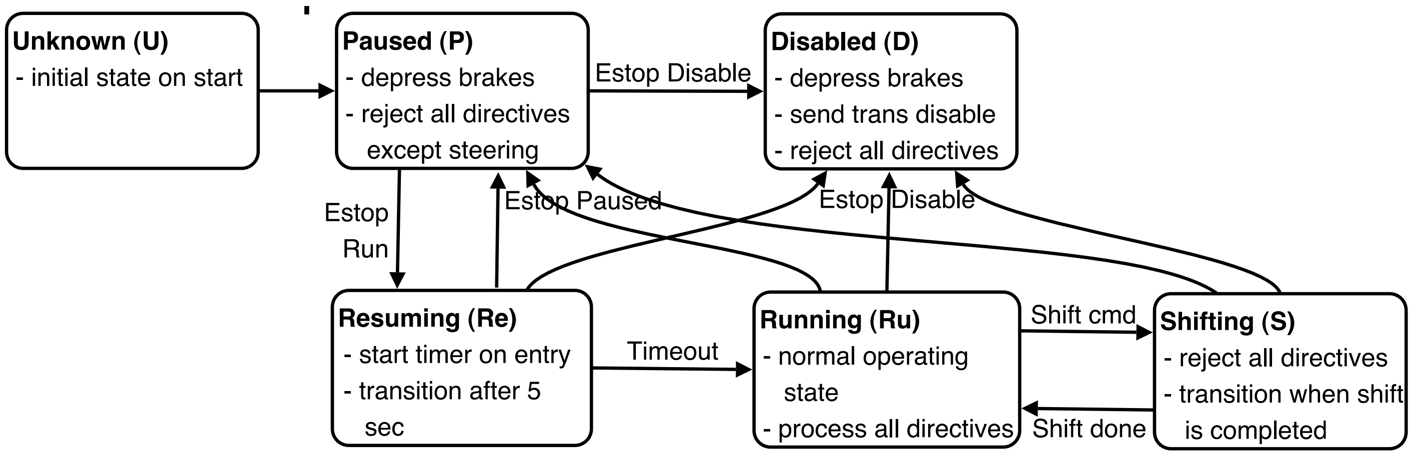


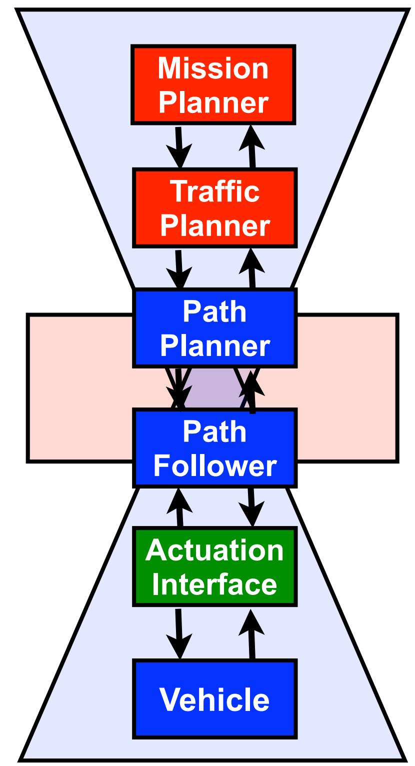
$$\begin{aligned}\Phi \;=\; & \Diamond(f = w = g = c = 1) \wedge \\ & \Box(w \neq g \vee f = g) \wedge \\ & \Box(g \neq c \vee f = g)\end{aligned}$$

$$\mathcal{L}_\omega(\mathcal{A}) = Words(\Phi)$$



Exercise 6: Alice Actuation Interface (adrive) Logic

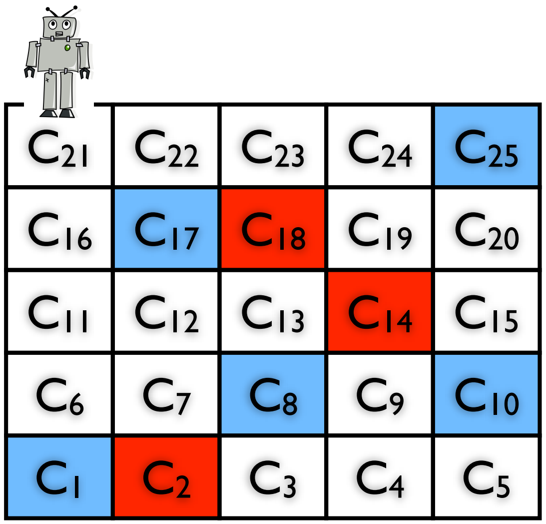




Desired properties

- If *Estop Disable* is received, gcdrive state will be *Disabled* and acceleration will be 'full brake' forever
- *Estop Paused*: if not disabled, gcdrive will eventually enter *Paused* state and acceleration will be 'full brake' (not forever)
- *Estop Run*: if not *Disabled*, gcdrive will eventually be *Running* or *Resuming* (or receive another pause or disable command)
- If *Resuming*, eventually *Running* (or receive another pause or disable)
- If current mode is *Disabled, Paused, Resuming* or *Shifting*, full brake is commanded
- After receiving an *Estop Pause*, vehicle may resume operation 5 seconds after run is received (suffices to show that we transition from *Resuming* to *Running* via *Timeout*)
- **Project:** verify correctness using SPIN model checker and *message channels*

Exercise 7: Robot Motion Planning

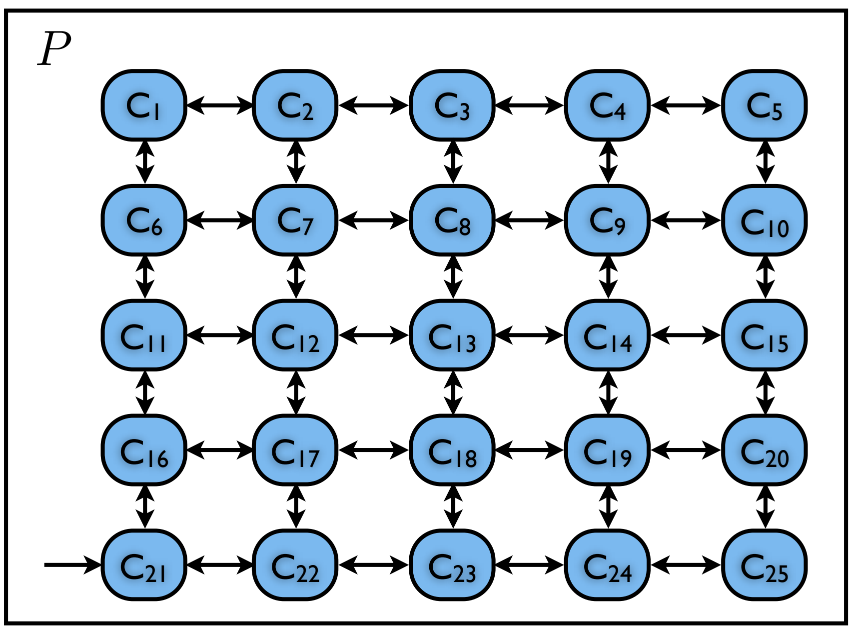


The robot starts from cell C_{21}.

Compute a trajectory for a robot to visit cell C_8, then C_1 and then cover C_{10}, C_{17} and C_{25} while avoiding obstacles C_2, C_{14}, C_{18}.

Physical constraints:

- The robot can only move to an adjacent cell



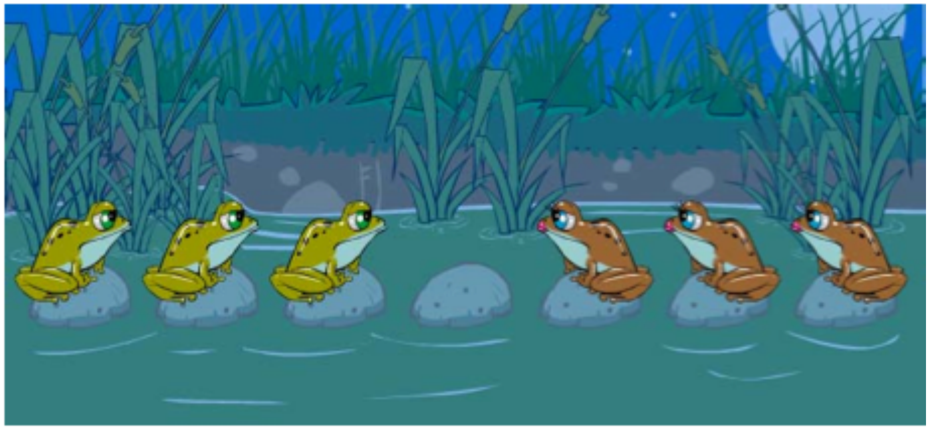
$$\Phi = \Diamond\big(C_8 \wedge \Diamond(C_1 \wedge \Diamond C_{10} \wedge \Diamond C_{17} \wedge \Diamond C_{25})\big) \wedge \Box\neg(C_2 \vee C_{14} \vee C_{18})$$

Example: frog puzzle

Find a way to send all the yellow frogs to the right hand side of the pond and send all the red frogs to the left hand side.

Constraints:

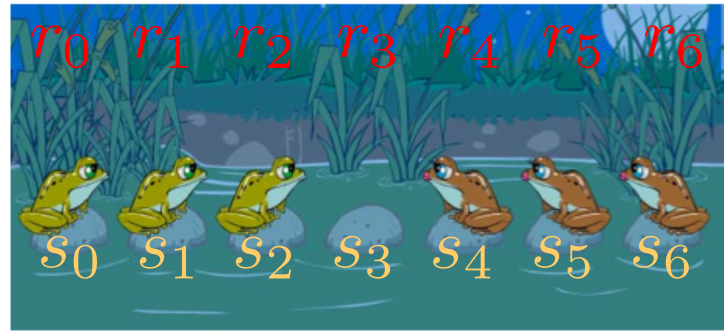
- Frogs can only jump in the direction they are facing.
- Frogs can either jump one rock forward if the next rock is empty or they can jump over a frog if the next rock has a frog on it and the rock after it is empty.

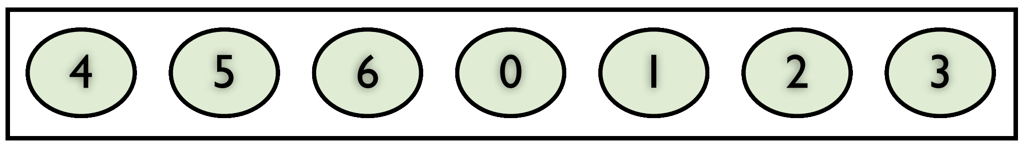


http://www.hellam.net/maths2000/frogs.html

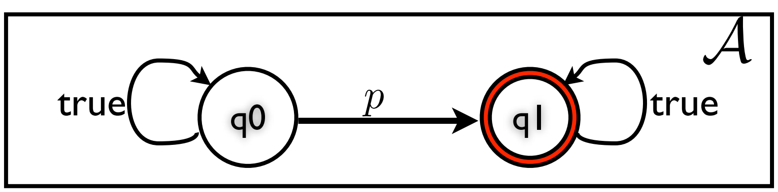
Solving the frog puzzle as logic synthesis

- Rock i is not occupied or occupied $r_i \in \{0, 1\}$
- State of frog i: $s(F_i) \in \{s_0, s_1 \ldots, s_6\}$
- Transition system of frog i: F_i
- Overall system model: $P = F_1 \parallel F_2 \parallel \cdots \parallel F_6$





$$\Phi = \Diamond\big(s(F_1), s(F_2), s(F_3) \in \{s_4, s_5, s_6\} \wedge s(F_4), s(F_5), s(F_6) \in \{s_0, s_1, s_2\}\big)$$



$$p \triangleq \big(s(F_1), s(F_2), s(F_3) \in \{s_4, s_5, s_6\} \wedge s(F_4), s(F_5), s(F_6) \in \{s_0, s_1, s_2\}\big)$$