Model Checking Using SPIN/PROMELA

Formal Verification :

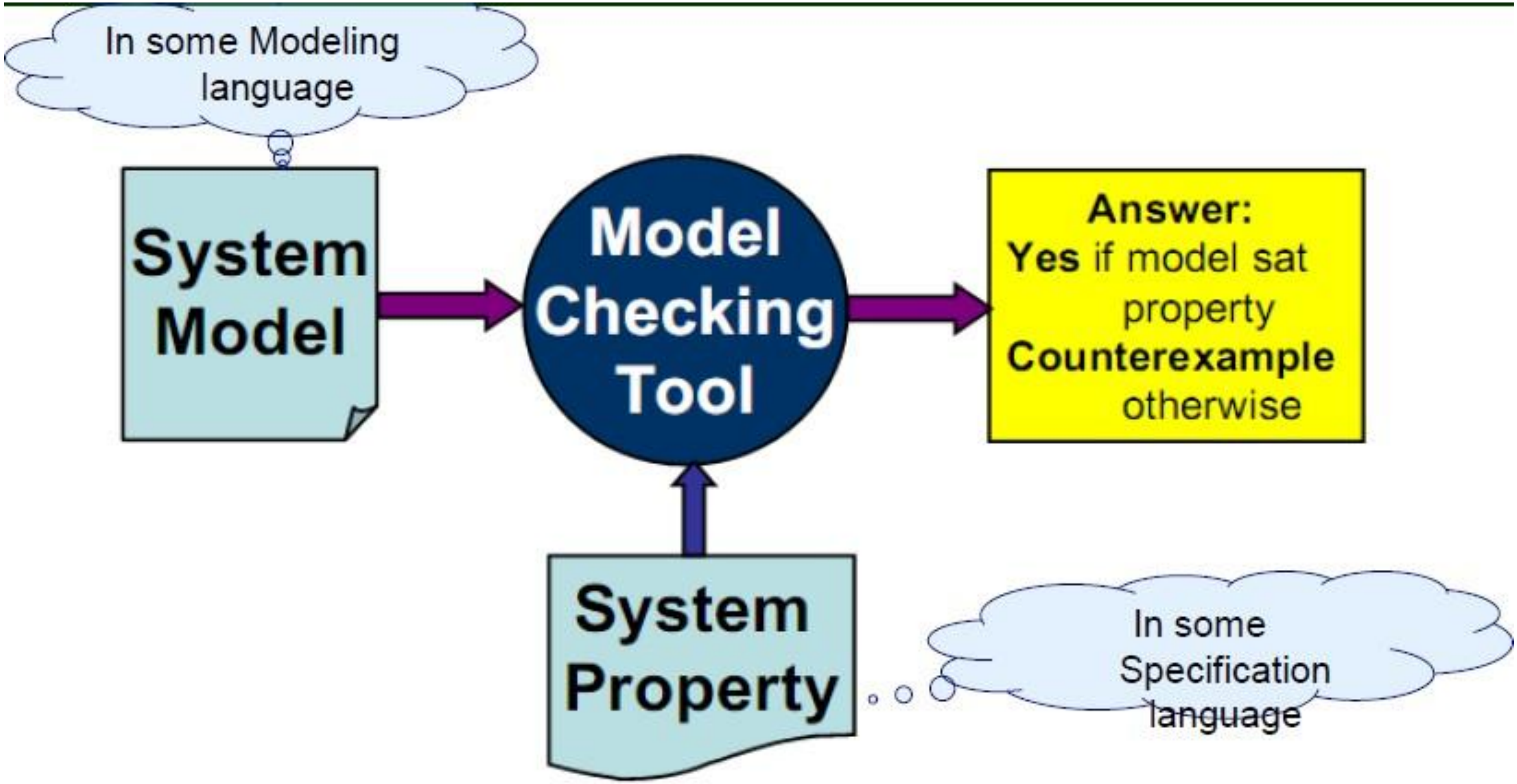
- The process of Formal Verification consists of
  - Requirements
  - Capture Modeling
  - Specification
  - Analysis
  - Documentation
- In practice, some phases may overlap
  - The overall process is iterative rather than sequential

Model Checking

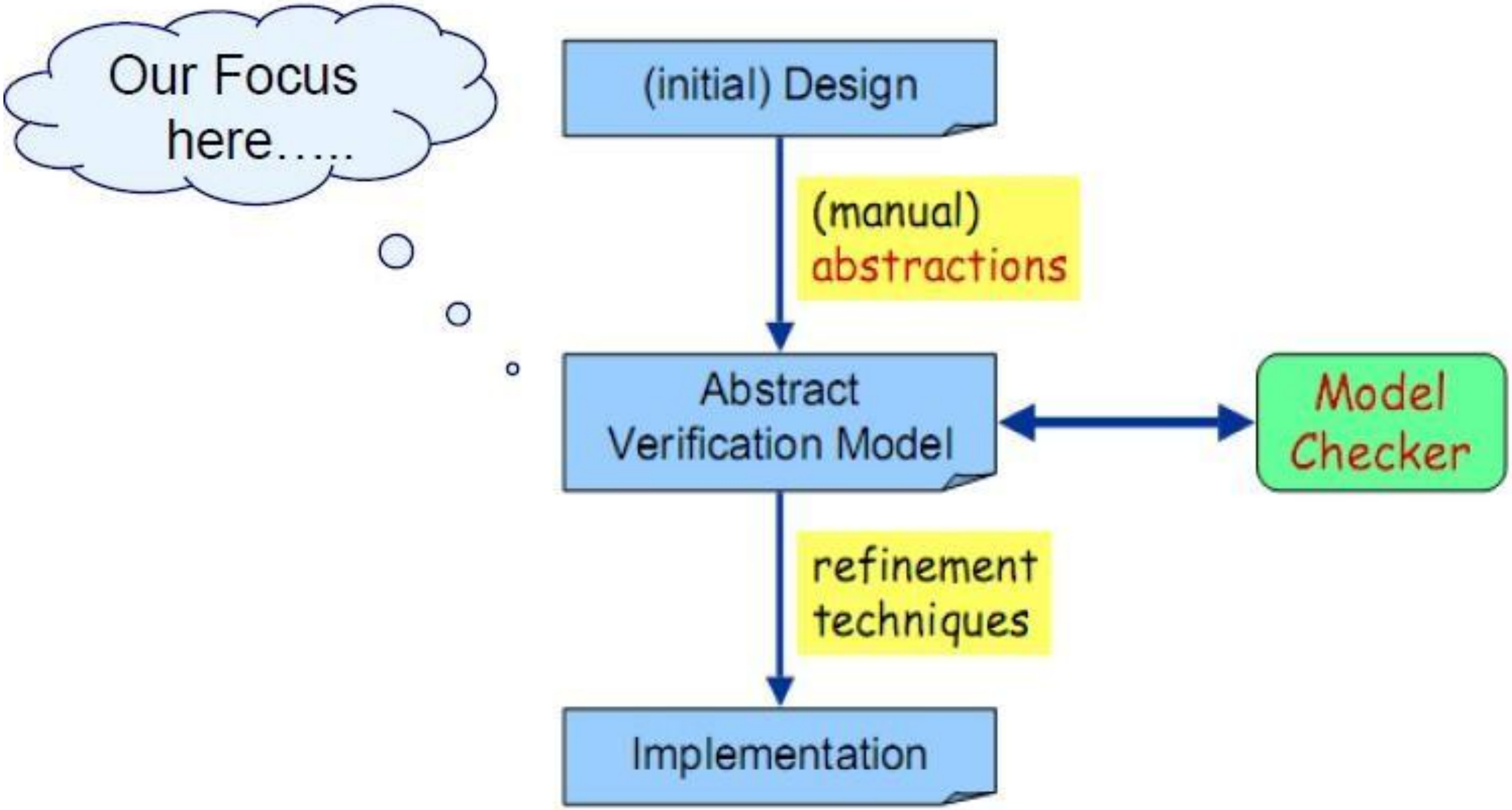
- Model checking
  - Is one of the powerful FORMAL VERIFICATION technique
  - Allows one to verify temporal properties of a finite state representation
  - The finite state representation is that of a typical concurrent system
  - The representation is a model of the system
- The basic idea is
  - That 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.

Model Checking

- State space explosion
  - Because a complete state space is generated for a given model, the analysis may fail due to lack of memory, if the model is too large.
  - Can be tackled via abstraction
- Verifying design model
  - Against specification for finite state concurrent systems
- It is an automated technique wherein
  - Inputs
    - Finite state model of the system and properties stated in some standard formalism
  - Outputs
    - Property valid against the model or not



The entire process is automated


Our Focus here.....
(initial) Design
(manual) abstractions
Abstract Verification Model
Model Checker
refinement techniques
Implementation

Applicability: Distributed

- 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
- Underspecification
  - Unexpected reception of messages
- Overspecification
  - Dead code
- Violations of constraints
  - Buffer overruns
  - Array bound violations

Model Checking Definition

- "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"

$$M, s \models p$$

  - Does system model **M** with initial state **s** satisfy system property **p**
  - **M** given as a state machine, that is finite-state
  - **p** is 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

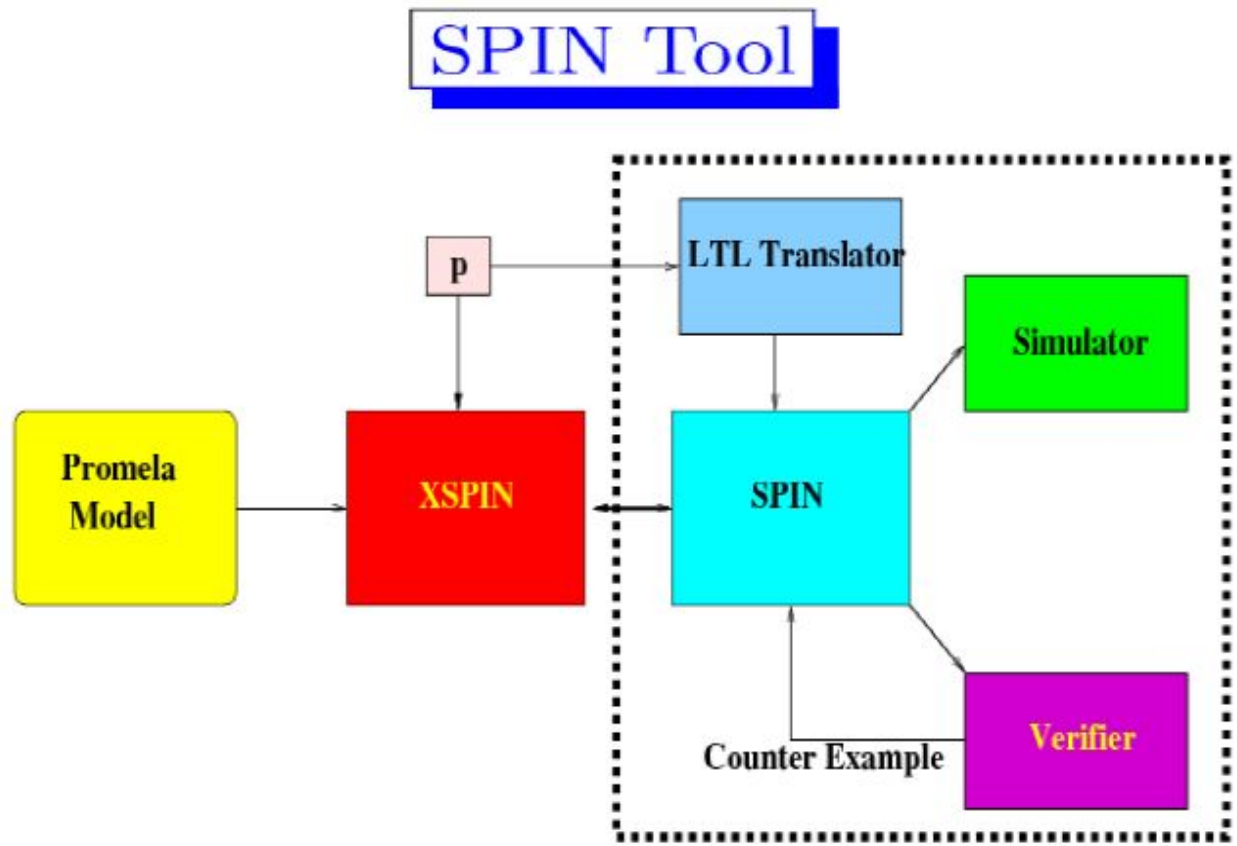
Introduction to SPIN

- SPIN – Simple Promela INterpreter
  - A tool for analyzing the logical consistency of concurrent systems, specifically of data communication protocols
  - System model of a concurrent system is described in PROMELA
- PROMELA – PRocess MEta Language
  - Specification language to model finite-state systems & allows dynamic creation of concurrent processes
  - Expressive enough to describe processes and their interactions in Synchronous as well as Asynchronous manner
  - Resembles the programming language C

Introduction to

- PROMELA and SPIN/XSPIN are
  - Developed by Gerard Holzmann at Bell
  - Labs Freeware for non-commercial use
  - Is a State-of-art model checker

Introduction to SPIN



Introduction to SPIN

- 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
  - “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 alogorithms)

Introduction to SPIN

- SPIN's starting page:
  - http://www.spinroot.com
  - Basic SPIN manual
  - Getting started with
  - Xspin Getting started
  - with SPIN Examples and
  - Exercises
  - Concise Promela Reference
- Proceedings of all SPIN
  - workshops

Gerard Holzmann's website for papers on SPIN

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 Distributed System NOT the implementation details

PROMELA

- PROMELA model consists of
  - Process declarations
  - Channel
  - declarations
  - Variable
  - declarations Type
  - 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

PROMELA

- A process type (proctype) consists of
  - A name, list of formal parameters, declarations of local variables and 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

PROMELA Variables and Basic Data T

- 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
  - 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	$-2^{31} - 1 \ldots 2^{31} - 1$

PROMELA Variables and Basic Data T

- PROMELA variables
  - Variables must be declared before they can be used
  - Variable declarations follow the style of the C programming language
    - 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
    - The execution of a PROMELA program begins with the init process
  - An init process
    - Takes the form:
      - init {/* local declarations and statements */ }
      - init {skip}
  - While a proctype definition declares the behavior of a process, the instantiation and execution of a process definition is coordinated via the init process.

Statement

- 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
  - E.g. x = 2;

Statement

- An expression is also a statement;
  - It is executable if it evaluates to non-zero
    - skip, 2 < 3 are always executable
    - X < 27 is executable only if the value of x is less than 27
  - A run statement is executable
    - Only if the process can be created
    - Returns 0 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.

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

Executability of

- 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 or it 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(); }
- init is the starting point
- run operator is executable only if process instantiation is possible
- If a run is executable than 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

- 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”)}

```
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

Other Data T

- Arrays
  - An array type is declared as int table[max]
  - This generates an array of integers i.e. table[0], table[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 record data type is declared as

    typedf 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
*/

```
/* 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);
}
```

- How many processes are created, here?

Statement

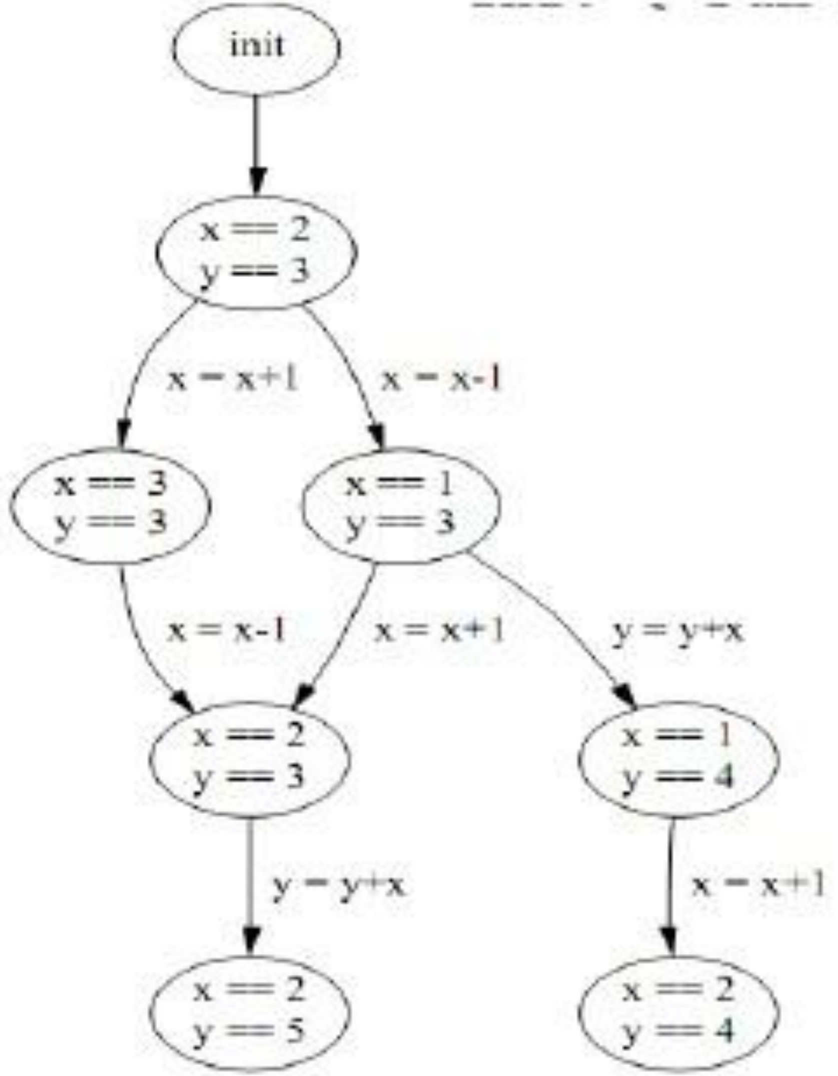
- Two types of statement delimiters
  - ; and ->
  - Use the one that is most appropriate at the given
  - situation Usually ; is used between ordinary statements
  - -> is often used after “guards” in a if OR do statement, pointing at what comes next
  - These can be used interchangeably.

Interleaving

- 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

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

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



Deterministic V/s Nondeterministic Behavior

- 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..

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 ?

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…

PROMELA Control Structures

- Three ways for achieving control flows
  - 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 example of case selection with

```
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 choose 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

- 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 once, 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 these system, in particular, when dealing with reactive systems

```
do
:: (level > max) -> outlet = open;
:: (level < min) -> outlet = close;
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	$-2^{31} - 1 \ldots 2^{31} - 1$

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

```
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 appear after 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 behavior 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 as global system 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

```
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

```
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…

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.

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

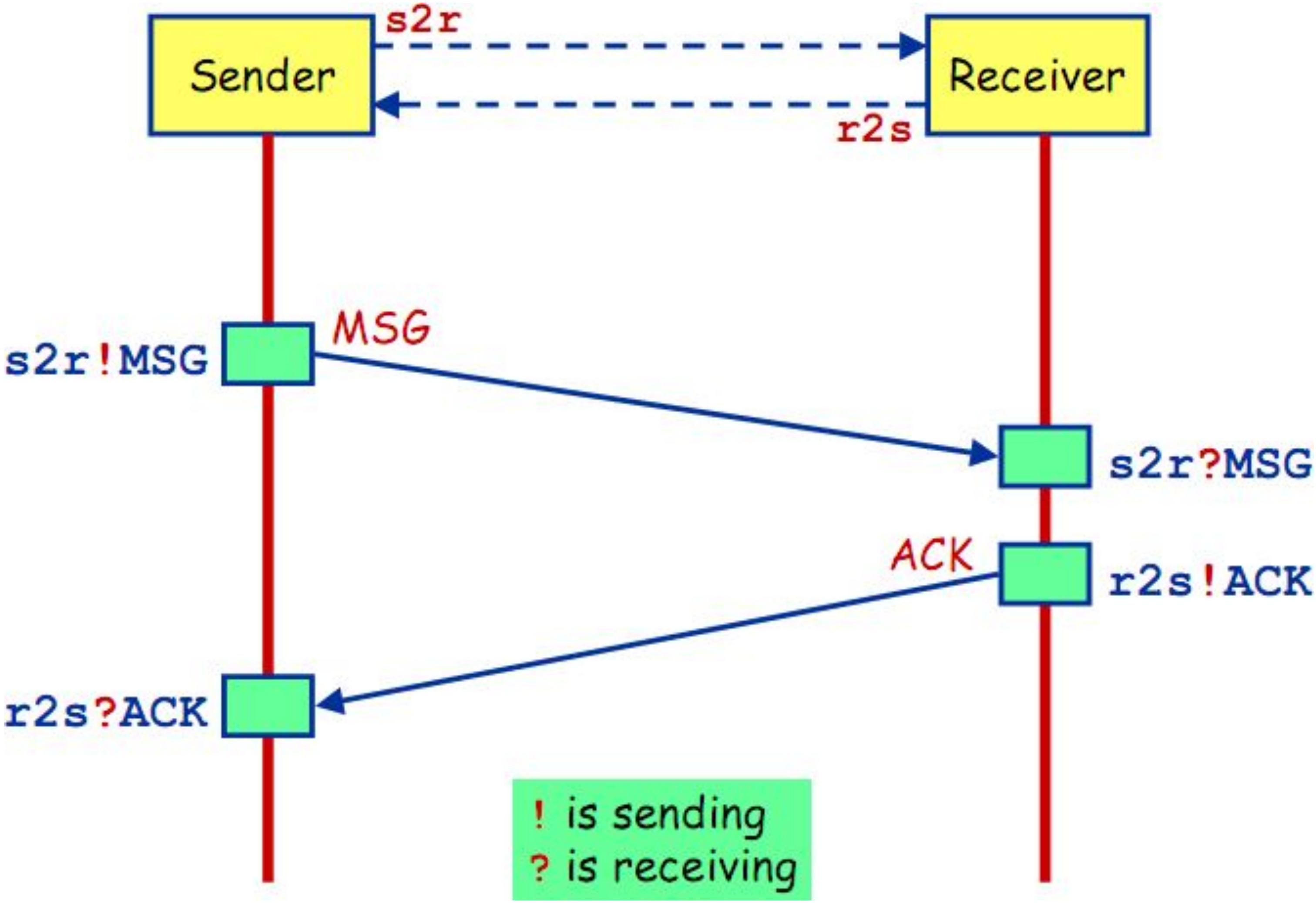
- The timeout condition becomes true when no other statements within the overall system being modeled are executable

Exceptions

- The unless statement
- A useful exception handling feature
- {statements-1} unless {statements-2}
- Consider an alternate watchdog process.

Message Channels

- Means to achieve communication between distinct processes?
- PROMELA supports message channels:
  - Provide a more natural and sophisticated means of modelling inter-process communication/data transfer
- Channel declaration:
  - chan <name> = <dim> of {<t1>,<t2>,<t3> … <tn>}
- E.g. chan q = [1] of {byte} ; chan q = [3] of {mtype, int}
- If dim = 0 then synchronous. E.g. chan q = [0] of {bit}
- A channel can be defined to be either local or global



Sender
s2r
Receiver
r2s
s2r!MSG
MSG
s2r?MSG
ACK
r2s!ACK
r2s?ACK
! is sending
? is receiving

Message Channels… Sending-in

- Sending messages through a channel – FIFO buffer (dim>0)
  - Achieved by ! Operator
  - E.g. in_data ! 4
  - Type of the channel and variable must match.
- If multiple data values are to be transferred via each message
  - out_data ! x+1, true, in_data
- The executability of a send statement is dependent upon the associated channel being non-full

Message Channels… Receiving from

- Receiving messages is achieved by ? Operator
  - E.g. in_data ? Msg
- If the channel is not empty, the first message is fetched from the channel and is stored in msg
- Multiple values can also be fetched.
  - E.g. out_data ? value1, value2, value 3;
- The executability of a receive statement is dependent upon the associated channel being non-empty,

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 ; 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 and 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 determine 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. Only evaluation, not execution. No message retrieved.

Channels as parameters

- Output?

```
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
}
```

Communication type

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

  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 perform ch ! 1, 3 +
- 7 Q wants to perform ch ? 1, x
- Then after communication x will be 10.

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 expects messages with a bit that is expected 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