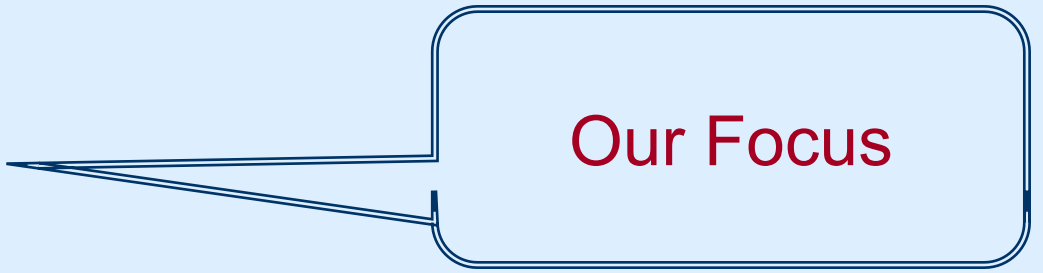
MODEL CHECKING USING SPIN/PROMELA

courtesy:
A K Bhattacharjee, BARC, Mumbai; CFDVS, IIT, Bombay
Scott Smolka, Stony Brook, USA
Theo C Ruys, Univ of Twente
Gerard Holzmann, Design and Validation of Computer Protocols

Formal Verification : Background

- The Process of Formal Verification consists of
  - Requirements Capture
  - Modeling
  - Specification
  - Analysis
  - Documentation



- In reality, some phases may overlap
  - the overall process is iterative rather than sequential

Model Checking

- Model checking
  - is one of the powerful FORMAL VERIFICATION technique.
  - allows one to verify temporal properties of a finite state representation.
- The finite state representation is that of a typical concurrent system…..
- What do we mean by representation here ?
- The representation is a model of the system……..

Model Checking…….

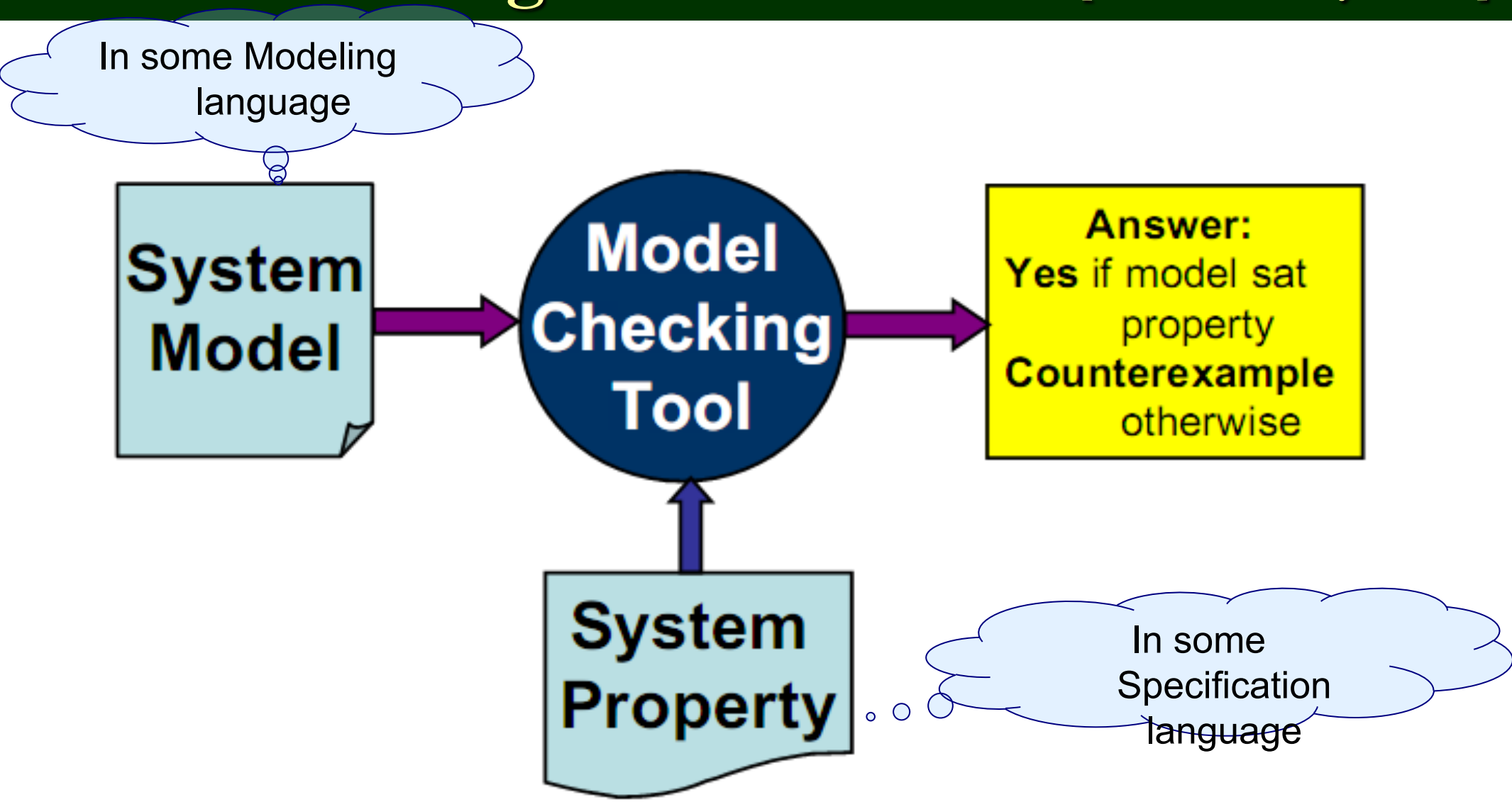
- The basic idea is
  - that a finite state model of a system is systematically explored in order to determine whether or not a given temporal property holds.
  - deliver a counter example if the specified property does not hold.
- State space explosion
  - because a complete state space is generated for a given model, the analysis may fail due lack of memory if the model is too large
  - can be tackled via abstraction.

Model Checking

- Verifying design models
  - against specification for finite state concurrent systems.
- It is an automated technique wherein
  - Inputs
    - finite state model of the system & properties stated in some standard formalism
  - Outputs
    - property valid against the model or not.
- Many Commercial and academic tools
  - Spin (Bell Labs.), Formal-Check (Cadence), VIS (UCB), SMV (CMU, Cadence)
  - In-house tools: Rule Base (IBM), Intel, SUN, Bingo (Fujitsu), etc

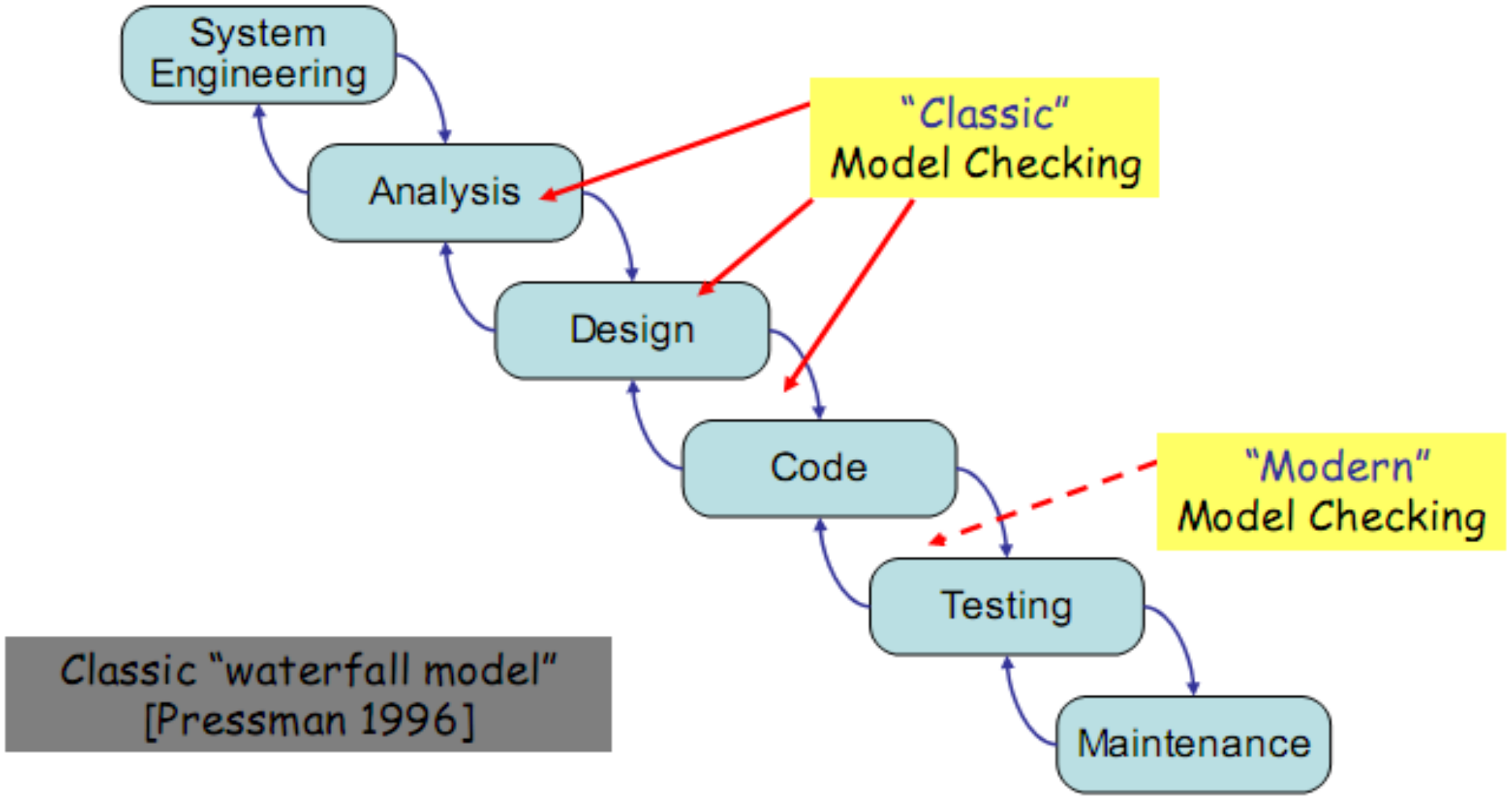
Model Checking

[Smolka-StonyBrook]

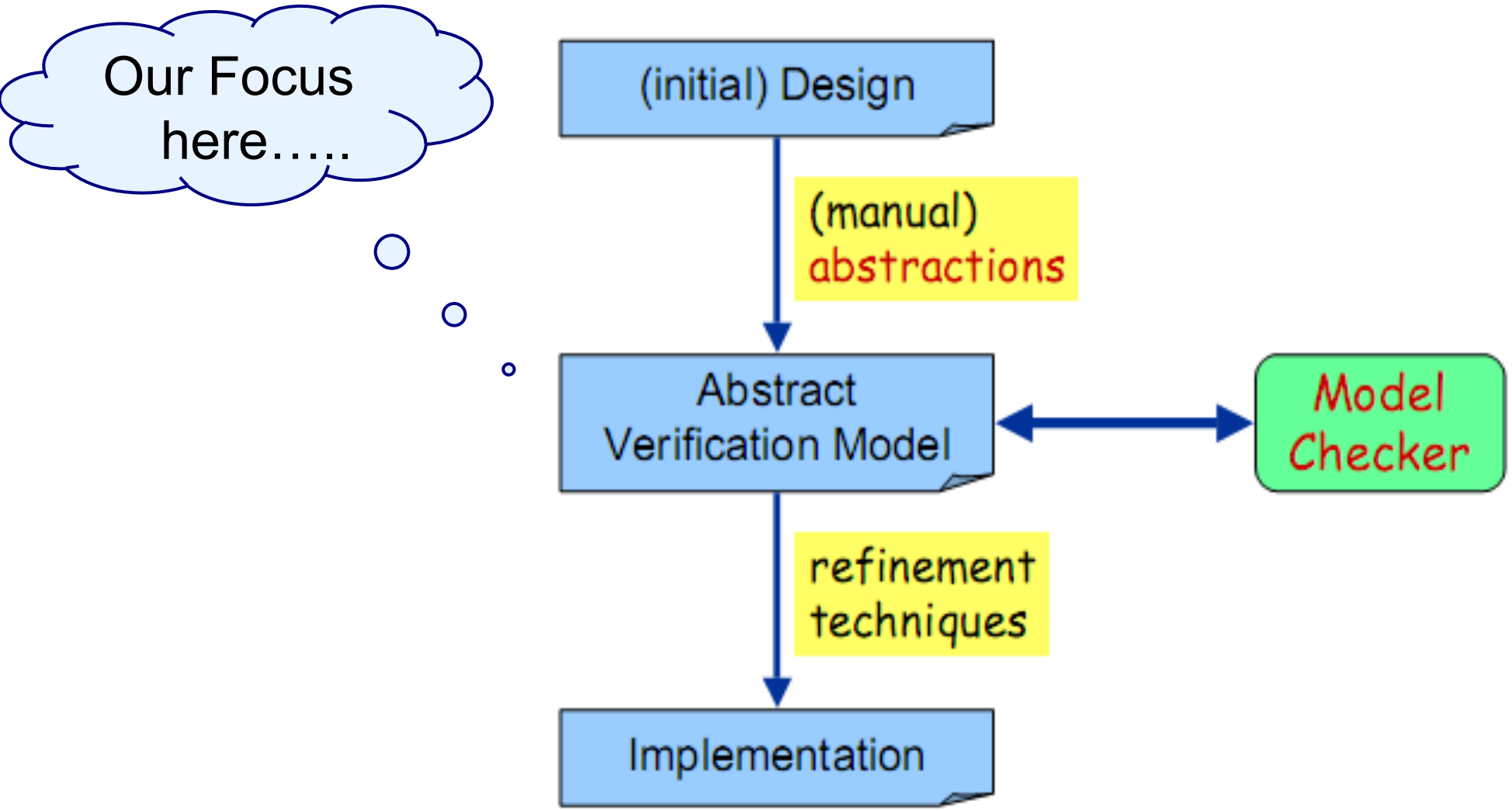


The entire process is automated

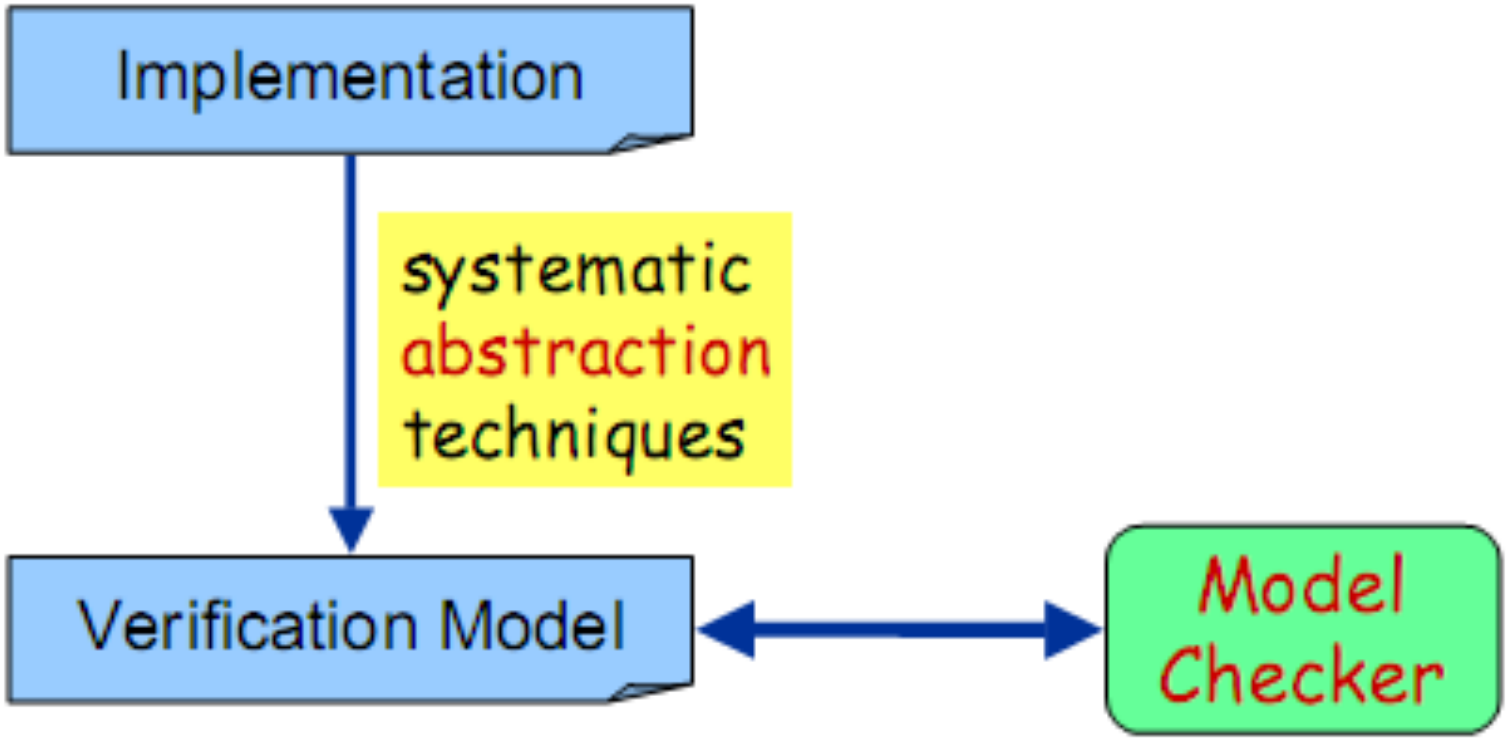
System Development [Smolka-StonyBrook]



Classic Model Checking [Theo Ruys]



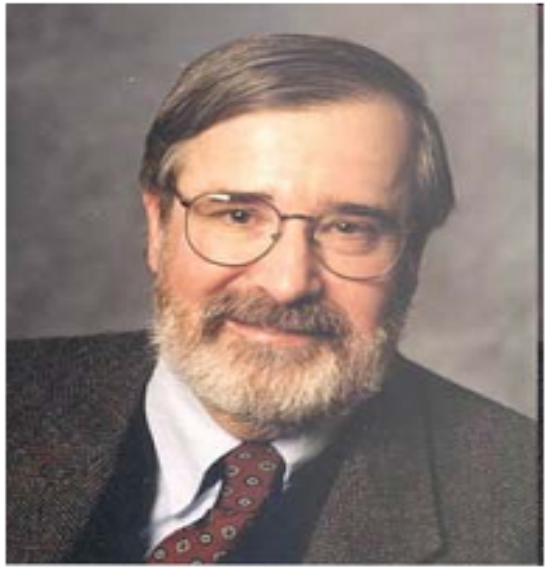
Modern Model Checking [Theo Ruys]



Who developed it?

- Developed independently by Clarke, Emerson, and Sistla and by Queille and Sifakis in early 1980's.

Clarke Emerson Sifakis Receive 2007 Turing Award



... they developed this fully automated approach [Model Checking] that is now the most widely used verification method in the hardware and software industries.

Applicability : Distributed Systems

- Specific concern on the distributed systems
  - Network applications
  - Data communication protocols
  - Multithreaded code
  - Client-Server applications
- Suffer from common design flaws

Common Design Flaws

- Deadlock
- Livelock,
- Starvation
- Race conditions…

What are these ?

How do we distinguish amongst them ?

Common Design Flaws…..

- Deadlock
- Livelock, Starvation
- Underspecification
  - unexpected reception of messages
- Overspecification
  - Dead code
- Violations of constraints
  - Buffer overruns
  - Array bounds violations
- Assumptions about speed
  - Logical correctness vs. real-time performance

Model Checking Definition [Clarke & Emerson 1981]

- "Model checking is an automated technique that,
  - given a finite-state model of a system and
  - a logical property, systematically checks whether this property holds for ("a given initial state in) that model."

$$M \stackrel{?}{\models} \varphi$$

Does system model **M** satisfy system property **φ**?
M given as a ***state machine***, that is Finite-state
φ usually specified in ***temporal logic***.

Model Checking with SPIN

- Involves Three Steps
- modeling
  - convert the design into a formalism to be accepted by the model checking tool SPIN
- specification
  - state the properties that the design must satisfy
  - must be complete
- verification
  - normally based on a tool i.e on SPIN
  - is automatic
  - analysis of verification results is, however, manual.

SPIN Primary Usage : Verifying Protocols

- Protocol definition
  - a service specification
  - explicit assumptions about the environment
  - the protocol vocabulary
  - formal definition
  - procedural rules
    - a protocol validation model
    - describes the interaction of processes in the system
      - in some formal language
    - give only a partial description of protocol. Why?

Introduction to SPIN

- SPIN – Simple Promela INterpreter
  - a tool for analysing the logical consistency of concurrent systems
    - specifically of data communication protocols
  - is a state-of-the-art model checker, used by >2000 users
  - system model of a concurrent system is described in PROMELA
- PROMELA – PROcess MEta Language
  - specification language to model finite-state systems
  - allows for dynamic creation of concurrent processes
  - expressive enough to describe processes, their interactions..
    - in synchronous as well as asynchronous manner
  - resembles the programming language C …plus CSP…inspired by Dijkstra's guarded commands

Introduction to SPIN…

- Best way to get started with SPIN is via XSPIN.
- XSPIN allows
  - To edit the Promela model
  - Simulate the model (Closed)
    - Random
    - Interactive
    - Guided
- Invoking the verifier
  - exhaustive OR bitstate hashing
- LTL Property Manager
- Help

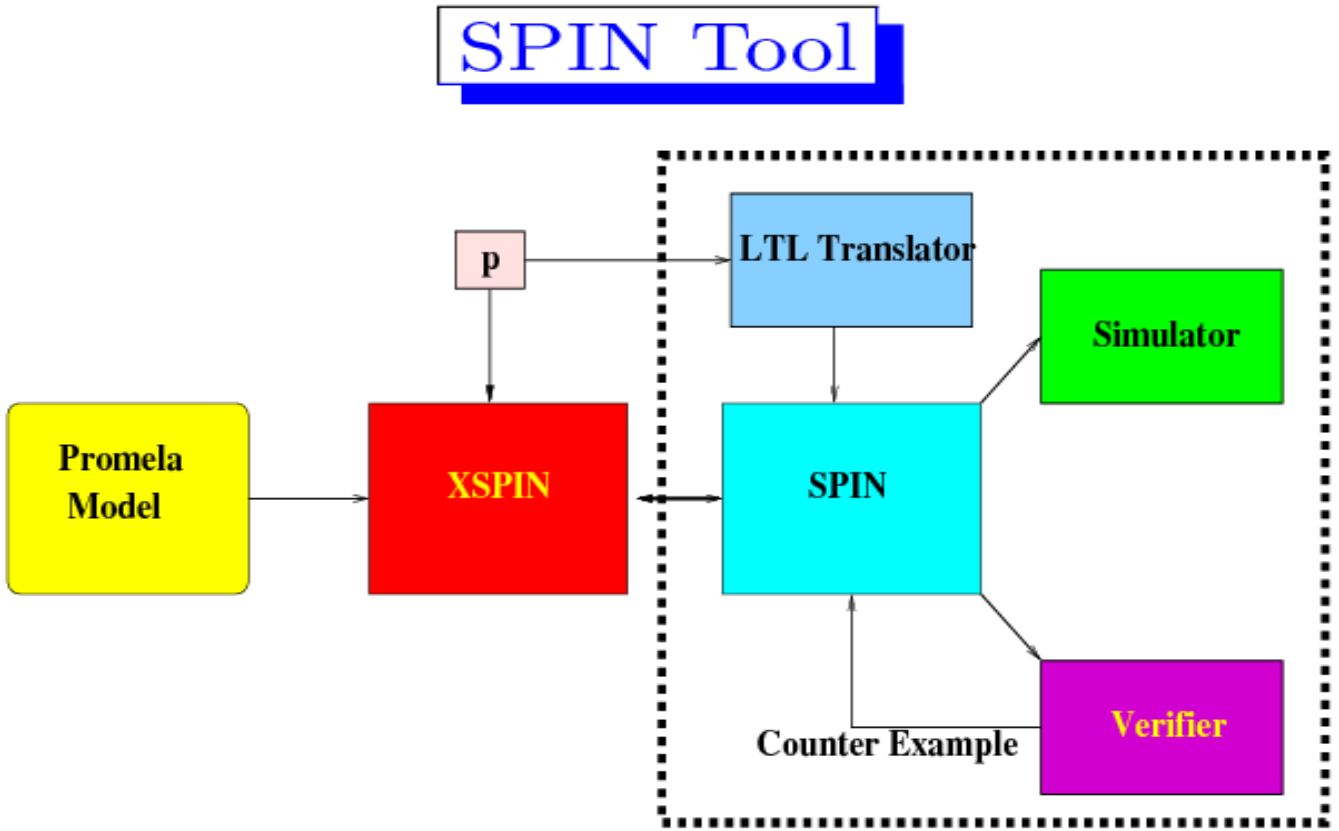
Introduction to SPIN…

- The look-and-feel of XSPIN
  - Main text window
    - basic edit & search capabilities.
  - File handling
    - basic browsing, loading, saving capabilities
  - File editing
    - cut, copy & paste .
  - Tool support
    - syntax checking, simulation, verification, automaton portray.
  - Help facilities

Introduction to SPIN…

- Promela and SPIN/XSPIN are
  - developed by Gerard Holzmann at Bell Labs
  - Freeware for non-commercial use
  - is a State-of-art model checker
    - another is SMV

The SPIN tool



Introduction to SPIN… [Theo Ruys]

- Major versions:

1.0	Jan 1991	initial version [Holzmann 1991]
2.0	Jan 1995	partial order reduction
3.0	Apr 1997	minimised automaton representation
4.0	late 2002	Ax: automata extraction from C code

- Some success factors of SPIN (subjective!):
  - "press on the button" verification (model checker)
  - very efficient implementation (using C)
  - nice graphical user interface (Xspin)
  - not just a research tool, but well supported
  - contains more than two decades research on advanced computer aided verification (many optimization algorithms)

Introduction to SPIN… [Theo Ruys]

- SPIN's starting page:
  http://netlib.bell-labs.com/netlib/spin/whatispin.html
  - Basic SPIN manual
  - Getting started with Xspin
  - Getting started with SPIN
  - Examples and Exercises

  Also part of SPIN's documentation distribution (file: `html.tar.gz`)
  - Concise Promela Reference (by Rob Gerth)
  - Proceedings of all SPIN Workshops
- Gerard Holzmann's website for papers on SPIN:
  http://cm.bell-labs.com/cm/cs/who/gerard/
- SPIN version 1.0 is described in [Holzmann 1991].

PROMELA

- defining a validation model consisting of
  - a set of states incorporating info about values of variables, program counters etc
  - a transition relation
- a representation of a FSM in terms of
  - processes
  - message channels
  - state variables
- only the design of consistent set of rules to govern interaction amongst processes in a DS
- NOT implementation details

PROMELA….

- PROMELA model consists of
  - type declarations
  - channel declarations
  - variable declarations
  - process declarations
  - init process
- As mentioned, PROMELA model = FSM (usually a very large)
  - but it is finite, and hence has
    - no unbounded data
    - no unbounded channels
    - no unbounded processes
    - no unbounded process creations.

Processes

- A process type (proctype) consists of
  - a name, list of formal parameters, declarations of local variables body
- A process
  - executes concurrently with all other processes
  - communicates with other processes using channels & global variables
  - may access shared variables
  - defined by proctype declarations
- Each process has
  - its program counter
  - local variables (local state)

There may be several processes of the same type !!!

PROMELA : An illustration

```
proctype hello(){
  printf("Hello\n")
}
proctype world(){
  printf("World\n")
}
init {
  run hello(); run world ();
}
```

Variables and Basic Data Types

- Promela variables
  - provide the means of storing information about the system being modelled.
  - may hold global information on the system or information that is local to a particular component (process).
  - supports five basic data types

Name	Size (bits)	Usage	Range
`bit`	1	unsigned	$0 \ldots 1$
`bool`	1	unsigned	$0 \ldots 1$
`byte`	8	unsigned	$0 \ldots 255$
`short`	16	signed	$-2^{15} - 1 \ldots 2^{15} - 1$
`int`	32	signed	$-2^{31} - 1 \ldots 2^{31} - 1$

Variables and Basic Data Types…..

- Like all modern well-structured programming languages, Promela also requires
  - that variables must be declared before they can be used
  - variable declarations follow the style of the C programming language ,
    - i.e. a basic data type followed by one or more identifiers and optional initializer

      ```
      byte count, total = 0;
      ```
  - by default all variables of the basic types are initialized to 0.
    - as in C, 0 (zero) is interpreted as false while any non-zero value is interpreted as true .

The `init` process

- All Promela programs must contain an init process
  - is similar to the main() function within a C program
    - i.e. the execution of a Promela program begins with the init process.
- An init process
  - takes the form :

    ```
    init { /*local declarations and statements*/ }
    ```

  - what could it do ???/
    - `init { skip }`
- While a proctype definition declares the behaviour of a process,
  - the instantiation and execution of a process definition is co-ordinated via the init process.

Statements

- A process body consists of sequence of statements.
- A statement is either
  - executable : it can be executed immediately
  - blocked : it cannot be executed
- An assignment statement is always executable

```
x=2;
```

Statements (contd)

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

    `skip,2< 3` always executable

    `x <27` executable if the value of x is less than 27

- A run statement is executable
  - only if the process can be created
  - returns zero if this cannot be done
  - value otherwise returned is a run-time process ID number
  - run() is defined as an operator and so can be embedded in other expressions.

Illustrations

- `run` operator can pass parameter values to a new process

```
proctype A(byte state; short set)
  {   (state==1) → state = set
  }
init {run A(1,3) }
```

- run allows more than one copies of a process to be created

```
byte state = 1;
proctype A() { (state==1) → state = state + 1}
proctype B() { (state==1) → state = state - 1}
init { run A(); run B() }
```

Executability of Statements

- Promela
  - does not make a distinction between a condition and a statement
    - e.g. the simple boolean condition a == b represents a statement in Promela.
  - statements are either executable or blocked.
    - the execution of a statement is conditional on it not being blocked.
  - notion of statement executability provides the basic means by which process synchronization can be achieved.
- e.g.

```
while (a != b) skip /* conventional busy wait */
(a == b) /* Promela equivalent */
```

Hello World

- A simple two process system

```
proctype hello(){ printf("Hello") }
proctype world(){ printf("World\n") }
init { run hello(); run world () }
```

- What does init do here ? What does run do here ?
  - The run operator is executable only
    - if process instantiation is possible.
  - If a run is executable then a pid is returned.
    - the pid for a process can be accessed via the predefined local variable _pid.
- The execution of run does not wait
  - for the associated process to terminate
    - i.e. further applications of run will be executed concurrently.

Process instantiation

- A process can be instantiated also by using active in front of proctype definition i.e.
  - HelloWorld can also be instantiated as

```
active proctype hello(){ printf("Hello") }
active proctype world(){ printf("World\n")}
```

- Multiple instances of the same proctype declaration can be generated using an optional array suffix, e.g.

```
active [4] proctype hello(){ printf("Hello") }
active [7] proctype world(){ printf("World\n") }
```

  - will generate 4 instances of hello and 7 instances of world.

PROMELA : An illustration…

```
proctype Foo(byte x) {
  ….
}

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

active [3] proctype Bar() {
  …
}
```

- a process can be created at any point in the execution (even within any process)
- processes can also be created using active in front of proctype declaration

PROMELA – an illustration….

```
byte state = 2;
proctype A() {
  (state == 1) → (state = 3)
}
proctype B() {
  state = state - 1
}
```

No init process
OR
“active” in
front of proctype
definition

; & → are statement separators

Other Data Types

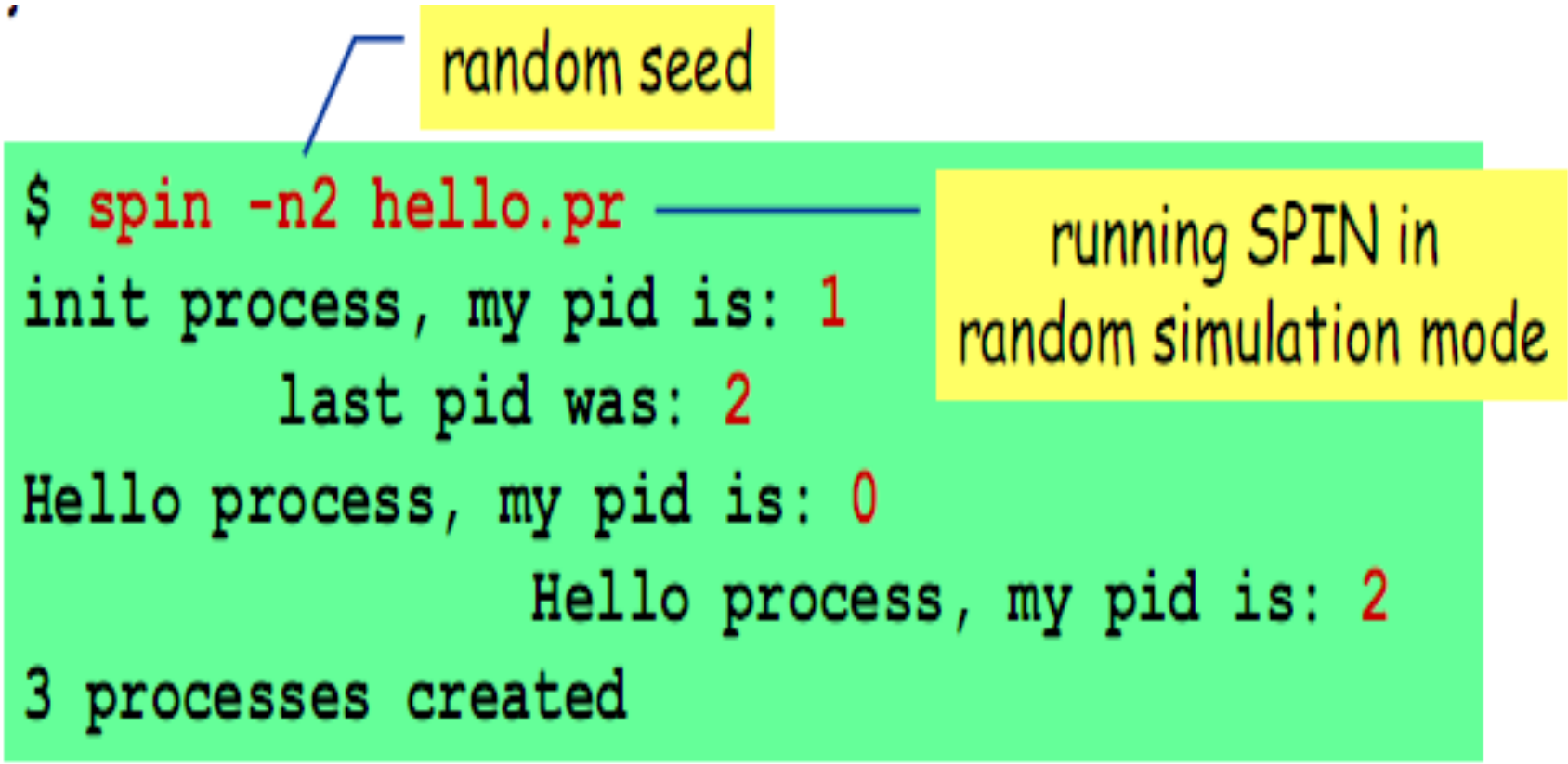
- Arrays
  - an array type is declared as **`int table[max]`**
  - this generates an array of **`max-1`** integers
    - i.e. **`table[0], table[1], ... table[max-1]`**
- Enumerated Types
  - a set of symbolic constants is declared as
  - **`mtype = {LINE_CLEAR, TRAIN_ON_LINE, LINE_BLOCKED}`**
    - a program can only contain one mtype declaration which must be global.
- Structures
  - a reord data type is declared as **`typdef msg {byte data[4], byte checksum}`**
  - Structure access is as in C
    - msg message;
    - ... message.data[0]

PROMELA – An illustration…

```
/* a Hello World PROMELA model for SPIN */
active proctype Hello() {
  printf("Hello process, my PID is : %d",_pid);
}
init {
  int lastPID;
  printf("init process, my PID is : %d", _pid);
  lastPID = run Hello();
  printf("last PID was : %d", _pid);
}
```

- How many processes are created, here ?

PROMELA – An illustration…Sample output



Statement Delimiters

- Two types of statement delimiters
  - ; and ->
  - use the one that is most appropriate at the given situation
  - usually, ; is used between ordinary statements
  - -> is often used after ”guards” in a if OR do statement, pointing at what comes next
  - these can be used interchangably:

Statements: Reviewing

- 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.

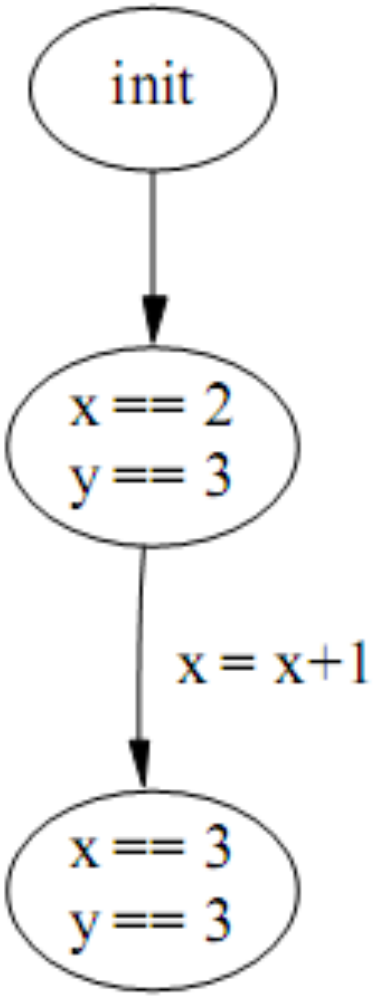
Interleaving Semantics

- Promela processes execute concurrently.
- Non-deterministic scheduling of the processes.
- Processes are interleaved
  - statements of different processes do not occur at the same time.
    - exception: rendezvous communication.
- All statements are atomic;
  - each statement is executed without interleaving with other processes.
- Each process may have several different possible actions enabled at each point of execution
  - only one choice is made, non-deterministically i.e. randomly

Processes as Automata

```
byte x = 2, y = 3;
proctype A(){x = x + 1}
proctype B(){
  x = x - 1;
  y = y + x
}
```

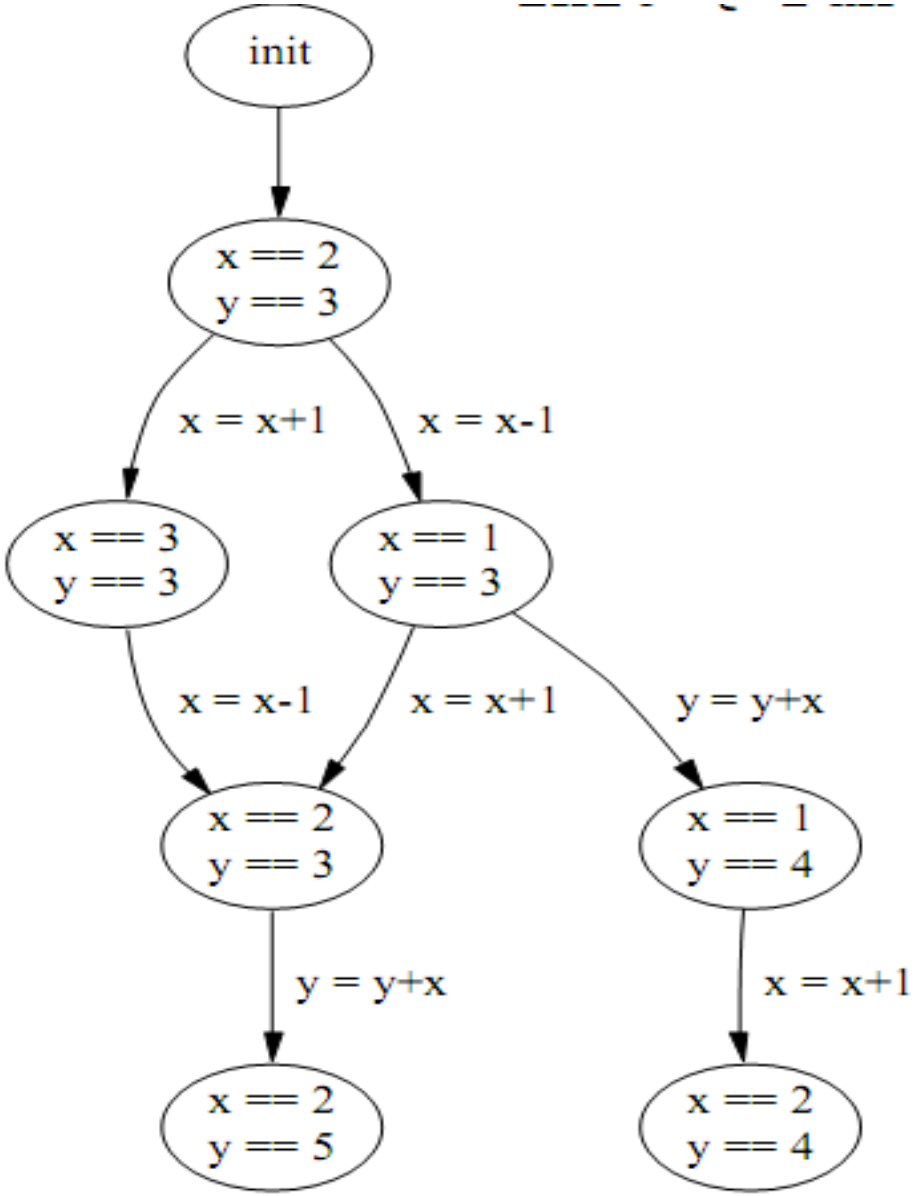
init {run A}; would cause this



Interleaved Execution

```
init {run A(); run B();}
```

```
byte x = 2, y = 3;
proctype A(){x = x + 1}
proctype B(){
  x = x - 1;
  y = y + x
}
```



Deterministic vrs Nondeterministic behaviour

- Deterministic behaviour
  - a process is deterministic if for a given start state…….it behaves in exactly the same way ……if supplied with the same stimuli from its environment.
- Non-deterministic behaviour
  - a process is non-deterministic if it need not always behave in exactly the same way….. each time it executes from a given start state with …..the same stimuli from its environment.

- Hence, race conditions can occur……
- What are race conditions….?

Race conditions

- Solutions
  - use standard mutex algorithms
  - use atomic sequences

```
byte state = 1;
proctype A() {      {(state==1) → state=state + 1}}
proctype B() {      {(state==1) → state=state - 1}}
init { run A(); run B() }
```

What could be the output ?

Race conditions

- Solutions
  - use standard mutex algorithms
  - use atomic sequences

```
byte state = 1;
proctype A() {atomic{(state==1) → state=state + 1}}
proctype B() {atomic{(state==1) → state=state - 1}}
init { run A(); run B() }
```

What could be the output ?

Another illustration: Atomic statements

```
proctype nr(short pid, a, b)
{ int res;
atomic {  res = (a * a + b)/2*a;
          printf("result %d: %d\n", pid,
                                  res)
          }
}
init { run nr(1,1,1); run nr(1,2,2); run
  nr(1,3,2)}
```

Using atomic statements…

```
byte value1 = 1, value2 = 2, value3 = 3;

proctype A() { value3 = value3 + value2;
               assert( value3 == 5 )
}
proctype B() { value2 = value2 + value1;
               assert( value3 == 5 )

}
init { atomic{ run A(); run B()}}
```

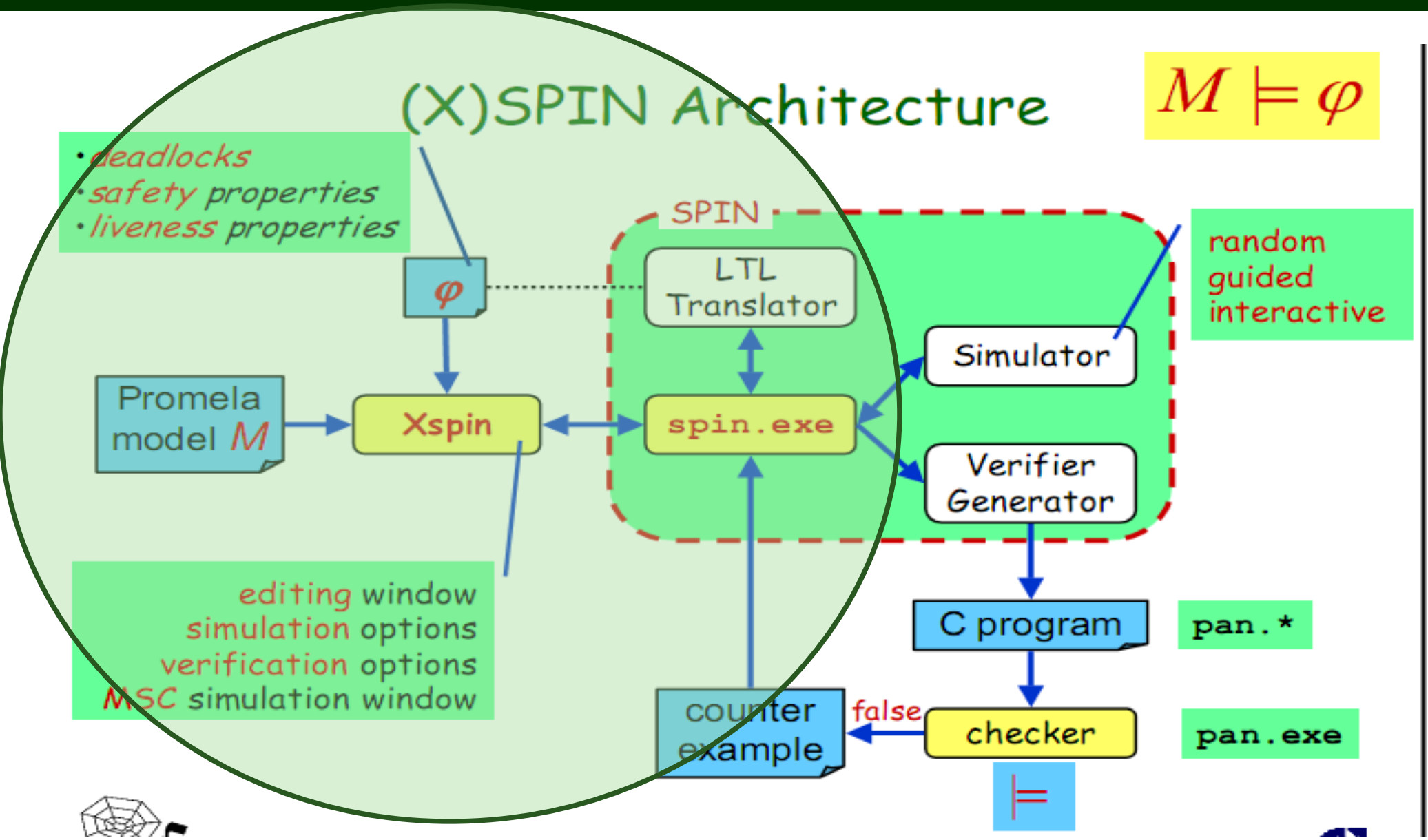
- When can **value3** here not be 5 – i.e. the assertion is violated ?

Using Atomic

- **`atomic`** keyword
  - helps avoid the undesirable interleaving of the PROMELA execution sequences….
  - restricts the level of interleaving and so
    - reduces complexity when it comes to validating a PROMELA model.

- However, **`atomic`** should be used carefully….Why ?

The XSPIN architecture



PROMELA Control Sturctures

- We have studied three ways for achieving control flow
  - Statement sequencing
  - Atomic sequencing
  - Concurrent process execution.
- Promela supports three additional control flow constructs
  - Case selection
  - Repetition
  - Unconditional jumps

Case Selection

```
byte count;
proctype counter()
{
  if
    :: count = count + 1
    :: count = count - 1
  fi
}
```

- chooses one of the executable choices.
- If no choice is executable, the if-statement is blocked.
  - The executability of the first statement (guard) in each sequence determines whether sequence is executed OR not

Case Selection: An illustration

- An example of case selection with guards

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

- If there is at least one choice (guard) executable,
  - the if statement is executable and SPIN non-deterministically chooses one of the alternatives.
- The operator "->" is equivalent to ";".
  - By convention, it is used within if-statements to separate the guards from the statements that follow the guards.

Case Selection: An illustration….

- Guards need not be mutually exclusive

```
if
:: (x >= y) -> max = x;
:: (y >= x) -> max = y;
fi
```

- If x and y are equal then
  - the selection of which statement sequence is executed is decided at random, giving rise to non-deterministic choice.

Repetition

- An example of repetition involving two statement sequences

```
do
    :: (x >= y) -> x = x - y; q = q + 1;
    :: (y > x) -> break;
od
```

- do statement is similar to the if statement…..
  - however, instead of executing a choice ones, it keeps repeating the execution.
  - the (always executable) break statement may be used to exit a do-loop statement and transfers control to the end of the loop.

Repetition

- The first statement sequence denotes the body of the loop
  - while the second denotes the termination condition.
  - termination, however, is not always a desirable property of a system , in particular, when dealing with reactive systems :

```
do
:: (level > max) -> outlet = open;
:: (level < min) -> outlet = close;
od
```

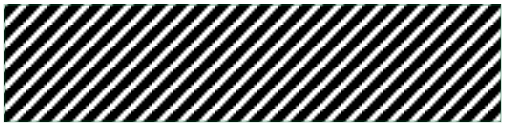
What is a reactive system ?

Repetition : An illustration

```
byte count;
proctype counter()
{
  do
  :: count = count + 1
  :: count = count - 1
  :: (count == 0) -> break
  od
}
```

Name	Size (bits)	Usage	Range
bit	1	unsigned	$0 \ldots 1$
bool	1	unsigned	$0 \ldots 1$
byte	8	unsigned	$0 \ldots 255$
short	16	signed	$-2^{15} - 1 \ldots 2^{15} - 1$
int	32	signed	$-2^{31} - 1 \ldots 2^{31} - 1$

Combining Guards, Case, Repetition for purpose



```
proctype counter()
{
  do
  :: (count != 0) →
        if
        :: count = count + 1
        :: count = count - 1
        fi
  :: (count == 0) → break
  od
}
```

Unconditional Jump

- Promela supports the notion of an unconditional jump via the goto statement.

```
do
    :: (x >= y) -> x = x - y; q = q + 1;
    :: (y > x) -> goto done;
od;
done:
    skip
```

- "done" denotes a label.
  - a label can only app ear before a statement.
  - a goto, like a skip, is always executable.

Assertions

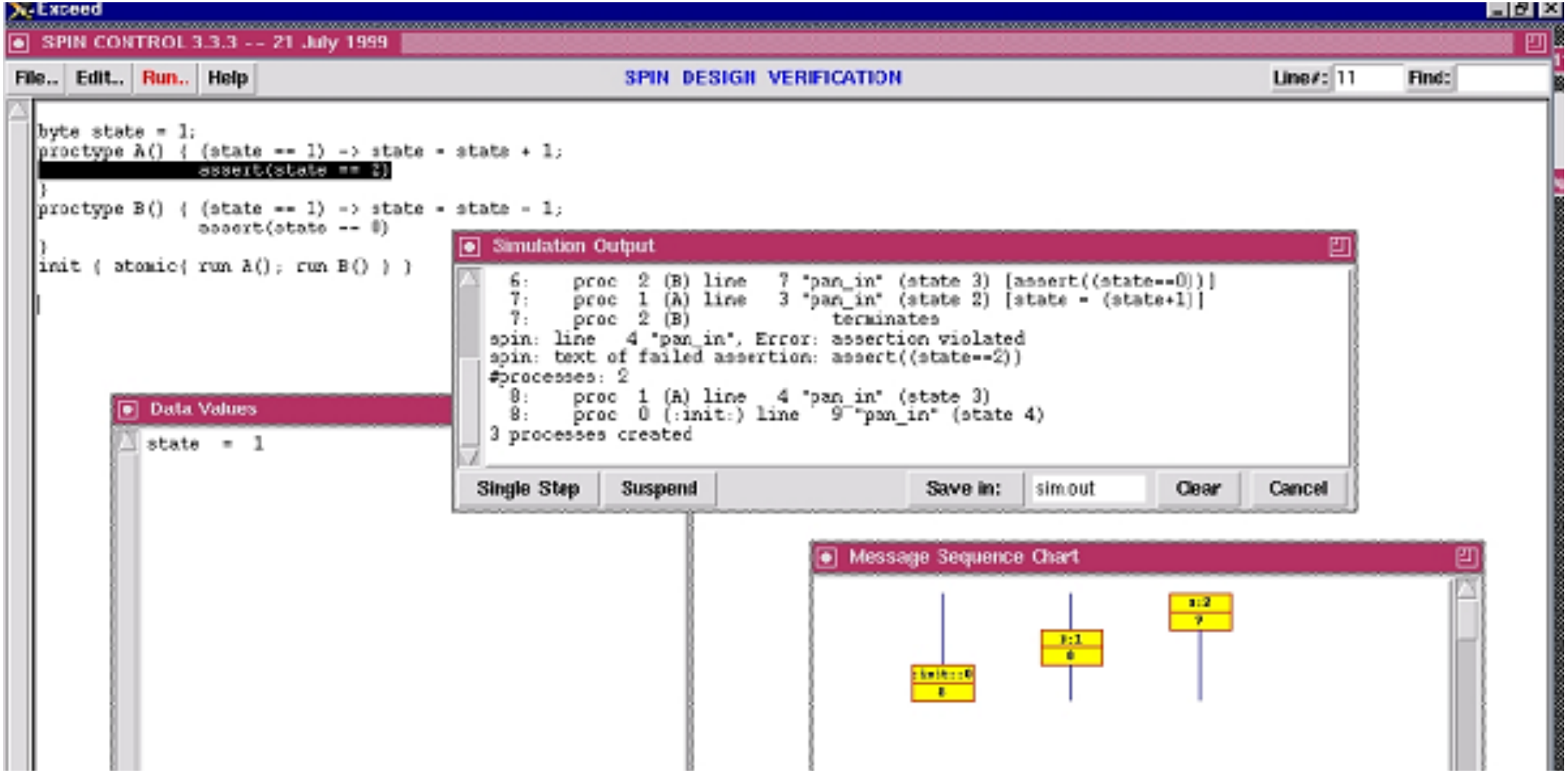
- An assertion is a statement which can be either true or false.
- Interleaving assertion evaluation with code execution provides
  - a simple yet very useful mechanism for checking desirable as well as erroneous behaviour with respect to our models.
- Assertion: Syntax within Promela
  - `assert( <logical-statement> )`
  - `e.g. assert(!(doors == open && lift == moving) )`
- Within Promela we can express local assertions as well global system assertions.

Assertions: Example of Local Assertion

```
byte state = 1;
proctype A() {
  (state == 1) -> state = state + 1;
  assert(state == 2)
}
proctype B() {
  (state == 1) -> state = state - 1;
  assert(state == 0)
}
init { atomic{ run A(); run B() } }
```

Will the assertion checking succeed here or fail?

Assertions…


Exceed
SPIN CONTROL 3.3.3 -- 21 July 1999
File.. Edit.. Run.. Help
SPIN DESIGN VERIFICATION
Line#: 11 Find:
byte state = 1;
proctype A() { (state == 1) -> state = state + 1;
assert(state == 2)
}
proctype B() { (state == 1) -> state = state - 1;
assert(state == 0)
}
init { atomic{ run A(); run B() } }
Simulation Output
6: proc 2 (B) line 7 "pan_in" (state 3) [assert((state==0))]
7: proc 1 (A) line 3 "pan_in" (state 2) [state = (state+1)]
7: proc 2 (B) terminates
spin: line 4 "pan_in", Error: assertion violated
spin: text of failed assertion: assert((state==2))
#processes: 2
8: proc 1 (A) line 4 "pan_in" (state 3)
8: proc 0 (:init:) line 9 "pan_in" (state 4)
3 processes created
Single Step
Suspend
Save in:
simout
Clear
Cancel
Data Values
state = 1
Message Sequence Chart

Assertions Global Assertions

- A global assertion is also known as a system invariant
  - is a property that is true in the initial system state and remains true in all possible execution paths.
- To express a system invariant within Promela
  - one must define a monitor process that contains the desired system invariant.
- To ensure that the global assertion is checked anypoint during the execution….
  - an instance of the monitor process has to be run along with the rest of the system model
- In the case of a simulation the checking is not exhaustive, this is achieved within verification mode.

Mutual Exclusion#1

models: `while (flag == 1) /*wait*/`

```
bit flag;
byte mutex;
proctype P(bit i) {
  flag != 1;
  flag = 1;
   mutex++;
   printf("MSC: P(%d) has entered the critical section\n, i);
   mutex--;
   flag = 0;
}
proctype monitor() {
   assert(mutex !=2);
}
init {          {run P(0); run P(1); run monitor(); } }
```

- What could be the eventual value of mutex ?
- Is it really achieved ?
- Is the assertion preserved or violated, here ?

Mutual Exclusion#1

```
bit flag;
byte mutex;
proctype P(bit i) {
   flag != 1;
   flag = 1;
   mutex++;
   printf("MSC: P(%d) has entered the critical section\n, i);
   mutex--;
   flag = 0;
}
proctype monitor() {
   assert(mutex !=2);
}
init { atomic {run P(0); run P(1); run monitor(); } }
```

- What could be the eventual value of mutex ?
- Is it really achieved ?
- Is the assertion preserved or violated, here ?

Mutual Exclusion#2

```
bit flag1, flag2;
byte mutex;
```

```
active proctype A() {
   flag1 = 1;
   flag2 == 0;
   mutex++;
   mutex--;
   flag1 = 0;
}
```

```
active proctype B() {
   flag2 = 1;
   flag1 == 0;
   mutex++;
   mutex--;
   flag2 = 0;
}
```

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

- What could be the eventual value of mutex ?
- Is it really achieved ?
- Is the assertion preserved or violated, here ?

"Invalid End state" in SPIN

Mutual Exclusion#3 - Dekker's algorithm

```
bit flag1, flag2; byte mutex, turn;
```

```
active proctype A() {
   flag1 = 1;
   turn = B_TURN;
   flag2 == 0 || turn == A_TURN;
   mutex++;
   mutex--;
   flag1 = 0;
}
```

```
active proctype B() {
   flag2 = 1;
   turn = A_TURN;
   flag1 == 0 || turn == B_TURN;
   mutex++;
   mutex--;
   flag2 = 0;
}
```

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

First software-only solution to the mutex problem for two processes…

Dekker's algorithm

```
#define true 1 false 0 Aturn 1 Bturn 0
bool ARuns, BRuns, t;
```

```
proctype A()
{
   ARuns = true;
   t = Bturn;
   (BRuns==false || t==Aturn);
   /*critical section */
   ARuns = false
}
```

```
proctype B()
{
   BRuns = true;
   t = Aturn;
   (ARuns==false || t==Bturn);
   /*critical section */
   BRuns = false
}
```

```
init { run A(); run B();}
```

Can this be genralized to a single process?

Identify the output

```
int a, b, quo, rem;
bit done, load;

proctype quo_rem()
{
    do
    :: (load == 1) -> quo = 0; rem = a; done = 0;
    :: (load != 1) -> if
                      :: (rem >= b) -> rem = rem-b; quo = quo+1
                      :: (b > rem)  -> done = 1
                      fi
    od
}
init { a = 7; b = 2; done = 1; load = 1;
       run quo_rem();
       done == 0; load = 0; done == 1;
       printf("result1  = %d\n", quo);
       printf("result2 = %d\n", rem)
}
```

Identify the difference and the output

```
byte in1, in2,  a, b, quo, rem;
bit  load = 0, done = 1;

proctype quo_rem() {
    do
    :: (load == 1) -> a = in1; b = in2;
                quo = 0; rem = a; done = 0;
    :: (load != 1) -> if
                :: (rem >= b) -> rem = rem-b; quo = quo+1
                :: (b > rem)  -> done = 1
            fi
    od
}
proctype env() {
   in1 = 7;  in2 = 2; load = 1;       /* init inputs */
    done == 0; load = 0; done == 1;    /* read results  */
    printf("quotient  = %d\n", quo);
    printf("remainder = %d\n", rem) }

init { atomic{ run quo_rem(); run env() }}
```

Timeouts

- Reactive systems typically require a means of aborting OR rebooting when a system deadlocks.
- Promela provides a primitive statement called timeout for the purpose e.g.

```
proctype watchdog ()
{ do
  :: timeout -> guard!reset
  od
}
```

- The timeout condition becomes true when no other statements within the overall system being modelled are executable.

Exceptions

- the **unless** statement
  - a useful exception handling feature

```
{ statements-1 }
unless
{ statements-2 }
```

- Consider an alternative watchdog process

```
proctype watchdog ()
{
do
:: process_data() unless guard?reset;
:: process_reset()
od
}
```

Message Channels

- What means have been studied so far to achieve communication between distinct processes ?
- However, Promela supports message channels also
  - provide a more natural and sophisticated means of modeling inter-process communication / data transfer.

Message Channels….

- a channel declaration could take any of the forms….

```
chan <name> =<dim> of {<t1>,<t2>,<t3> ...<tn>}
```

- Semantics

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

- e.g.

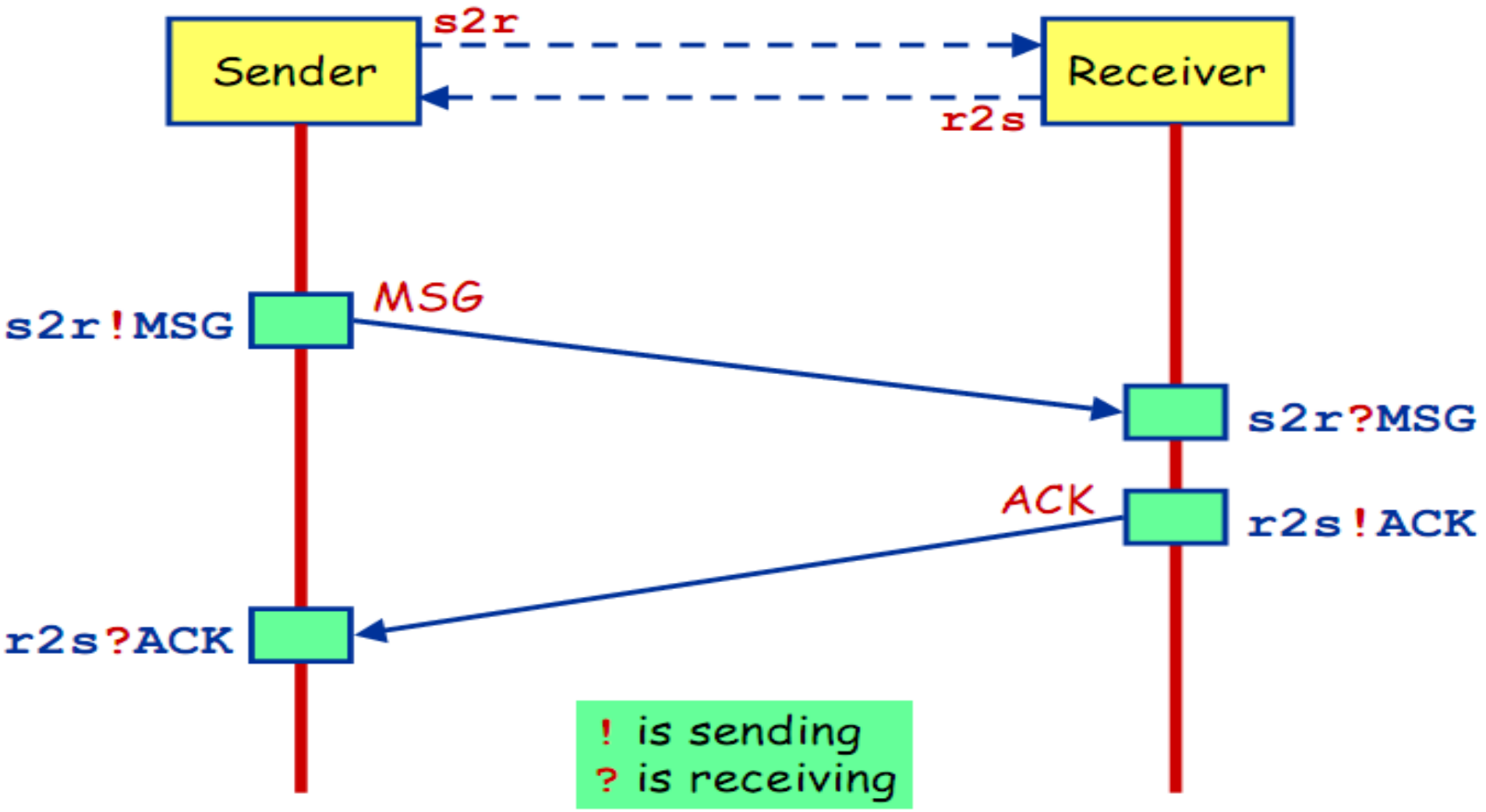
```
chan q = [1] of {byte}
chan q = [3] of {mtype, int}
```

- if dim=0 then synchronous

```
chan q = [0] of {bit}
```

- A channel can be defined to be either local or global.

Message Channels….



Message Channels….Sending-in

- Sending messages through a channel - FIFO buffer `(dim >0)`
  - achieved by the ! operator,
  - e.g. `in_data ! 4;`
  - type of the channel and variable must match
- For multiple data values are to be transferred via each message
  - `out_data ! x + 1, true, in_data;`

    where x is of type byte.
- The executability of a send statement is dependent upon the associated channel being non-full.

Message Channels….Receiving from

- Receiving messages is achieved by the ? operator

  e.g. **`in_data ? msg;`**
- if the channel is not empty, the first message is fetched from the channel and the is stored **`in_data`**
- Multiple values can also be fetched

  e.g. **`out_data ? value1, value2, value3;`**
- the executability of a receive statement is dependent up on the associated channel being non-empty,
  - e.g. the following statement will be blocke d

    `in_data ? value;`

    unle ss in_data contains at least one message .

An illustration : Message Channels….

```
#define N 128
#define SIZE 16
chan in = [size] of {short};
chan large = [size] of {short};
chan small = [size] of {short};
active proctype split(){
  short cargo;
  do
  ::in ? cargo →
      if
      :: (cargo >= N) → large!cargo
      :: (cargo < N) → small!cargo
      fi
  od
}
```

Message Channels….

- If more data values are sent per message than can be stored by a channel then the extra data values are lost

  e.g. `in_data ! msg1, msg2;`

  the msg2 will be lost.

- If fewer data values are sent per message than are expected then the missing data values are undefined,

  e.g. `out_data ! 4, true;`

  ```
  out_data ? x, y, z;
  ```

  x and y will be assigned the values 4 and true respectively while the value of z will be undefined.

Message Channels….

- **`len`** operator
  - to determining the number of messages in a channel

    `e.g. len(in_data)`
  - if the channel is empty then the statement will block.
- **`empty, full`** operators
  - determine whether or not messages can be received or sent respectively,

    e.g. **`empty(in_data); full(in_data)`**
- Non-destructive retrieve

  `out_data ? [x, y, z]`

  returns 1 if **`out_data?x,y,z`** is executable otherwise 0.
  - No side-effects – evaluation, not execution, i.e. no message retrieved.

Channels as parameters

```
proctype A(chan q1) {
   chan q2;
   q1?q2;
   q2!123
}
proctype B(chan qforb) {
   int x;
   qforb?x;
   printf("x = %d\n", x)
}
init {
   chan qname[2] = [1] of { chan };
   chan qforb = [1] of { int };
   run A(qname[0]); run B(qforb);
   qname[0]!qforb
}
```

What will be the output ?

Channels as parameters….

```
proctype A(chan q1){
   chan q2;
   q1?q2;
   q2!99
}
proctype B(chan qforb) {
   int x;
   qforb?x;
   x++;
   printf("x == %d\n", x)
}
init {
   chan qname = [1] of { chan };
   chan qforb = [1] of { int };
   run A(qname); run B(qforb); qname!qforb
}
```

What will be the side-effect of running this program?

Channels as parameters :Division

```
proctype division(int x,y,q; chan res) {
  if
  :: (y > x) -> res!q,x;
  :: (x >= y) -> run division(x - y, y, q + 1, res);
  fi
}
init{
  int q,r;
  chan child = [1] of { int, int };
  run division(7, 3, 0, child);
  child ? q,r;
  printf("result: %d %d\n", q,r)
}
```

Note that this is a tail-recursive program

process factorial

```
proctype fact(int n, chan p)
{ int result;
  if
  ::  (n <=1) → p ! 1
  ::  (n >=2 →
      chan child = [1] of {int};
      run fact(n-1, child);
      child ? result;
      p ! n *result
  fi
}
```

```
init {
   int result;
   chan child = [1] of {int};
   run fact(7,child);
   child ? result;
   printf("result:%d\n", result)
}
```

Reason why is it NOT a tail-recursive program

Communication type

- What was the type of communication pattern observed in the examples till now ? i.e.
  - Synchronous OR Asynchronous communication ?
  - Why ?

Synchronous communication

- when the channel declaration is

  `chan ch = [0] of {bit, byte};` i.e.when **`dim=0`**

- if **`ch!x`** is enabled and
  - if there is a corresponding receive **`ch?x`**
  - that can be executed simultaneously and both the statements are enabled
  - both statements will handshake and together do the transition

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

P wants to do **`ch!1,3+7`**

Q wants to do **`ch?1,x`**

then after communication x will be 10

Synchronous communication: An illustration

```
#define msgtype 33
chan name = [0] of { byte, byte }
proctype A() {
  name!msgtype(124);
  name!msgtype(121)
}
proctype B() {
  byte state;
  name?msgtype(state)
}
init { atomic { run A(); run B() }}
```

Rendezvous communication

```
#define msgtype 13
byte name;
chan name = [0] of {byte, byte}
proctype A()
{
  name ! msgtype(124);
  name ! msgtype(121)
}`
proctype B()
{
  byte state;
  name ? msgtype(state)
}
init
{ atomic {run A(); run B()}}
```

What are the implications of atomic?

What if the channel sizes were two? one?

Dijkstra's semaphores

```
#define p 0 v 1
chan sema = [0] of {bit};
proctype dijkstra (){
do
:: sema ! p → sema ? v
od
}
proctype user() {
   sema ? p;
       /*critical section */
   sema!v
}
init{
   atomic { run dijkstra(); run user(); run user();
   run user(); }
}
```

Semaphores Revisited : What is the difference?

```
#define p 0 #define v 1
chan sema = [0] of { bit };

proctype semaphore() {
   do :: sema!p -> sema?v od
}
byte count;
proctype user() {
   sema?p;
   count = count + 1;
   skip; /* critical section */
   count = count - 1;
   sema!v; skip /* non-critical section */ }

proctype monitor(){
   do :: assert(count == 0 || count == 1) od
}
init { atomic {run monitor(); run semaphore(); run user(); run
   user(); run user()}}
```

What does this protocol do ?

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

Alternating Bit Protocol

- To every message, the sender adds a bit.
- The receiver acknowledges each message by sending the received bit back.
- The receiver only excepts messages with a bit that it excepted to receive.
- If the sender is sure that the receiver has correctly received the previous message, it sends a new message and it alternates the accompanying bit.

Alternating Bit protocol

```
mtype = {msg, ack};
chan to_sndr = [2] of {mtype, bit};
cha to_rcvr = [2] of {mtype, bit};
proctype Sender(chan in, chan out)
{
   bit sendval, recval;
   do
   :: out!msg(sendval) ->
        in?ack(recval);
        if
        :: recval == sendval ->
               sendval = 1 - recval
        :: else -> skip
        fi
   od
}
```

```
proctype Receiver(chan in, chan
   out)
{
   bit recval;
   do
   :: in!msg(recval) ->
        out?ack(recval);
   :: timeout ->
         out?ack(recval);
   od
}


init {
   run Sender(to_sndr, to_rcvr);
   run Receiver(to_rcvr, to_sndr);
}
```

Modeling Loss of Messages

Reviewing Interleaving Semantics…

- Transition has two components
  - Side-effect free condition and an atomic action
- A transition is executable if its condition holds, otherwise it is blocked
  - Assignments are always executable
  - Run statements are executable is new processes can be created
  - Conditions are executable if the truth value is true
  - Send statements are executable if channel not full (or ...)
  - Receive statements are executable if channel is nonempty and patterns match
  - Skip statements are always executable

Traffic Lights

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

An illustration

```
#define train 1

chan BlockSecAB = [2] of { bit };
chan BlockSecBC = [2] of { bit };

proctype SignalA(chan out_track)
{
        do
        :: out_track!1
        od
}
proctype SignalB(chan in_track,
                 out_track)
{
        do
        :: in_track?train;
           out_track!train
        od
}
```

```
proctype SignalC(chan in_track)
{
        do
        :: in_track?train;
        od
}
init {  atomic{
        run SignalA(BlockSecAB);
        run SignalB(BlockSecAB,
                    BlockSecBC);
        run SignalC(BlockSecBC)
   }
}
`
```

An illustration modified

```
chan TunnelAB = [2] of { byte };
chan TunnelBC = [2] of { byte };
chan TunnelCD = [2] of { byte };
chan TunnelDA = [2] of { byte };

proctype Station(chan in_track, out_track)
{
        byte train;

  do
  :: in_track?train; out_track!train
  od
}
```

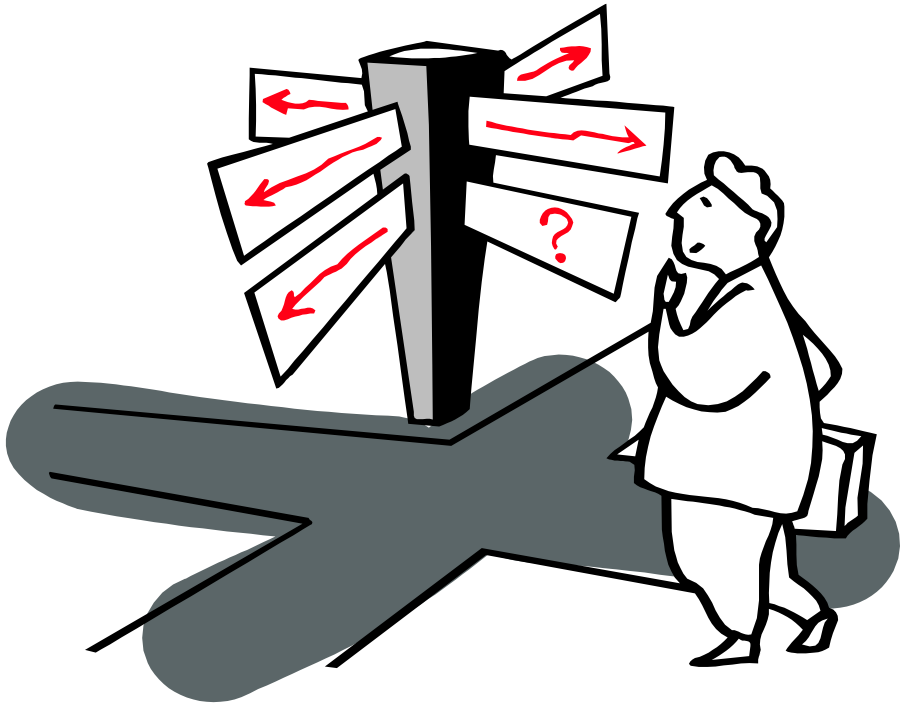
An illustration modified…

```
proctype Setup(chan track; byte train)
{
   track!train;
}

init {  atomic{
 run Setup(TunnelBC, 1); /* introduce train 1 before station C */
 run Setup(TunnelDA, 2); /* introduce train 2 before station A */

 run Station(TunnelDA, TunnelAB);  /* station A */
 run Station(TunnelAB, TunnelBC);  /* station B */
 run Station(TunnelBC, TunnelCD);  /* station C */
 run Station(TunnelCD, TunnelDA)}  /* station D */
}
```

To teach is to learn twice !!



Hence, Thank You !!!