Credits:

- Many slides (all the figures with blue background and few others) were taken from Holzmann’s slides on “Logical Model Checking”, course given at Caltech
Outline

1 Overview of Promela
   - Rules for Executability
   - Control Flow

2 Promela Semantics
   - Motivation
   - Operational Model
   - Interpreting Promela Models
   - Something on Verification
On Promela Expressions

- Depending on the system state each statement in a Spin model can be
  - Executable
  - Blocked

- Any expression in Promela can be used as statements in any context

- Expressions are executable iff they evaluate to true or to a non-zero integer value

- For instance, instead of writing a waiting loop like:
  ```
  while (a != b)
  skip /* do nothing and wait till a==b */
  ```
  it is possible to write in Promela
  ```
  (a==b);
  ```
On Promela Expressions

- Expressions must be side-effect free
  - Reason: a blocking expression may be evaluated many times
  - Exception to this rule: expressions containing the `run` operator can have side effects
    - Syntactic restriction: There can be only one `run` operator in an expression and it cannot be combined with any other operator
  - `run` must be used carefully – Consider the following model:

```c
active proctype splurge(int n)
{
  printf("%d\n", n);
  run splurge(n+1)
}
```

- After the 255th attempt to instantiate a new process, Promela will fail
- The `run` expression will evaluate to zero and it will permanently block the process
- Since the process will not reach the end of its code, then will not terminate (nor die) and none of its predecessor could die either
Rules for Executability
6 Types of Basic Statements

- Always executable:
  - **assignment:**  \( x++ \),  \( x-- \),  \( x = x+1 \),  \( x = \text{run } P() \)
  - \( b = c++ \) is not a valid expression (right-hand side is not side-effect free)
  - **print:**  \texttt{printf(''x = %d\n'', x)}
  - **assertion:**  \texttt{assert(1+1==2)}

- Executable when *true* (non-zero):
  - **expression statement:**
    \((x), (1), \text{run } P(), \text{skip}, \text{true}, \text{else}, \text{timeout} \)

- Executable when target channel is non-full
  - **send:**  \( q!\text{ack}(m) \)

- Executable when target channel is non-empty, and constraints are met
  - **receive:**  \( q?\text{ack}(n) \)
Defining the Control Flow

- Basic statements (so far)
  - print, assignment, assertions, expressions, send and receive
  - Notice that run is not a statement but an operator and skip is an expression (equivalent to \((1)\) or true)

- Five ways to define control flow
  - Semi-colons, gotos and labels
  - Structuring aids
    - inlines
    - macros
  - Atomic sequences (indivisible sequences)
    - atomic {...}
    - d_step {...}
  - Non-deterministic selection and iteration
    - if ... fi
    - do ... od
  - Escape sequences (for error handling/interruptions)
    - {...} unless {...}
The (non-deterministic) `if` statement is inspired on Dijkstra’s guarded command language.

```plaintext
/* find the max of x and y */
if
  :: x >= y -> m = x
  :: x <= y -> m = y
fi

/* pick a number 0..3 */
if
  :: n=0
  :: n=1
  :: n=2
  :: n=3
fi
```

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

```plaintext
/* find the max of x and y */
if
  :: (n % 2 != 0) -> n = 1
  :: (n >= 0) -> n = n-2
  :: (n % 3 == 0) -> n = 3
  :: else /* -> skip */
fi
```

Underlying non-deterministic automaton

Gerardo Schneider (Ifi, UiO)
else is a predefined variable

where in C one writes:

```c
if (x <= y)
    x = y-x;
    y++;
```

i.e., omitting the ‘else’

in Promela this is written:

```promela
if:: (x <= y) -> x = y-x:: else
fi;
y++
```

i.e., the ‘else’ part cannot be omitted.

in this case ‘else’ evaluates to:

```promela
!(x <= y)
```

the else clause always has to be explicitly present without it, the if- statement would block until (x<=y) becomes true (it then gives only one option for behavior).
timeout is also a predefined variable

```promela
if
:: q?msg -> ...
:: q?ack -> ...
:: q?err -> ...
:: timeout -> ...
fi
```

wait until an expected message arrives, or recover when the system as a whole gets stuck (e.g., due to message loss)

checking for bad timeouts:
spin –Dtimeout=true model

note carefully that using ‘else’ instead of ‘timeout’ is dubious in this context
Selection

- **else** and **timeout** are related
  - Both are predefined Boolean variables
  - Their values are set to *true* or *false* by the system, depending on the context
- They are, however, not interchangeable
  - **else** is true iff no other statement in the same *process* is executable
  - **timeout** is true iff no other statement in the same *system* is executable
- A **timeout** may be seen as a system level **else**
- Are these equivalent?

  ```
  if :: q?msg -> ... :: q?msg -> ...
  :: q?ack -> ... :: q?ack -> ...
  :: timeout -> ... :: else -> ...
  fi
  fi
  ```

- **No!** In the second, if a message is not received when the control is at the **if** then the **else** is taken immediately
The do statement is an if statement caught in a cycle

\[
\begin{align*}
\text{do} \\
:: \text{guard1} &\rightarrow \text{stmt1.1}; \text{stmt1.2}; \text{stmt1.3}; \ldots \\
:: \text{guard2} &\rightarrow \text{stmt2.1}; \text{stmt2.2}; \text{stmt2.3}; \ldots \\
:\vdots \\
:: \text{guardn} &\rightarrow \text{stmtn.1}; \text{stmtn.2}; \text{stmtn.3}; \ldots \\
\text{od}
\end{align*}
\]

- Only a break or a goto can exit from a do
- A break transfers control to the end of the loop
Repetition

There are many ways of writing a waiting loop, by exploiting the executability rules it’s possible to simplify the model.

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

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

The skip is not needed here and can introduce an unnecessary control state.

Note that 'break', like 'goto', is not a basic statement but a control-flow specifier.
Atomic Sequences

atomic { guard -> stmt1; stmt2; ... stmtn }

- executable if the guard statement is executable
- any statement can serve as the guard statement
- executes all statements in the sequence *without* interleaving with statements in other processes
- if any statement other than the guard blocks, atomicity is lost
  atomicity can be regained when the statement becomes executable
- example: mutual exclusion with an indivisible test&set:

```proctype P()
{ atomic { (busy == false) -> busy = true }
  mutex++;
  assert(mutex==1);
  mutex--;
  busy = false;
}
```
**Deterministic Steps**

*d_steps* are more restrictive and more efficient than atomic sequences

```
  d_step { guard -> stmt1; stmt2; ... stmtn }

  - like an atomic, but *must be deterministic* and *may not block* anywhere

  - especially useful to perform intermediate computations with a deterministic result, in a single indivisible step

```d_step { /* reset array elements to 0 */
i = 0;
do:: i < N -> x[i] = 0; i++:: else -> break
  od;
i = 0
}
```

- atomic and *d_step* sequences are often used as a model reduction method, to lower complexity of large models (improving tractability)
Atomic Sequences, Deterministic Steps and Gotos

- **goto-jumps into and out of atomic sequences are allowed**
  - atomicity is preserved only if the jump starts inside an atomic sequence and ends inside another atomic sequence, and the target statement is executable

- **goto-jumps into and out of d_step sequences are forbidden**

```plaintext
d_step {
    i = 0;
    do
    :: i < N -> x[i] = 0; i++
    :: else -> break
    od
};
x[0] = x[1] + x[2];
```

- This is a jump out of the d_step sequence and it will trigger an error from Spin.
- The problem is prevented in this case by adding a "; skip" after the od keyword - there's no runtime penalty for this, since it's inside the d_step.
Both sequences are executable only when the first (guard) statement is executable

- **atomic**: if any other statement blocks, atomicity is lost at that point; it can be regained once the statement becomes executable later
- **d_step**: it is an error if any statement other than the (first) guard statement blocks

Other differences:

- **d_step**: the entire sequence is executed as one single transition
- **atomic**: the sequence is executed step-by-step, but without interleaving, it can make non-deterministic choices

Remark:

- Infinite loops inside atomic or **d_step** sequences are not detected
- The execution of this type of sequence models an indivisible step, which means that it cannot be infinite
Deterministic Steps and Atomic Sequences

active proctype P1() { t1a; t1b }
active proctype P2() { t2a; t2b }

execution with full interleaving

execution without atomics or d_steps
Deterministic Steps and Atomic Sequences

active proctype P1() { atomic { t1a; t1b } }
active proctype P2() { t2a; t2b }

P1 could make alternate choices at the intermediate states (e.g., in if or do-statements)
P2 can be interrupted, but not P1
Deterministic Steps and Atomic Sequences

active proctype P1() { d_step {t1a; t1b} }
active proctype P2() { t2a; t2b }

no intermediate states are created: faster, smaller graph, but no non-determinism possible inside d_step sequence itself

P1 now has only one transition...
Escape Sequences

- **Syntax:** `{ P } unless { Q }
- Execution starts with the statements from P
- Before executing each statement in P, the executability of the first statement in Q is checked
- Execution of P statements continue only if the first instruction of Q is not executable
- As soon as the Q first statement can be executed, then control changes and execution continues in Q
- **Example**

```
A; { do
   :: b1 -> B1
   :: b1 -> B1
   ...
   od } unless { c -> C };
```

C acts here as a *watchdog*: as soon as it becomes true, C is executed and then D
Inline definitions

- somewhere in between a macro and a procedure
- used as replacement text with *textual* name substitution through parameters (it is a named piece of text with optional parameters)
- an inline is *not* a function – it cannot return values to the caller
- can help to structure a model
- compare:

```c
#define swap(a,b) tmp = a; \ 
   a = b; \ 
   b = tmp
```

```c
inline swap(a,b) {
   tmp = a;
   a = b;
   b = tmp
}
```

*hint:* when confused, use `spin -l spec.pml` to show the result of all inlining and macro preprocessing operations...

looks a little cleaner
line nr refs are better
Automata and “proctypes”

- Each Promela proctype defines a finite state automaton $(S, s_0, L, T, F)$
  - The set of states $S$ corresponds to the possible points of control within proctype
  - The transition relation $T$ defines the flow of control
  - The labels in $L$ relates each transition in $T$ with a basic statement that defines the executability and the effect of that transition
  - The set of final states $F$ is defined with Promela end-states, accept-states and progress-states

- There are only 6 basic statements: assignments, assertions, print, send and receive statements and expression statements

- Anything else (if, goto, do, break, unless, atomic, d_step) serves only to specify control flow and cannot appear as labels on transitions

- Every basic statement (including expression statements) has a precondition defining when it can be executable and its effects

$^1$More on such states on next lecture
Automata and “proctypes”

Example

active proctype not_euclid()
{
S: if
:: x == y ->
   assert(x != y);
   goto L
:: x > y ->
   L: x = x - y
:: x < y ->
   y = y - x
fi;
E: printf(“%d
”, x)
}
We have seen that each Promela process defines a finite state automaton \((S, s_0, L, T, F)\)

- The specification of a collection of asynchronous processes may be written in Promela as the asynchronous product of the automata.
- We can obtain a (possible huge) graph (automaton) containing all the reachable states where each edge represents a single possible execution step.
- The structure of such graph (automaton) is determined by the semantics of Promela.
  - Without such semantics we cannot know which execution paths are possible.
- We will see in a future lecture, how to specify properties in Promela which will express claims about presence or absence of subgraphs or paths in the reachability graph.
What are the possible executions?

There are two possible handshakes

- \( y!0 \) with \( y?0 \)
- \( x!0 \) with \( x?0 \)

Are both handshakes possible?

- We will find answers to these kind of questions later
The operational model is based on the specification of a semantic engine which determines how a Promela model defines system executions, including the rules that apply to interleaved execution of process actions.

The semantic engine operates on the global reachability graph. Such graph contains abstract objects corresponding to asynchronous processes, variables and message channels.

It only deals with local states and transitions. It does not know anything about control-flow construct (as if, do, break and goto).

The global reachability graph determines:

- A global system state (nodes), defined in terms of:
  - Variables, messages, message channels and processes
- A state transition (edges), defined in terms of:
  - The basic statements labeling the transitions
  - Transition selection and transition execution
Abstract Objects

We will see formal definitions of the following abstract objects (in which the semantic engine operates on)

- Variables
- Messages
- Channels
- Processes
- Transitions
- System states

Note: Formal definitions of basic terms like sets, identifiers, integers and booleans are not presented
Variables

• A Promela variable is defined by a five-tuple:

{name, scope, domain, inival, curval}

```c
short x=2, y=1;
active proctype not_euclid()
{
S: if
   :: x > y  ->  L: x = x - y
   :: x < y  ->    y = y - x
   :: x == y ->    assert(x != y); goto L
fi;
E: printf("%d\n", x)
}
```

- **name**: short
- **scope**: global
- **domain**: -2^{15}..2^{15}-1
- **inival**: x: 2
- **curval**:
  - at S: 2
  - at E: 1

---

Gerardo Schneider (Ifi, UiO)  
INF5140 - Lecture 9: Promela Semantics  
05.04.2006 52 / 69
• a message is a finite, ordered set of variables
  (messages are stored in channels – defined next)

```
mtype = { req, resp, ack };
chan q = [2] of { mtype, bit };
active proctype not_very_useful()
{  bit p;
    do
      :: q?req,p -> q!resp,p;
      :: q?resp,p -> q!ack,1-p
      :: q?ack,_
      :: timeout -> break
    od
}
```

place names for values held in message channel:
  slot1.field1
  slot1.field2

domains:
  mtype
  bit

parallel value assignment
Channels

- A message channel is defined by a 3-tuple \{ch_id, nslots, contents\}

\[
\text{chan q = [2] of \{ mtype, bit \};}
\]

- An ordered set of messages maximally with nslots elements:
  \[
  \{\{\text{slot}_1.\text{field}_1, \text{slot}_1.\text{field}_2\},\
  \{\text{slot}_2.\text{field}_1, \text{slot}_2.\text{field}_2\}\}
  \]

A ch_id is an integer 1..MAXQ that can be stored in a variable

(ch_id's \(\leq 0\) or \(> MAXQ\) do not correspond to any instantiated channel, so the default initial value of a variable of 0 is not a valid ch_id)

Variables of type chan are either local or global,

but channels always have global scope

(so, ch_id's are always meaningful when passed from one process to another)
**Channels**

**Channel Scope**

```plaintext
chan r = [0] of { chan };
active proctype A()
{
  chan q = [0] of { int };
  r!q;
  q?100
}
active proctype B()
{
  chan s;
  r?s;
  s!100
}
```

- **Global variable** `r` holds the ch_id of `C1`
  - `C1` never disappears
- **Local variable** `q` in `A` holds the ch_id of `C2`
  - `C2` is created when `A()` is instantiated
  - `C2` disappears when `A()` disappears
- **Local variable** `s` in `B` is initialized to 0 (not a valid ch_id)
  - It is set to `C2` in the receive from `r == C1`
  - `C2` is a globally visible object with a limited lifetime...

In the initial system state, `r`, `q`, `s`, `C1`, and `C2` all exist.
- `r` points to `C1`, `q` points to `C2`
• A process is defined by a six-tuple

\{\text{pid}, \text{lvars}, \text{lstates}, \text{inistate}, \text{curstate}, \text{transitions}\}

- The current state
- The initial state
- A finite set of integers defining local proc states
- A finite set of transitions (to be defined) between elements of lstates
- Finite set of local variables
- Process instantiation number
- Process p
  - p.curstate
  - p.pid
  - etc.
• A transition is defined by a seven-tuple
\{tri_id, source-state, target-state, cond, effect, priority, rv\}

condition and effect are defined for each basic statement, and they are typically defined on variable and channel values, possibly also on process states
System States

- A *global state* is defined by an eight-tuple

\{ gvars, procs, chans, exclusive, handshake, timeout, else, stutter \}

- A *finite set of global variables*
- A *finite set of processes*
- A *finite set of message channels*
- Predefined integer system variables that are used to define the semantics of atomic, d_step, and rendezvous
- Predefined Boolean system variables for stutter extension rule
- The global system state is called the system "state vector"
The semantics engine executes a Spin model step-by-step
In each step, one *executable* basic statement is selected
To determine whether a given statement is executable, the *executable clause* –specified in the Promela manual– must be evaluated
If more than one statement is executable, one of them is non-deterministically chosen
- Correctness of Spin models is independent of the selection criterion
For the selected statement, the *effect clause* (specified in the Promela manual) is applied
The control state of the process that executes the statement is updated
The semantics engine continues executing statements until no executable statements remain
- There is no more processes to execute; or
- The remaining processes are deadlocked
The Semantics Engine

The Algorithm

```c
global states s, s'
processes     p, p'
transitions   t, t'

1 while ((E = executable(s)) != {}) 
2 { 
3      for some (process p and transition t) from E 
4      {    s' = apply(t.effect, s) 
5           if (handshake == 0) 
6           {    s = s' 
7                p.curstate = t.target 
8           } else 
9           { /* try to complete rv handshake */ 
10                E' = executable(s') 
11                /* if E' is {}, s is unchanged */ 
12           } 
13           for some (process p' and transition t') from E' 
14           {    s = apply(t'.effect, s') 
15                p.curstate = t.target 
16                p'.curstate = t'.target 
17                handshake = 0 
18                break 
19           } 
20      }     }   }
21 } 
22 while (stutter) { s = s } /* stutter extension rule */
```
Examples
Rendez-vous Handshake

Example 1: Priority for sending in both processes  
Example 2: Priority for sending in one process and receiving in the other  
Example 3: Priority for receiving in both processes

chan x = [0] of { bit };  
chan y = [0] of { bit };  
active proctype A()  
{  
   x?0 unless y!0  
}  
active proctype B()  
{  
   y?0 unless x!0  
}  

Q: what is the combined system behavior?  
A: a non-deterministic selection between x!0;x?0 and y!0;y?0
Examples
Comparing Example 1 and 3

chan x = [0] of { bit };
chan y = [0] of { bit };

active proctype A()
{
    x!0 unless y?0
}

active proctype B()
{
    y!0 unless x?0
}
end

x!0 y!0
y?0 x?0

same global behavior
but for very different reasons....

chan x = [0] of { bit };
chan y = [0] of { bit };
active proctype A()
{
    x?0 unless y!0
}
active proctype B()
{
    y?0 unless x!0
}
end

x!0 y!0
y?0 x?0

same global behavior
but for very different reasons....
Remarks

- The semantics engine does not establish the validity or invalidity of correctness requirements.
  - The judgment of what is a correct system behavior is outside the definition of the Promela semantics.

- The addition of a verification option does not affect the semantics of a Promela model.
  - The semantics engine has no interpretation of valid end states, accepting states, non-progress states, never claims, trace assertions, etc.
  - These language elements have no formal semantics within the model.
  - Assertion statements, special labels, never claims, etc, are meta-statements about the semantics of the model.
  - How such meta-statements are to be interpreted is defined in a verifier, as part of the verification algorithm.
Without entering into details, these are few examples of how the verifier and the semantics engine are related

- When a verifier checks for safety properties, the predefined system variable `stutter` (used in the last line of the semantics engine) is set to `false`.

- When a verifier checks for liveness properties, the predefined system variable `stutter` is set to `true`.

- A `never` claim does not define new semantics, but is used to identify which part of the existing semantics can violate an independently stated correctness criterion.
  - Only infinite executions that are consistent with the formal semantics of the model and with the constraint expressed by the `never` claim can be generated.

These comments will become clear after next lecture (on the definition of correctness claims)…
The first part of this talk was based on Chapter 3 of Holzmann’s book “The Spin Model Checker” and the second part was based on Chap. 7 of the same book.

Next lecture we’ll see how to define Correctness Claims (Holzmann’s book Chap. 4).