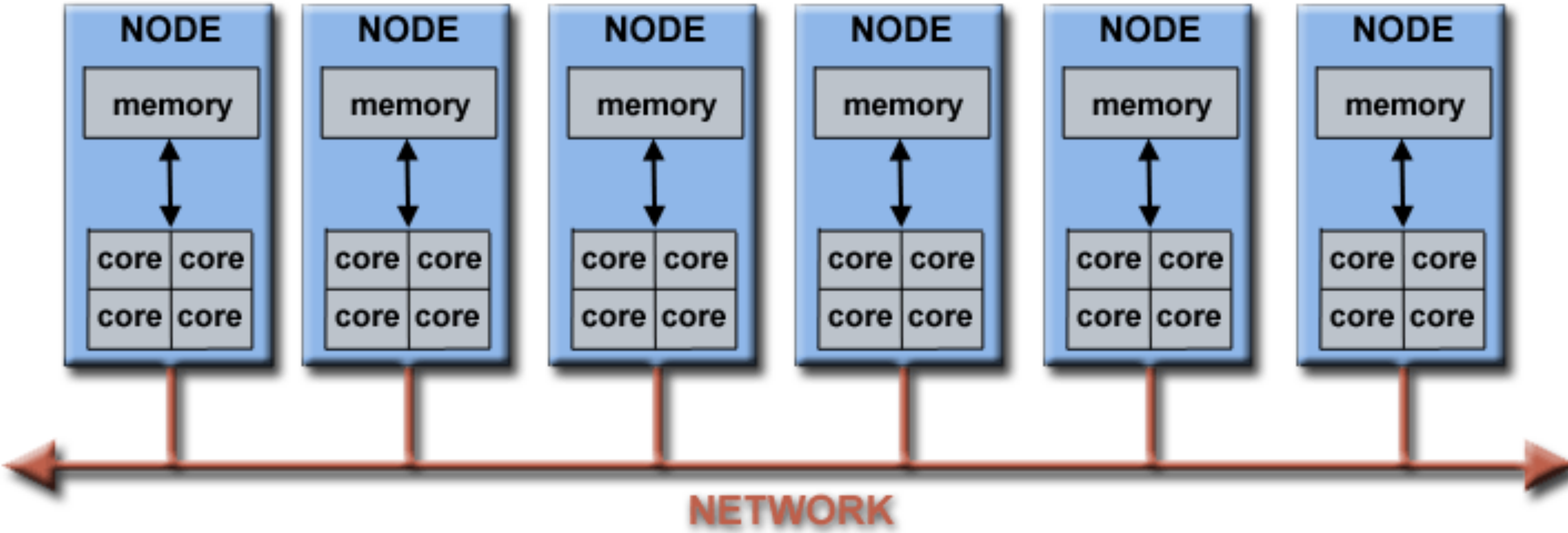
A Tutorial on Model Checker SPIN

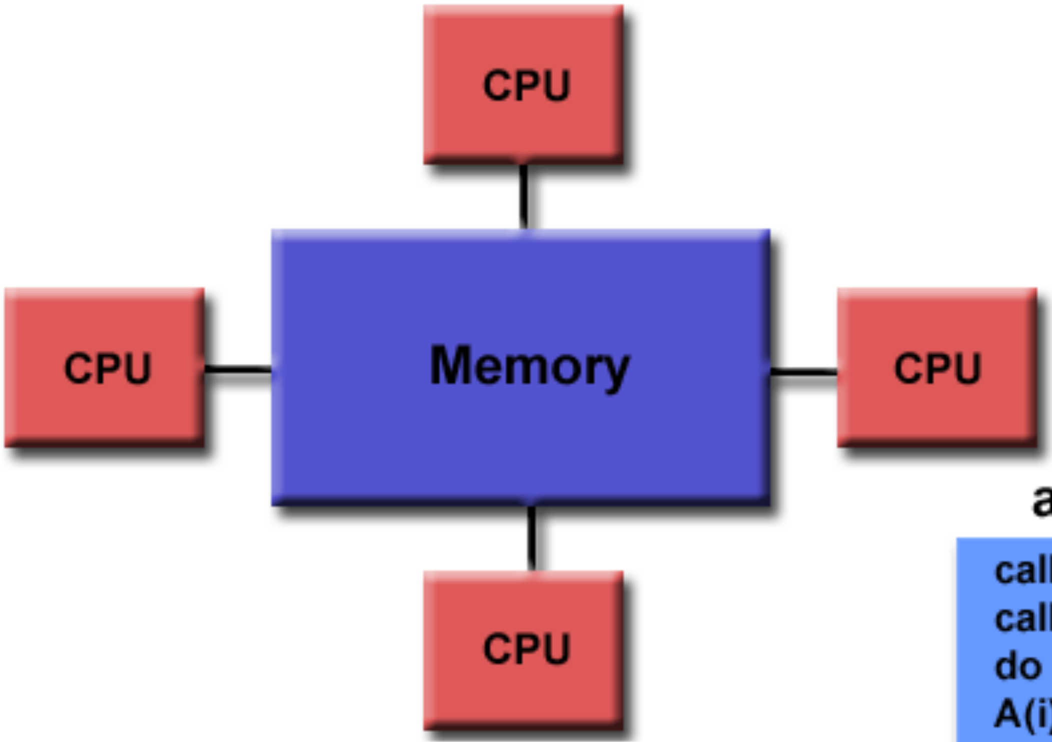
Instructor: Hao Zheng

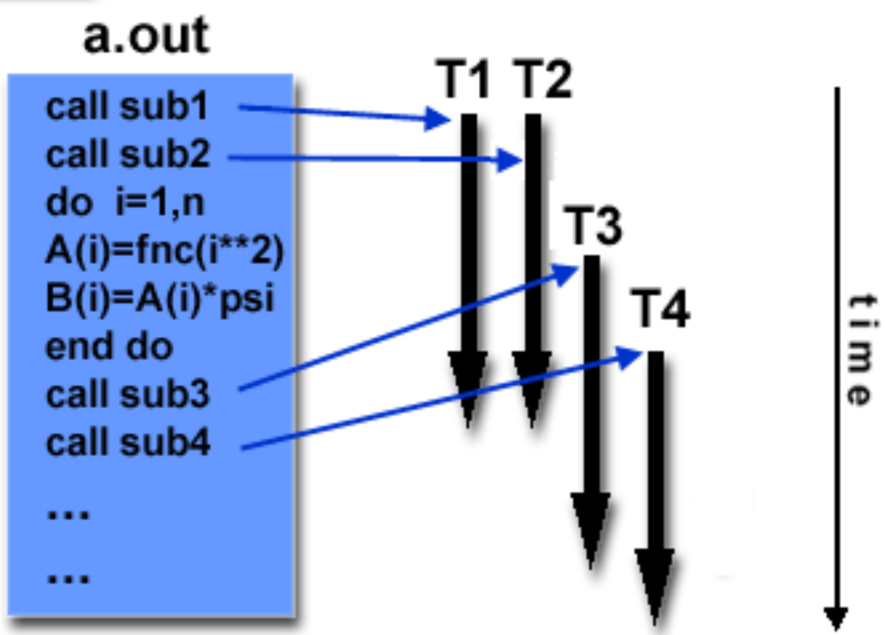
Department of Computer Science and Engineering
University of South Florida
Tampa, FL 33620
Email: haozheng@usf.edu
Phone: (813)974-4757
Fax: (813)974-5456

Overview of Concurrent Systems

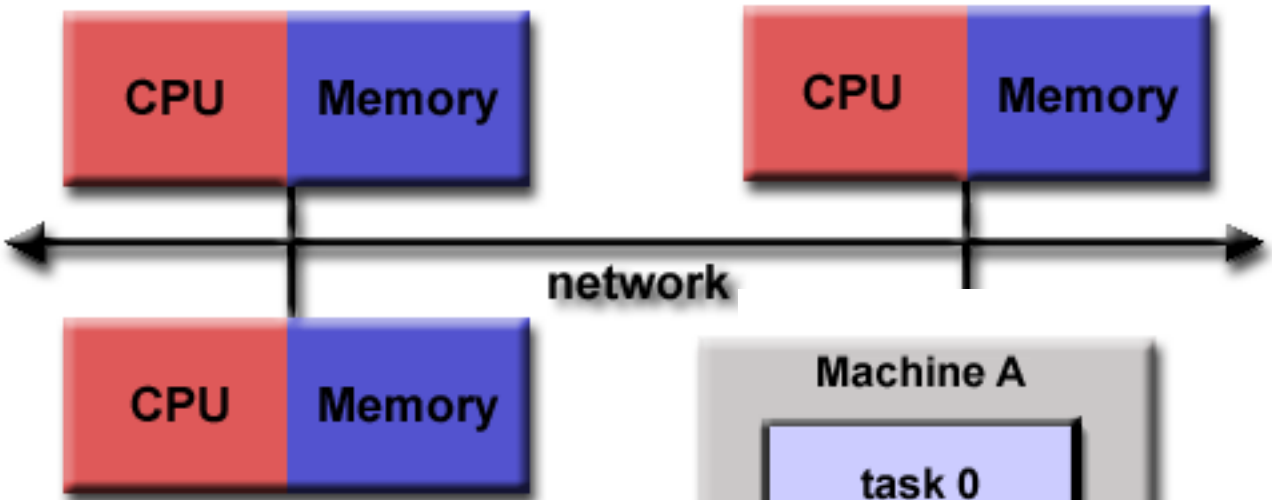


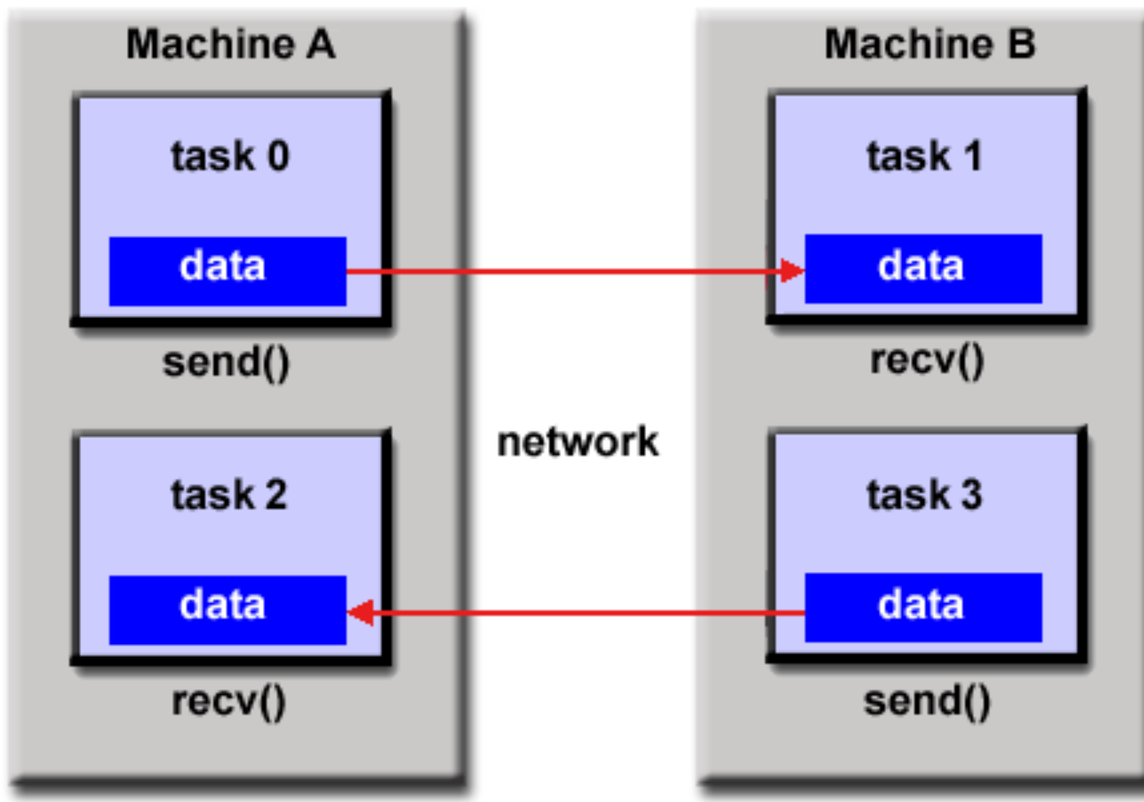
Shared Memory Model





Distributed Memory Model





SPIN & PROMELA

- SPIN is an explicit model checker
  - State space represented as a directed graph
  - Can also perform random simulation
- PROMELA is the modeling language for SPIN
- A model is a set of sequential processes communicating over
  - Global variables for modeling shared memory structures
  - Channels for modeling distributed structures
- PROMELA is NOT a programming language

Download & Install SPIN

- Go to http://spinroot.com/

Modeling Language Promela – Overview

The "Hello World" Example

```
/* hello.pml */
active proctype hello()
{
   printf("Hello world!\n");
}
```

```
>  spin hello.pml
Hello world
```

DEMO

Hello World!

```
/* A "Hello World" Promela model for SPIN. */
active proctype Hello() {
    printf("Hello process, my pid is: %d\n", _pid);
}
init {
    int lastpid;
    printf("init process, my pid is: %d\n", _pid);
    lastpid = run Hello();
    printf("last pid was: %d\n", lastpid);
}
```

instantiate a copy of process Hello

random seed

```
$ spin -n2 hello.pr
init process, my pid is: 1
        last pid was: 2
Hello process, my pid is: 0
                Hello process, my pid is: 2
3 processes created
```

running SPIN in random simulation mode

Promela Model Structure

- Promela model consist of:
  - type declarations
  - channel declarations
  - variable declarations
  - process declarations
  - [`init` process]
- A Promela model corresponds with a (usually very large, but) finite transition system, so
  - no unbounded data
  - no unbounded channels
  - no unbounded processes
  - no unbounded process creation

```
mtype = {MSG, ACK};
chan toS = ...
chan toR = ...
bool flag;

proctype Sender() {
   ...
}

proctype Receiver() {
   ...
}

init {
   ...
}
```

process body

creates processes

Processes – 1

- A process type (**`proctype`**) consist of
  - a name
  - a list of formal parameters
  - local variable declarations
  - body

name

formal parameters

```
proctype Sender(chan in; chan out) {
    bit sndB, rcvB;
    do
    :: out ! MSG, sndB ->
            in ? ACK, rcvB;
            if
            :: sndB == rcvB -> sndB = 1-sndB
            :: else -> skip
            fi
    od
}
```

local variables

body

The body consist of a sequence of statements.

Processes – 2

- A process
  - is defined by a `proctype` definition
  - executes concurrently with all other processes, independent of speed of behaviour
  - communicate with other processes
    - using global (shared) variables
    - using channels
- There may be several processes of the same type.
- Each process has its own local state:
  - process counter (location within the `proctype`)
  - contents of the local variables

A Simple Multi-Thread Program

```
byte a;  // global variable

active proctype p1() {
    byte b = 0;  // local variable
    a=1;
    b=a+b
}

active proctype p2() {
    a=2;
}
```

Processes – 3

- Process are created using the **run** statement (which returns the process id).
- Processes can be created at any point in the execution (within any process).
- Processes start executing after the **run** statement.
- Processes can also be created by adding **active** in front of the **proctype** declaration.

```
proctype Foo(byte x) {
   ...
}

init {
   int pid2 = run Foo(2);
   run Foo(27);
}

active[3] proctype Bar() {
   ...
}
```

number of procs. (opt.)

parameters will be initialised to 0

Variables & Types – 1

- Five different (integer) basic types.
- Arrays
- Records (structs)
- Type conflicts are detected at runtime.
- Default initial value of basic variables (local and global) is 0.

Basic types

`bit turn=1;`	[0..1]
`bool flag;`	[0..1]
`byte counter;`	[0..255]
`short s;`	$[-2^{16}-1..\ 2^{16}-1]$
`int msg;`	$[-2^{32}-1..\ 2^{32}-1]$

Arrays

```
byte a[27];
bit  flags[4];
```

array indicing start at 0

Typedef (records)

```
typedef Record {
  short f1;
  byte  f2;
}
Record rr;
rr.f1 = ..
```

variable declaration

Variables & Types – 2

- Variables should be declared.
- Variables can be given a value by:
  - assignment
  - argument passing
  - message passing (see communication)
- Variables can be used in expressions.

Most arithmetic, relational, and logical operators of C/Java are supported, including bitshift operators.

```
int ii;
bit bb;

bb=1;
ii=2;

short s=-1;

typedef Foo {
  bit bb;
  int ii;
};
Foo f;
f.bb = 0;
f.ii = -2;

ii*s+27 == 23;
printf("value: %d", s*s);
```

assignment =

declaration + initialisation

equal test ==

Statements – Specifying Behavior

Statements – 1

- The body of a process consists of a sequence of statements. A statement is either
  - executable: the statement can be executed immediately.
  - blocked: the statement cannot be executed.

executable/blocked depends on the global state of the system.

- An assignment is always executable.
- An expression is also a statement; it is executable if it evaluates to non-zero.

`2 < 3`	always executable
`x < 27`	only executable if value of `x` is smaller `27`
`3 + x`	executable if `x` is not equal to `–3`

Peterson’s Algorithm for Mutual Exclusion

```
bool turn, flag[2];
byte cnt;
active [2] proctype proc() {
        pid i,j;
        i = _pid;      // acquire pid of the calling proc
        j = 1 - _pid; // pid of the other process
again:  flag[i]=true;
        turn=i;
        (flag[j]==false || turn !=i) ->
        cnt++;    //enter critical section
        assert(cnt==1); //only one proc can be in critical section
        cnt – –;   //exit critical section
        goto again;
}
```

Statements – 2

- `assert(<expr>);`
  - The `assert`-statement is always executable.
  - If `<expr>` evaluates to zero, SPIN will exit with an error, as the `<expr>` “has been violated”.
  - The `assert`-statement is often used within Promela models, to check whether certain properties are valid in a state.

```
proctype monitor() {
  assert(n <= 3);
}

proctype receiver() {
  ...
  toReceiver ? msg;
  assert(msg != ERROR);
  ...
}
```

Statement – goto

```
goto label
```

– transfers execution to `label`
– each Promela statement might be labelled
– quite useful in modelling communication protocols

```
wait_ack:
  if
  :: B?ACK -> ab=1-ab ; goto success
  :: ChunkTimeout?SHAKE ->
     if
     :: (rc < MAX)  -> rc++; F!(i==1),(i==n),ab,d[i];
                       goto wait_ack
     :: (rc >= MAX) -> goto error
     fi
  fi ;
```

Timeout modelled by a channel.

Part of model of BRP

if-statement (1)

inspired by:
Dijkstra's guarded command language

```
if
:: choice1 -> stat1.1; stat1.2; stat1.3; …
:: choice2 -> stat2.1; stat2.2; stat2.3; …
:: …
:: choicen -> statn.1; statn.2; statn.3; …
fi;
```

- If there is at least one **choice**$_i$ (guard) executable, the **if**-statement is executable and SPIN non-deterministically chooses one of the executable choices.
- If no **choice**$_i$ is executable, the **if**-statement is blocked.
- The operator "**->**" is equivalent to "**;**". By convention, it is used within **if**-statements to separate the guards from the statements that follow the guards.

if-statement (2)

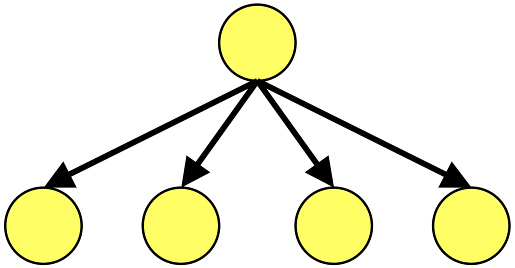
```
if
:: (n % 2 != 0) -> n=1
:: (n >= 0)      -> n=n-2
:: (n % 3 == 0) -> n=3
:: else          -> skip
fi
```

- The **else** guard becomes executable if none of the other guards is executable.

give **n** a random value

```
if
:: skip -> n=0
:: skip -> n=1
:: skip -> n=2
:: skip -> n=3
fi
```

non-deterministic branching



skips are redundant, because assignments are themselves always executable...

IF-Statement

```
if
::  a > b   -> c++
::  else    -> c=0
fi
```

```
if
::  a > b   -> c++
::  true    -> c=0
fi
```

- The **else** branch is selected only when all other branches are not selected
- The **true** branch is always selected
- The above if statements are always executable

do-statement (1)

```
do
:: choice1 -> stat1.1; stat1.2; stat1.3; …
:: choice2 -> stat2.1; stat2.2; stat2.3; …
:: …
:: choicen -> statn.1; statn.2; statn.3; …
od;
```

- With respect to the choices, a `do`-statement behaves in the same way as an `if`-statement.
- However, instead of ending the statement at the end of the choosen list of statements, a `do`-statement repeats the choice selection.
- The (always executable) `break` statement exits a `do`-loop statement and transfers control to the end of the loop.

Statement do-od

enumeration type

```
/* Traffic light controller */
mtype = {RED, GREEN YELLOW};

active proctype TrafficLight() {
   do
   :: (state == RED) -> state = GREEN;
   :: (state == GREEN) -> state = YELLOW;
   :: (state == YELLOW) -> state = RED;
   od
}
```

Statements (2)

Statements are separated by a semi-colon: “;”.

- The **skip** statement is always executable.
  - “does nothing”, only changes process’ process counter
- A **run** statement is only executable if a new process can be created (remember: the number of processes is bounded).
- A **printf** statement is always executable (but is not evaluated during verification, of course).

```
int x;
proctype Aap()
{
  int y=1;
  skip;
  run Noot();
  x=2;
  x>2 && y==1;
  skip;
}
```

Executable if **Noot** can be created…

Can only become executable if a some other process makes **x** greater than 2.

Conditional Expressions

```
max = (a > b -> a : b)
```

- Conditional expressions must be contained in parentheses.
- The following causes syntax errors

```
max = a > b -> a : b
```

Promela Model

- A Promela model consist of:
  - type declarations
  - channel declarations
  - global variable declarations
  - process declarations
  - [`init` process]

`mtype`, typedefs, constants

`chan ch = [`*dim*`] of {`*type, ...*`}`
asynchronous: *dim* > 0
rendez-vous: *dim* == 0

can be accessed by all processes

behaviour of the processes: local variables + statements

initialises variables and starts processes

Promela statements

are either executable or blocked

`skip`	always executable
`assert(`*<expr>***`)`**	always executable
expression	executable if not zero
assignment	always executable
`if`	executable if at least one guard is executable
`do`	executable if at least one guard is executable
`break`	always executable (exits **`do`**-statement)
send **`(ch!)`**	executable if channel **`ch`** is not full
receive **`(ch?)`**	executable if channel **`ch`** is not empty

macros - cpp preprocessor

- Promela uses **cpp**, the C preprocessor to preprocess Promela models. This is useful to define:
  - constants

```
#define MAX 4
```

All cpp commands start with a hash: #define, #ifdef, #include, etc.

  - macros

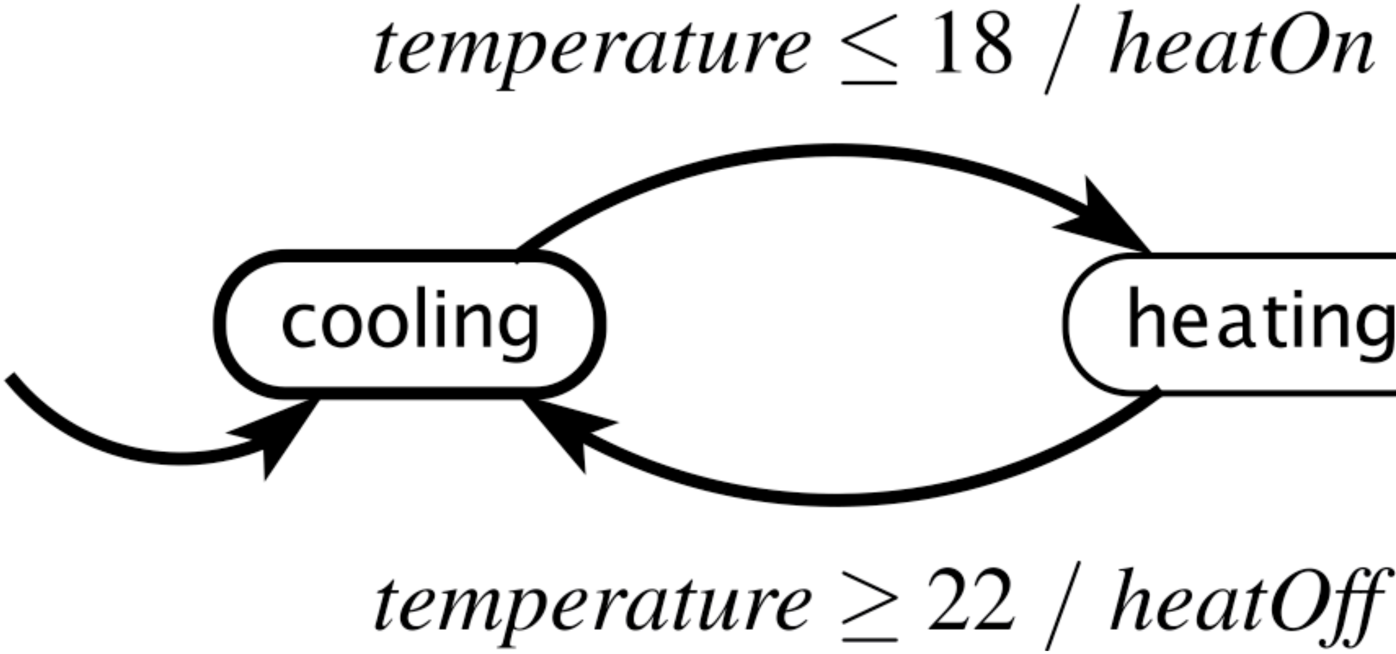
```
#define RESET_ARRAY(a) \
    d_step { a[0]=0; a[1]=0; a[2]=0; a[3]=0; }
```

  - conditional Promela model fragments

```
#define LOSSY 1
…
#ifdef LOSSY
active proctype Daemon() { /* steal messages */ }
#endif
```

A FSM Example

input: $temperature : \mathbb{R}$
outputs: $heatOn, heatOff$: pure



```
#define cooling 0
#define heating 1

proctype thermalStat(byte temp) {
    byte state = cooling;
    bool heaton = false;
    bool heatoff = false;
    do
    ::  state==cooling && temp <= 18 ->
                heaton = true;
                heatoff = false;
                state = heating;
    ::  state==heating && temp >= 22 ->
                heaton = false;
                heatoff = true;
                state = cooling;
    od;
}
```

```
proctype thermalStat(byte temp) {
    bool heaton = false;
    bool heatoff = false;

cooling:  temp <= 18 ->
                heaton = true;
                heatoff = false;
                goto heating;
heating:  temp >= 22 ->
                heaton = false;
                heatoff = true;
                goto cooling;
}
```

Another Example

```
int x = 0;

proctype Inc() {  do
                  ::true -> if :: (x < 200) -> x = x+1 fi
                  od
}
proctype Dec() {  do
                  :: true -> if :: (x > 0) -> x = x-1 fi
                  od
}
proctype Reset() {do
                  :: true -> if :: (x == 200) -> x = 0 fi
                  od
}
proctype Check() { assert (x >= 0 && x <= 200) }

init { atomic {
         run Inc(); run Dec(); run Reset(); run Ceck();
}}
```

Operational Semantics

Interleaving

- Processes execute concurrently
- A process is suspended if
  - next statement is blocked
- Only one process executes at a time.
  - Process executions are interleaved
- Execution scheduling is non-deterministic.
- Each basic statement executes atomically
  - e.g. `a = 5`;
- Each process may have more than one statements enabled to execute.
  - Selection is non-deterministic

Why this Example Fails?

```
int x = 0;

proctype Inc() {  do
                  ::true -> if :: (x < 200) -> x = x+1 fi
                  od
}
proctype Dec() {  do
                  :: true -> if :: (x > 0) -> x = x-1 fi
                  od
}
proctype Reset() {do
                  :: true -> if :: (x == 200) -> x = 0 fi
                  od
}
proctype Check() { assert (x >= 0 && x <= 200) }

init { atomic {
         run Inc(); run Dec(); run Reset(); run Ceck();
}}
```

What happens when x = 200?

Why this Example Fails?

```
int x = 0;

proctype Inc() {  do
                  ::true -> if :: (x < 200) -> x = x+1 fi
                  od
}
proctype Dec() {  do
                  :: true -> if :: (x > 0) -> x = x-1 fi
                  od
}
proctype Reset() {do
                  :: true -> if :: (x == 200) -> x = 0 fi
                  od
}
proctype Check() { assert (x >= 0 && x <= 200) }

init { atomic {
         run Inc(); run Dec(); run Reset(); run Ceck();
}}
```

x == 0

Why this Example Fails?

```
int x = 0;

proctype Inc() {  do
                  ::true -> if :: (x < 200) -> x = x+1 fi
                  od
}
proctype Dec() {  do
                  :: true -> if :: (x > 0) -> x = x-1 fi
                  od
}
proctype Reset() {do
                  :: true -> if :: (x == 200) -> x = 0 fi
                  od
}
proctype Check() { assert (x >= 0 && x <= 200) }

init { atomic {
         run Inc(); run Dec(); run Reset(); run Ceck();
}}
```

$x == -1$

Atomic Sequences

```
atomic { stmt1; stmt2, ..., stmtn}
```

- Group statements in an atomic sequence; all statements are executed in a single step.
- If $\texttt{stmt}_i$ is blocked, the sequence is suspended.

```
d_step { stmt1; stmt2, ..., stmtn}
```

- More efficient than atomic: no intermediate states are generated.
- Only the first statement in the sequence $\texttt{stmt}_1$ can be blocked.
- It is a runtime error if $\texttt{stmt}_i$ $(i > 1)$ is blocked.
- No **goto** or **break** statements in the sequence.

Atomic Sequences: Example

```
int x = 0;

proctype Inc() {
   do
   :: true -> atomic { if :: (x < 200) -> x = x+1 fi }
   od
}
proctype Dec() {
   do
   :: true -> atomic { if :: (x > 0) -> x = x-1 fi }
   od
}
proctype Reset() {
   do
   :: true -> atomic { if :: (x == 200) -> x = 0 fi }
   od
}
...
```

Interleaving or Not?

- Using atomic reduces interleavings
  - Can eliminate errors caused by interleavings
  - Can also reduce state space significantly
  - e.g. 8400 states (no atomic) vs 4010 states (w. atomic) for the previous example.
- Whether to use atomic is a modeling decision.
  - Need to consider the granularity of execution of individual threads.

Conditional Execution of Processes

```
byte a, b;
active proctype A() provided (a > b)  {
    ...
}
```

- Process A() is executable in any global state where (a>b) evaluates to true
- Can be used to schedule executions of process
  - Avoid non-deterministic interleavings

System States

- A system state is uniquely defined by a state vector, which consists of
  - All global variables
  - Contents of all message channels
  - Local states of all processes
    - All local variables
    - Process counters
- It is important to minimize size of state vector

Modeling Inter-Process Communications

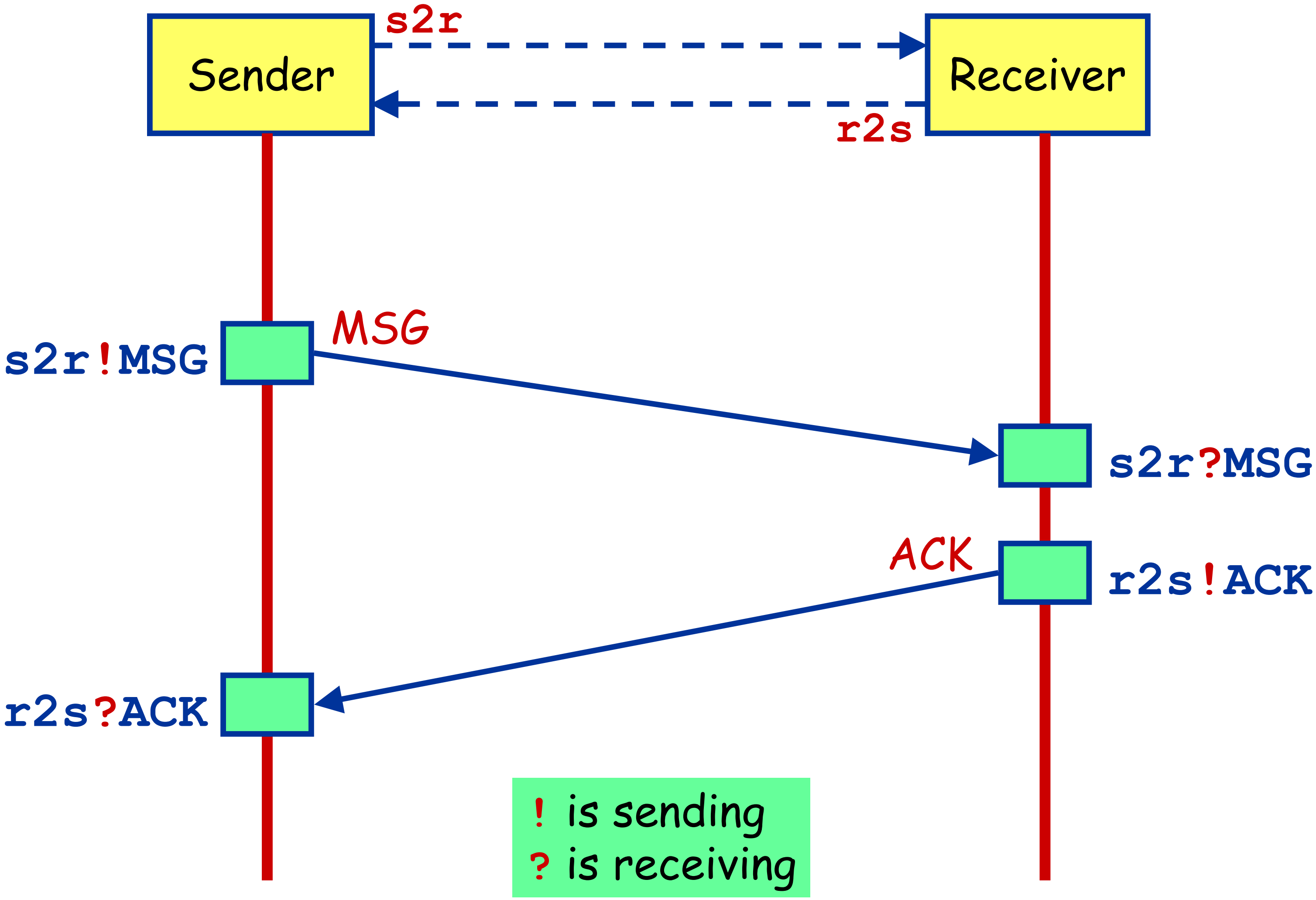
Communication via Shared Variables

```
bit  x, y;      /* signal entering/leaving the section  */
byte mutex;     /* # of procs in the critical section.  */
byte turn;      /* who's turn is it?                     */

active proctype A() {              active proctype B() {
  x = 1;                             y = 1;
  turn = B_TURN;                     turn = A_TURN;
  y == 0 ||                          x == 0 ||
    (turn == A_TURN);                  (turn == B_TURN);
  mutex++;                           mutex++;
  mutex--;                           mutex--;
  x = 0;                             y = 0;
}                                  }

active proctype monitor() {
  assert(mutex != 2);
}
```

Can be generalised to a single process.

First "software-only" solution to the mutex problem (for two processes).

Communications by Channels



Communications by Channels

- Communication between processes is via channels:
  - message passing
  - rendez-vous synchronisation (handshake)
- Both are defined as channels:

also called:
queue or buffer

chan <name> = [<dim>] of {<t_1>,<t_2>, … <t_n>};

name of
the channel

type of the elements that will be
transmitted over the channel

number of elements in the channel
dim==0 is special case: rendez-vous

```
chan c       = [1] of {bit};
chan toR     = [2] of {mtype, bit};
chan line[2] = [1] of {mtype, Record};
```

array of
channels

Communications by Channels

- channel = FIFO-buffer (for `dim>0`)

! Sending - *putting a message into a channel*

ch ! <expr$_1$>, <expr$_2$>, … <expr$_n$>;

- The values of **<expr$_i$>** should correspond with the types of the channel declaration.
- A send-statement is executable if the channel is not full.

? Receiving - *getting a message out of a channel*

<var> + <const> can be mixed

ch ? <var$_1$>, <var$_2$>, … <var$_n$>;

message passing

- If the channel is not empty, the message is fetched from the channel and the individual parts of the message are stored into the **<var$_i$>**s.

ch ? <const$_1$>, <const$_2$>, … <const$_n$>;

message testing

- If the channel is not empty and the message at the front of the channel evaluates to the individual **<const$_i$>**, the statement is executable and the message is removed from the channel.

Communications by Channels

- Rendez-vous communication

  `<dim>` == 0

  The number of elements in the channel is now zero.

  – If send **`ch!`** is enabled and if there is a corresponding receive **`ch?`** that can be executed simultaneously and the constants match, then both statements are enabled.

  – Both statements will "handshake" and together take the transition.

- *Example:*

  ```
  chan ch = [0] of {bit, byte};
  ```

  – P wants to do **`ch ! 1, 3+7`**

  – Q wants to do **`ch ? 1, x`**

  – Then after the communication, **`x`** will have the value **`10`**.

Predefined Functions for Channels

- **`len(c)`**: returns the number of messages in c.
- **`empty(c)`**: return true if channel c is empty.
- **`nempty(c)`**: return true if channel c is not empty.
  - Writing **`!empty(c)`** not allowed in Promela.
- **`full(c)`**: return true if channel c contains the maximal number of messages.
- **`nfull(c)`**: return true if channel c is not full.
  - Writing **`!full(c)`** not allowed in Promela.
- More details on predefined functions can be found at

  http://spinroot.com/spin/Man/promela.html#section5

Randez-vous Communication Example

```
mtype { red, yellow, green };
chan ch = [0] of { mtype, byte, bool };

active proctype Sender() {
  ch ! red, 20, false;
  printf("Sent message\n")
}

active proctype Receiver() {
  mtype color;
  byte time;
  bool flash;
  ch ? color, time, flash;
  printf("Received message %e, %d, %d\n",
         color, time, flash)
}
```

Sender   Receiver

⋮   ⋮

ch!(red ,20,false) ⟶ ch ? color, time, flash;

⋮   ⋮

Randez-vous Communication Example

```
mtype { red, yellow, green };
chan ch = [0] of { mtype, byte, bool };

active proctype Sender() {
  ch ! red, 20, false;
  printf("Sent message\n")
}

active proctype Receiver() {
  mtype color;
  byte time;
  bool flash;
  ch ? color, time, flash;
  printf("Received message %e, %d, %d\n",
           color, time, flash)
}
```

Sender	Receiver
⋮	⋮
ch!(red ,20,false) ⟶	ch ? color, time, flash;
⋮	⋮

Randez-vous Communication Example

```
mtype { red, yellow, green };
chan ch = [0] of { mtype, byte, bool };

active proctype Sender() {
  ch ! red, 20, false;
  printf("Sent message\n")
}

active proctype Receiver() {
  mtype color;
  byte time;
  bool flash;
  ch ? color, time, flash;
  printf("Received message %e, %d, %d\n",
         color, time, flash)
}
```

Sender	Receiver
⋮	⋮
ch!(red ,20,false) →	ch ? color, time, flash;
⋮	⋮

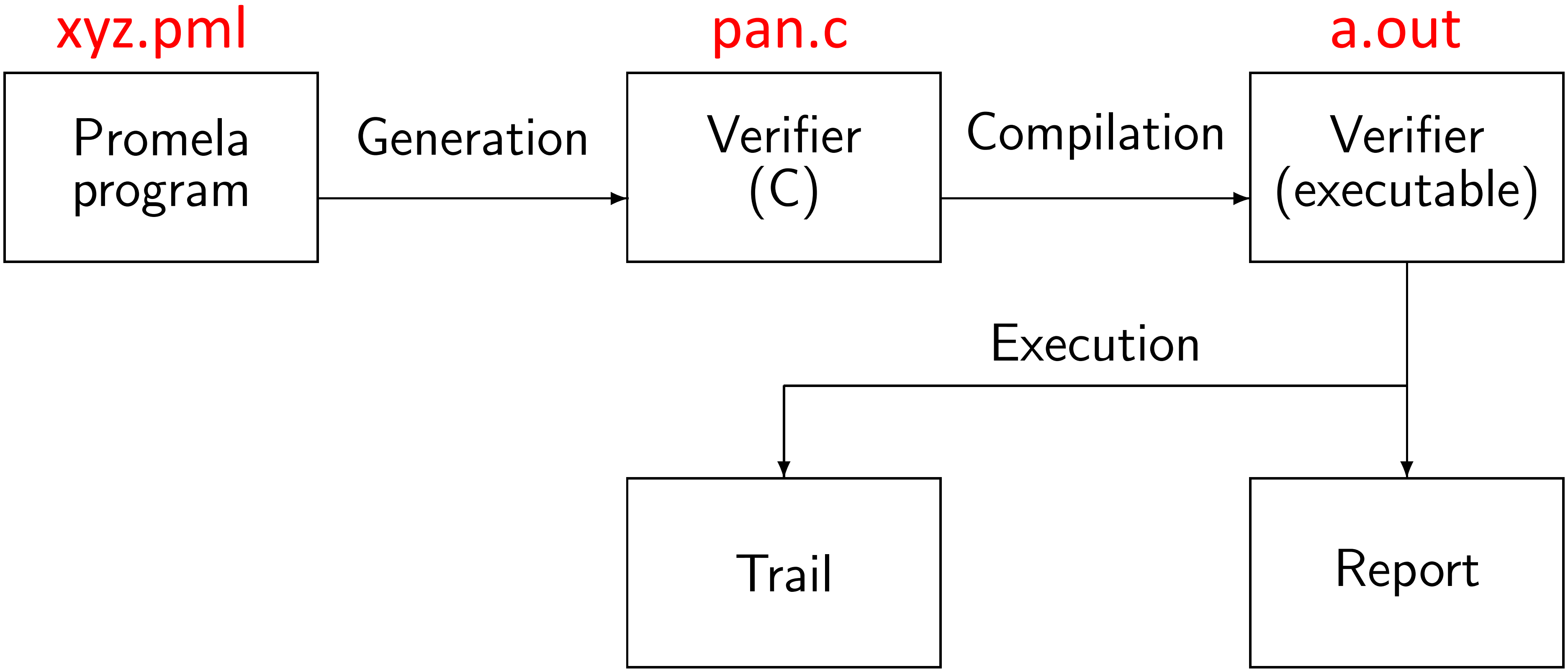
Randez-vous Communication Example

```
mtype { red, yellow, green };
chan ch = [0] of { mtype, byte, bool };

active proctype Sender() {
  ch ! red, 20, false;
  printf("Sent message\n")
}

active proctype Receiver() {
  mtype color;
  byte time;
  bool flash;
  ch ? color, time, flash;
  printf("Received message %e, %d, %d\n",
          color, time, flash)
}
```

```
Sender                                   Receiver
  :                                         :
ch!( red ,20,false) ----------------> ch ? color, time, flash;
  :                                         :
```

Use of SPIN

Architecture of SPIN



How to Run SPIN

/* Use SPIN to generate a verification model in pan.c */

```
../Src6.2.5/spin -a model.pml
```

/* Compile pan.c to an executable */

```
gcc -O2 -DNOFAIR -DNOREDUCE -DSAFETY -o pan pan.c
```

/* Run the executable */

```
./pan
```

SPIN Output

```
pan:1: invalid end state (at depth 188)
pan: wrote hw3-p2.pml.trail

(Spin Version 6.2.5 -- 3 May 2013)
Warning: Search not completed

Full statespace search for:
never claim          - (none specified)
assertion violations +
cycle checks         - (disabled by -DSAFETY)
invalid end states +

State-vector 36 byte, depth reached 263, errors: 1
      453 states, stored
      192 states, matched
      645 transitions (= stored+matched)
       65 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
    0.024 equivalent memory usage for states (stored*(State-vector + overhead))
    0.285 actual memory usage for states
  128.000 memory used for hash table (-w24)
    0.458 memory used for DFS stack (-m10000)
  128.653 total actual memory usage
```

SPIN Output: Dissection

```
pan:1: invalid end state (at depth 188)
pan: wrote hw3-p2.pml.trail

(Spin Version 6.2.5 -- 3 May 2013)
Warning: Search not completed

Full statespace search for:
never claim             - (none specified)
assertion violations +
cycle checks           - (disabled by -DSAFETY)
invalid end states +
```

SPIN Output: Dissection

```
State-vector 36 byte, depth reached 263, errors: 1
      453 states, stored
      192 states, matched
      645 transitions (= stored+matched)
       65 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
    0.024 equivalent memory usage for states (...)
    0.285 actual memory usage for states
  128.000 memory used for hash table (-w24)
    0.458 memory used for DFS stack (-m10000)
  128.653 total actual memory usage
```

Specification of Requirements

Properties – 1

- Model checking tools automatically verify whether
  $$M \models \phi \quad (M \text{ includes all execution traces})$$
  holds, where M is a (finite-state) model of a system and property ϕ is stated in some formal notation.

- With SPIN one may check the following type of properties:
  – deadlocks (invalid endstates)
  – assertions
  – unreachable code
  – LTL formulae
  – liveness properties
    - non-progress cycles (livelocks)
    - acceptance cycles

Properties – 2

safety property

– "nothing bad ever happens"

– invariant
  x is always less than 5

– deadlock freedom
  the system never reaches a state where no actions are possible

– SPIN: find a trace leading to the "bad" thing. If there is not such a trace, the property is satisfied.

liveness property

– "something good will eventually happen"

– termination
  the system will eventually terminate

– response
  if action X occurs then eventually action Y will occur

– SPIN: find a (infinite) loop in which the "good" thing does not happen. If there is not such a loop, the property is satisfied.

LTL Specification

- LTL formulae are used to specify liveness properties.
  LTL ≡ propositional logic + temporal operators
  - `[]P` always P
  - `<>P` eventually P
  - `P U Q` P is true until Q becomes true

- Some LTL patterns
  - invariance `[] (p)`
  - response `[] ((p) -> (<> (q)))`
  - precedence `[] ((p) -> ((q) U (r)))`
  - objective `[] ((p) -> <>((q) || (r)))`

Checking LTL Properties in SPIN

LTL formula

$\mathbf{G}(a \rightarrow \mathbf{F}b)$

In SPIN

`[](a -> <>b)`

Operator	Math	SPIN
NOT	$\neg$	`!`
AND	$\wedge$	`&&`
OR	$\vee$	`||`
implies	$\rightarrow$	`->`
equal	$\leftrightarrow$	`<->`
always	$\mathbf{G}$	`[]`
eventually	$\mathbf{F}$	`<>`
until	$\mathbf{U}$	`U`
release	R	R

Checking LTL Properties in SPIN

Inline LTL formulas must be placed outside all proctype or init process.

```
ltl [ name ] '{'  formula '}'
```

Inline properties are taken as positive properties that must be satisfied by the model.

Use the following command to generate a pan verifier including the LTL formula to check.

```
spin –a –f `[]p’ x.pml
```

Checking LTL Properties in SPIN

- Store a LTL formula in a one-line file
- Ex: Store !([]p) in file model.prp
- Use the following command to compile the model.

spin –a –F model.prp model.pml

- Note that the inline formula is *positive* while the formula provided on the commandline or from a file is *negative*.

http://spinroot.com/spin/Man/ltl.html