

# Formal Methods and Functional Programming Modelling

**Peter Müller**

Chair of Programming Methodology  
ETH Zurich

The slides in this section are partly based on the course *Automata-based System Analysis* by  
Felix Klaedtke

# Example 1: Protocol Verification

- Protocol for file access with primitives `open`, `close`, and `write`
- Task: verify that a program obeys the following (informal) rules:
  - A. All opened files must be closed eventually
  - B. An opened file must be closed before the next open and vice versa
  - C. All files must be opened before executing a write operation

There is only one file, which is initially closed

- Problem is typical for verification of protocols
  - Locking (acquire, access, release)
  - Authentication (authenticate, access)

# Example 1: Encoding in IMP

- File is represented by variable  $f$ 
  - Write is encoded by assignment to  $f$
  - Variable  $o$  counts how often file was opened/closed
- Encoding of primitives:
  - open:  $o := o + 1$
  - close:  $o := o - 1$
  - write:  $f := e$
- Informal rules:
  - A. After  $o$  has been set to one, it must eventually be re-set to zero
  - B. In all execution states,  $o$  is zero or one
  - C. When  $f$  is being assigned to,  $o$  must be greater than zero

Variable  $o$  is initially zero

# Example 1: Specification in NS and Hoare Logic

A. For a terminating program  $s$ ,  $o$  must be zero in the terminal state

$$\langle s, \sigma \rangle \rightarrow \sigma' \text{ and } \sigma(o) = 0 \text{ then } \sigma'(o) = 0$$

$$\{ o = 0 \} s \{ o = 0 \}$$

Property cannot be expressed for non-terminating programs

# Example 1: Specification in NS and Hoare Logic

A. For a terminating program  $s$ ,  $o$  must be zero in the terminal state

$$\langle s, \sigma \rangle \rightarrow \sigma' \text{ and } \sigma(o) = 0 \text{ then } \sigma'(o) = 0$$

$$\{ o = 0 \} s \{ o = 0 \}$$

Property cannot be expressed for non-terminating programs

B. In all execution states,  $o$  is zero or one

- Natural semantics and Hoare logic can express properties of initial and terminal states, but **not of intermediate states**

# Example 1: Specification in NS and Hoare Logic

A. For a terminating program  $s$ ,  $o$  must be zero in the terminal state

$$\langle s, \sigma \rangle \rightarrow \sigma' \text{ and } \sigma(o) = 0 \text{ then } \sigma'(o) = 0$$

$$\{ o = 0 \} s \{ o = 0 \}$$

Property cannot be expressed for non-terminating programs

B. In all execution states,  $o$  is zero or one

- Natural semantics and Hoare logic can express properties of initial and terminal states, but **not of intermediate states**

C. When  $f$  is being assigned to,  $o$  must be greater than zero

- Natural semantics and Hoare logic can express properties of initial and terminal states, but **not of intermediate states**

# Example 1: Specification in SOS (A)

- A: After o has been set to one, it must eventually be re-set to zero
- For a **terminating** program s

$$\langle s, \sigma \rangle \rightarrow_1^* \sigma' \text{ and } \sigma(o) = 1 \text{ then } \sigma'(o) = 0$$

# Example 1: Specification in SOS (A)

- A: After  $o$  has been set to one, it must eventually be re-set to zero
  - For a **terminating** program  $s$

$$\langle s, \sigma \rangle \rightarrow_1^* \sigma' \text{ and } \sigma(o) = 0 \text{ then } \sigma'(o) = 0$$

- For a **deterministic, non-terminating** program  $s$

$$\langle s, \sigma \rangle \rightarrow_1^* \langle s', \sigma' \rangle \text{ and } \sigma(o) = 0 \text{ and } \sigma'(o) = 1 \text{ then there exist } s'', \sigma'' \text{ such that } \langle s', \sigma' \rangle \rightarrow_1^* \langle s'', \sigma'' \rangle \text{ and } \sigma''(o) = 0$$



# Example 1: Specification in SOS (A)

- A: After  $o$  has been set to one, it must eventually be re-set to zero
  - For a **terminating** program  $s$

$$\langle s, \sigma \rangle \rightarrow_1^* \sigma' \text{ and } \sigma(o) = 0 \text{ then } \sigma'(o) = 0$$

- For a **deterministic, non-terminating** program  $s$

$$\langle s, \sigma \rangle \rightarrow_1^* \langle s', \sigma' \rangle \text{ and } \sigma(o) = 0 \text{ and } \sigma'(o) = 1 \text{ then there exist } s'', \sigma'' \text{ such that } \langle s', \sigma' \rangle \rightarrow_1^* \langle s'', \sigma'' \rangle \text{ and } \sigma''(o) = 0$$

- For a **non-deterministic, non-terminating program**  $s$

$$wc : \text{Stm} \times \text{State} \times \mathbb{N} \rightarrow \text{Bool}$$

$$wc(s, \sigma, n) \Leftrightarrow \sigma(o) = 0 \vee$$

$$\text{(for all } s', \sigma' : \text{if } \langle s, \sigma \rangle \rightarrow_1 \langle s', \sigma' \rangle \text{ then there exists } m \in \mathbb{N} \text{ such that } m < n \text{ and } wc(s', \sigma', m))$$

$$\langle s, \sigma \rangle \rightarrow_1^* \langle s', \sigma' \rangle \text{ and } \sigma(o) = 0 \text{ and } \sigma'(o) = 1 \text{ then there exists } n \in \mathbb{N} \text{ such that } wc(s', \sigma', n)$$

# Example 1: Specification in SOS (B and C)

- B: In all execution states, o is zero or one

$$\langle s, \sigma \rangle \rightarrow_1^* \langle s', \sigma' \rangle \text{ and } \sigma(o) = 0 \text{ then } \sigma'(o) = 0 \text{ or } \sigma'(o) = 1$$

# Example 1: Specification in SOS (B and C)

- B: In all execution states, o is zero or one

$$\langle s, \sigma \rangle \rightarrow_1^* \langle s', \sigma' \rangle \text{ and } \sigma(o) = 0 \text{ then } \sigma'(o) = 0 \text{ or } \sigma'(o) = 1$$

- C: When f is being assigned to, o must be greater than zero
  - SOS cannot express properties of occurrences of statements
  - Work-around: Enrich state and make sure write changes the state

# Example 1: Verification

A. For a terminating program  $s$

$$\langle s, \sigma \rangle \rightarrow_1^* \sigma' \text{ and } \sigma(o) = 0 \text{ then } \sigma'(o) = 0$$

- Proof needs to consider **all possible derivations** of  $\langle s, \sigma \rangle \rightarrow_1^* \sigma'$  to find all possible terminal states
- Problematic in the presence of non-determinism or parallelism

# Example 1: Verification

A. For a terminating program  $s$

$$\langle s, \sigma \rangle \rightarrow_1^* \sigma' \text{ and } \sigma(o) = 0 \text{ then } \sigma'(o) = 0$$

- Proof needs to consider **all possible derivations** of  $\langle s, \sigma \rangle \rightarrow_1^* \sigma'$  to find all possible terminal states
- Problematic in the presence of non-determinism or parallelism

For a deterministic, non-terminating program  $s$

$$\langle s, \sigma \rangle \rightarrow_1^* \langle s', \sigma' \rangle \text{ and } \sigma(o) = 0 \text{ and } \sigma'(o) = 1 \text{ then there exist } s'', \sigma'' \text{ such that } \langle s', \sigma' \rangle \rightarrow_1^* \langle s'', \sigma'' \rangle \text{ and } \sigma''(o) = 0$$

- Proof about **infinite** derivation sequence **requires invariant**, which cannot be found automatically

# Example 1: Verification

A. For a terminating program  $s$

$$\langle s, \sigma \rangle \rightarrow_1^* \sigma' \text{ and } \sigma(o) = 0 \text{ then } \sigma'(o) = 0$$

- Proof needs to consider **all possible derivations** of  $\langle s, \sigma \rangle \rightarrow_1^* \sigma'$  to find all possible terminal states
- Problematic in the presence of non-determinism or parallelism

For a deterministic, non-terminating program  $s$

$$\langle s, \sigma \rangle \rightarrow_1^* \langle s', \sigma' \rangle \text{ and } \sigma(o) = 0 \text{ and } \sigma'(o) = 1 \text{ then there exist } s'', \sigma'' \text{ such that } \langle s', \sigma' \rangle \rightarrow_1^* \langle s'', \sigma'' \rangle \text{ and } \sigma''(o) = 0$$

- Proof about **infinite** derivation sequence **requires invariant**, which cannot be found automatically

B. In all execution states,  $o$  is zero or one

$$\langle s, \sigma \rangle \rightarrow_1^* \langle s', \sigma' \rangle \text{ and } \sigma(o) = 0 \text{ then } \sigma'(o) = 0 \text{ or } \sigma'(o) = 1$$

- Prove needs to consider **all possible derivations**

## Example 2: Verification of Parallel Programs

- A (simplified) Java program

```
class Cell {  
    int x = 0;  
  
    static void main(...) {  
        Cell c = new Cell();  
        Thread t1 = new Even(c);  
        Thread t2 = new Even(c);  
        t1.start(); t2.start();  
        t1.join();  t2.join();  
        System.out.println(c.x);  
    }  
}
```

```
class Even extends Thread {  
    Cell c;  
  
    Even(Cell c) {  
        this.c = c;  
    }  
  
    void run() {  
        c.x = c.x + 1;  
        c.x = c.x + 1;  
    }  
}
```

## Example 2: Verification of Parallel Programs

- A (simplified) Java program

```
class Cell {  
    int x = 0;  
  
    static void main(...) {  
        Cell c = new Cell();  
        Thread t1 = new Even(c);  
        Thread t2 = new Even(c);  
        t1.start(); t2.start();  
        t1.join(); t2.join();  
        System.out.println(c.x);  
    }  
}
```

```
class Even extends Thread {  
    Cell c;  
  
    Even(Cell c) {  
        this.c = c;  
    }  
  
    void run() {  
        c.x = c.x + 1;  
        c.x = c.x + 1;  
    }  
}
```

3



## Example 2: Encoding in IMP

- The following program  $s$  represents the core of the Java program
  - $x$  represents shared variable  $c.x$
  - $y$  and  $z$  represent thread-local state

```
(y := x;   y := y + 1;   x := y;  
  y := x;   y := y + 1;   x := y)  
par  
  (z := x;   z := z + 1;   x := z;  
   z := x;   z := z + 1;   x := z)
```

- Desired property:  
If  $x$  is zero in the initial state then  $x$  is even in the terminal state
  - NS and Hoare logic cannot handle parallelism
  - SOS specification:

$$\langle s, \sigma \rangle \rightarrow_1^* \sigma' \text{ and } \sigma(x) = 0 \text{ then } \sigma'(x) \bmod 2 = 0$$

## Example 2: Verification

- In this case, spotting the counterexample is easy, but how to attempt a formal proof?
- Induction does not work because there is **no suitable induction hypothesis**
  - Observation also holds for corrected example
- Proof strategy: **enumerate all possible derivations** of  $\langle s, \sigma \rangle \rightarrow_1^* \sigma'$  and inspect terminal state  $\sigma'$ 
  - **Number of derivations grows exponentially** in number of executed statements
  - Here,  $\frac{12!}{6! \times 6!} = 924$  possible derivations!
  - Manual enumeration not feasible, especially for programs with loops

# Examples: Observations

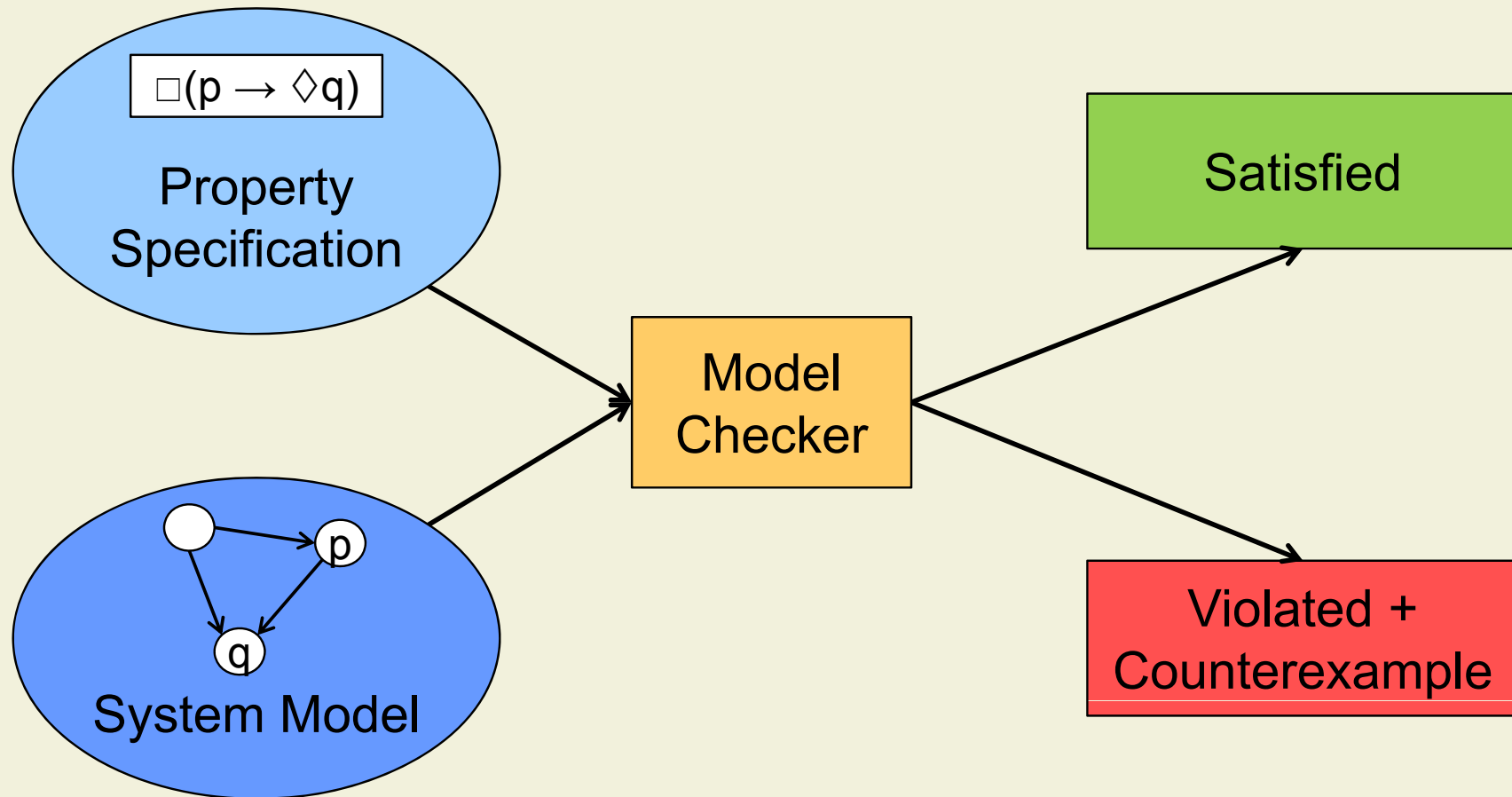
- Specification challenge
  - How to specify **properties of sequences of states** concisely
- Verification challenges
  - **Concurrent systems**: How to prove properties of all possible program executions
  - **Reactive systems**: How to prove automatically properties of infinite derivation sequences

# Model Checking

Model checking is an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for (a given state in) that model. [Baier and Katoen]

- Model checkers enumerate all possible states of a system:
  - Explicit state model checking:  
represent state explicitly through concrete values
  - Symbolic model checking:  
represent state through (boolean) formulas
- We focus on explicit state model checking

# Model Checking



# Model Checking Process

- Modelling phase
  - Model the system under consideration using the description language of your model checker (possibly a programming language)
  - Formalize the properties to be checked
- Running phase
  - Run the model checker to check the validity of the property in the system model
- Analysis phase
  - If property is satisfied, celebrate and move on to next property
  - If property is violated, analyze counterexample
  - If out of memory, reduce model and try again

# Main Purposes of Model Checking

- Model checking is mainly used to analyze **system designs** (as opposed to implementations)
- Typical properties to be analyzed include
  - Deadlocks
  - Reachability of undesired states
  - Protocol violations

# Modelling Concurrent Systems

- Systems are modelled as **finite transition systems**
- We model systems as **communicating sequential processes** (agents)
  - Finite number of processes
  - Interleaved process execution
- Processes can communicate via:
  - Shared variables
  - Synchronous message passing
  - Asynchronous message passing



# Protocol Meta Language Promala

- Input language of the Spin model checker
- Main objects are processes, channels, and variables
- C-like syntax

```
init {  
    printf("Hello World!\n")  
}
```

- Spin can “execute” (simulate) models
- References
  - Quick reference: [www.spinroot.com/spin/Man/Quick.html](http://www.spinroot.com/spin/Man/Quick.html)
  - Further references: [www.spinroot.com/spin/Man/index.html](http://www.spinroot.com/spin/Man/index.html)

# Promela Programs

- Constant declarations

```
#define N 5  
mtype = { ack, req };
```

- Structure declarations

```
typedef vector { int x; int y };
```

- Global channel declarations

```
chan buf = [2] of { int };
```

- Global variable declarations

```
byte counter;
```

- Process declarations

```
proctype myProc(int p) { ... }
```

# Promela Process Declarations

- Simple form

```
proctype myProc(int p) { ... }
```

- Body consists of a sequence of variable declarations, channel declarations, and statements
- No arrays as parameters

# Promela Process Declarations

- Simple form

```
proctype myProc(int p) { ... }
```

- Body consists of a sequence of variable declarations, channel declarations, and statements
- No arrays as parameters

- General form

```
active [N] proctype myProc(...) provided(E) priority M { ... }
```

- active: Start N instances of myProc in the initial state
- provided: E is an enabling condition, evaluated in the initial state
- priority: M indicates probability during random simulation ( $M \geq 1$ )

# Promela Process Declarations

- Simple form

```
proctype myProc(int p) { ... }
```

- Body consists of a sequence of variable declarations, channel declarations, and statements
- No arrays as parameters

- General form

```
active [N] proctype myProc(...) provided(E) priority M { ... }
```

- active: Start N instances of myProc in the initial state
  - provided: E is an enabling condition, evaluated in the initial state
  - priority: M indicates probability during random simulation ( $M \geq 1$ )
- Init process is started in the initial state

# Promela Process Declarations

- Simple form

```
proctype myProc(int p) { ... }
```

- Body consists of a sequence of variable declarations, channel declarations, and statements
- No arrays as parameters

- General form

```
active [N] proctype myProc(...) provided(E) priority M { ... }
```

- active: Start N instances of myProc in the initial state
  - provided: E is an enabling condition, evaluated in the initial state
  - priority: M indicates probability during random simulation ( $M \geq 1$ )
- Init process is started in the initial state
  - Deterministic processes lead to extra check during model analysis

```
D_proctype myProc(int p) { ... }
```

# Promela Types

- Primitive types

Type	Value range
bit or bool	0...1
byte	0...255
short	$-2^{15} \dots 2^{15} - 1$
int	$-2^{31} \dots 2^{31} - 1$

- No floats and mathematical integers

- Used-defined types

- Arrays: `int name[4]`
- Structures
- Type of symbolic constants: `mtype`

- Channel type: `chan`

# Promela Variable and Channel Declarations

- Variable declarations

```
byte a, b = 5, c;  
int d[3], e[4] = 3;  
mtype msg = ack;  
vector v;
```

- Variables are initialized to zero-equivalent values



# Promela Variable and Channel Declarations

- Variable declarations

```
byte a, b = 5, c;  
int d[3], e[4] = 3;  
mtype msg = ack;  
vector v;
```

- Variables are initialized to zero-equivalent values

- Channel declarations

```
chan c1 = [2] of { mtype, bit, chan };  
chan c2 = [0] of { int };  
chan c3;
```

- c1 can store up to two messages  
Messages sent via c1 consist of three parts (triples)
- c2 models rendez-vous communication
- c3 is uninitialized; must be assigned an initialized channel before usage

# Promela Variable and Channel Declarations

- Variable declarations

```
byte a, b = 5, c;  
int d[3], e[4] = 3;  
mtype msg = ack;  
vector v;
```

- Variables are initialized to zero-equivalent values

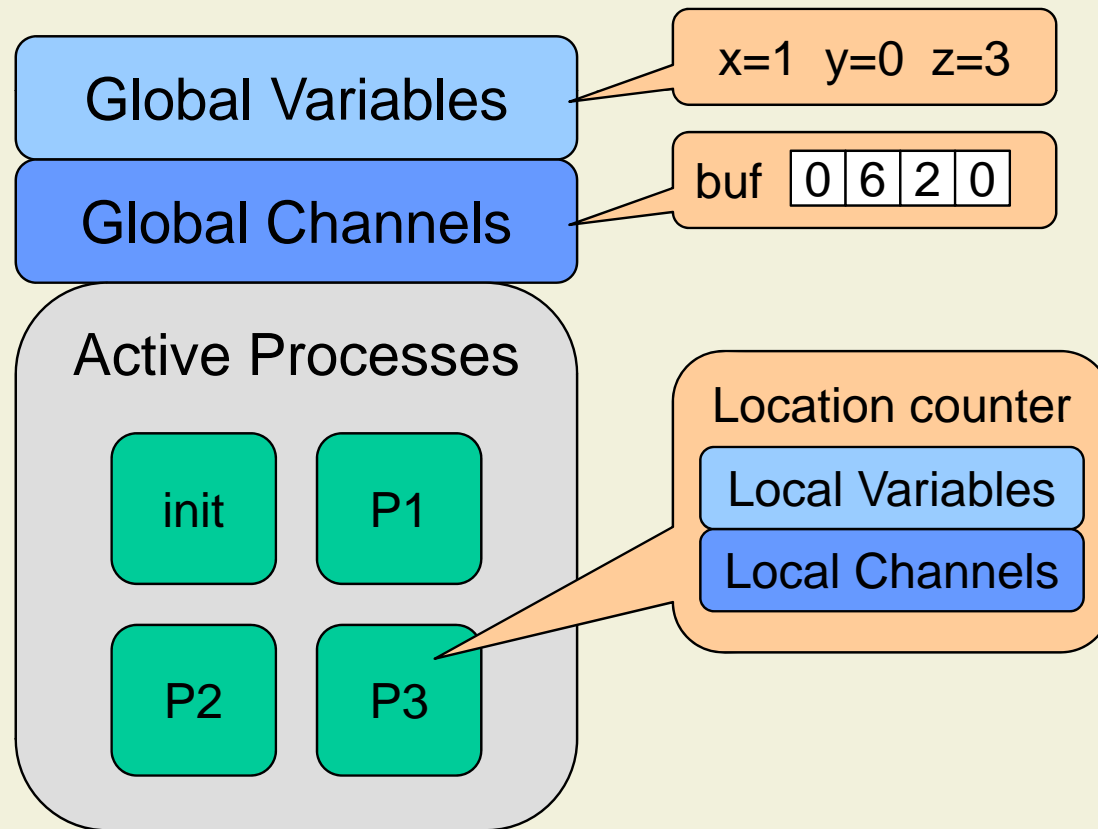
- Channel declarations

```
chan c1 = [2] of { mtype, bit, chan };  
chan c2 = [0] of { int };  
chan c3;
```

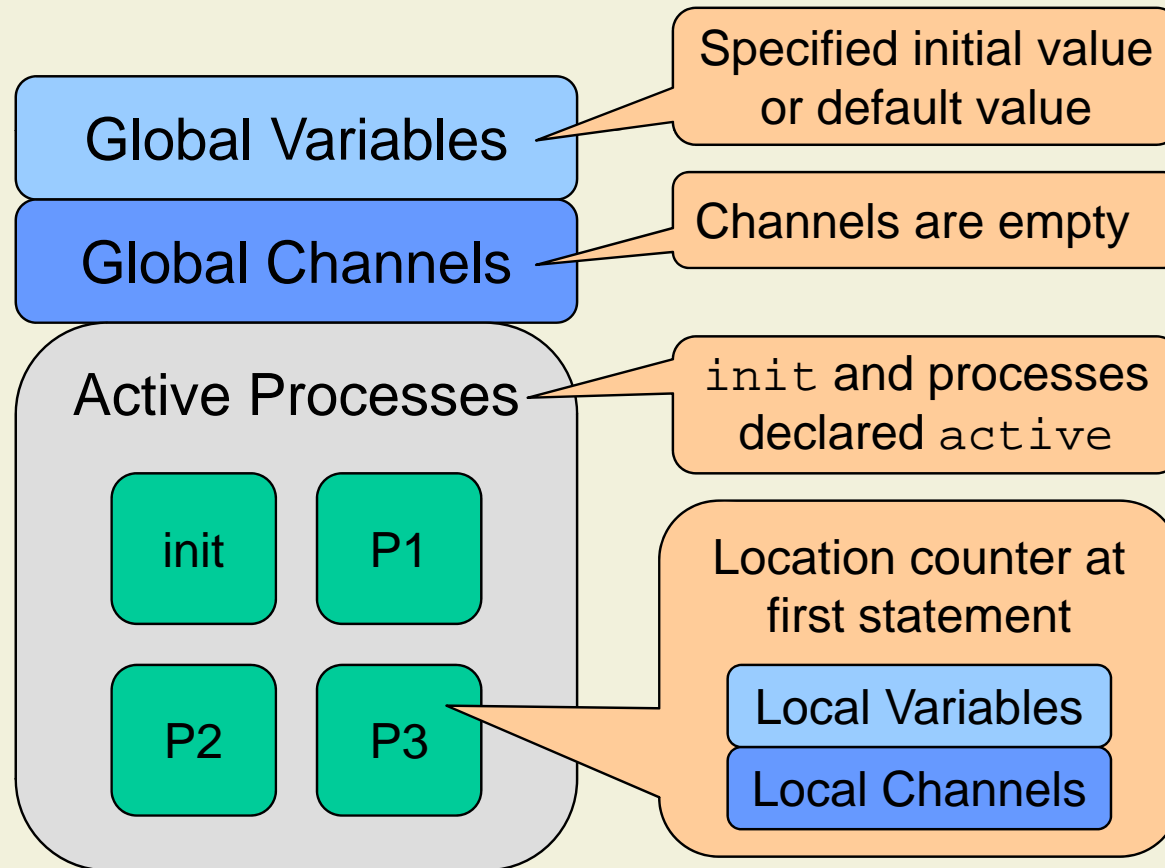
- c1 can store up to two messages  
Messages sent via c1 consist of three parts (triples)
- c2 models rendez-vous communication
- c3 is uninitialized; must be assigned an initialized channel before usage

- Variable and channel declarations are **local** to a process or **global**

# State Space of a Promela System



# Initial State



# State Transitions

- A statement can be **executable** or **blocked**
  - Send is blocked if channel is full
  - $s1;s2$  is blocked if  $s1$  is blocked
  - `timeout` is executable if all other statements are blocked

A transition is made in three steps:

- Determine **all executable statements** of all active processes
  - If no executable statement exists, transition system gets stuck
- Choose **non-deterministically** one of the executable statements
  - Non-determinism models concurrency through interleaving
- Change the state according to the chosen statement

# Promela Expressions

- Variables, constants, and literals
- Structure and array accesses
- Unary and binary expressions with operators

+	-	*	/	%	>
>=	<	<=	==	!=	!
&		&&		~	>>
<<	^	++	--		

- Function applications

len()	empty()	nempty()	nfull()	full()
run	eval()	enabled()	pcvalue()	

- Conditional expressions: (E1 -> E2 : E3)

# Promela Statements

- skip
  - Does not change the state
  - Always executable
- timeout
  - Does not change the state
  - Executable if all other statements in the system are blocked
- assert(E)
  - Aborts execution if expression E evaluates to zero; otherwise equivalent to skip
  - Always executable
- Assignment
  - $x = E$  assigns the value of E to variable x
  - $a[n] = E$  assigns the value of E to array element a[n]
  - Always executable

# Promela Statements (cont'd)

- Sequential composition
  - $s1; s2$  is executable if  $s1$  is executable
- Expression statement
  - Evaluates expression  $E$
  - Executable if  $E$  evaluates to value different from zero
  - $E$  must not change state (no side effects)
  - Common case: function applications
  - Examples:

```
printf("Hello World!\n");  
run myProcess;  
x > 0;
```



# Motivation: Verification of Parallel Programs

- A (simplified) Java program

```
class Cell {  
    int x = 0;  
  
    static void main(...) {  
        Cell c = new Cell();  
        Thread t1 = new Even(c);  
        Thread t2 = new Even(c);  
        t1.start(); t2.start();  
        t1.join(); t2.join();  
        System.out.println(c.x);  
    }  
}
```

```
class Even extends Thread {  
    Cell c;  
  
    Even(Cell c) {  
        this.c = c;  
    }  
  
    void run() {  
        c.x = c.x + 1;  
        c.x = c.x + 1;  
    }  
}
```

3

# Example: Modelling Even.run

```
class Even extends Thread {  
    Cell c;  
  
    Even(Cell c) {  
        this.c = c;  
    }  
  
    void run() {  
        c.x = c.x + 1;  
        c.x = c.x + 1;  
    }  
}
```

```
int x;  
  
proctype EvenRun() {  
    x = x + 1;  
    x = x + 1;  
}
```

```
int x;  
  
proctype EvenRun() {  
    int y = x;  
    y = y + 1;  
    x = y;  
    y = x;  
    y = y + 1;  
    x = y;  
}
```

# Example: Modelling Cell.main

```
class Cell {
    int x = 0;

    static void main(...) {
        Cell c = new Cell();
        Thread t1 = new Even(c);
        Thread t2 = new Even(c);
        t1.start(); t2.start();
        t1.join(); t2.join();
        System.out.println(c.x);
    }
}
```

```
init {
    x = 0;

    run EvenRun();
    run EvenRun();

    /* wait for termination */
    _nr_pr == 1;

    printf("x: %d\n", x);
    assert x % 2 == 0;
}
```

- `_nr_pr` is a predefined global variable that yields the number of active processes
- Simulation in Spin shows the possible outcomes 2, 3, and 4 (like Java program)

# Promela Statements: Selection

```
if
:: s1    /* option 1 */
:: ...
:: sn    /* option n */
fi
```

- Executable if at least one of its options is executable
- Chooses an option **non-deterministically** and executes it

```
if /* Move a sprite */
:: x < maxX -> x = x + 1;
:: x > minX -> x = x - 1;
:: y < maxY -> y = y + 1;
:: y > minY -> y = y - 1;
:: color = color + 1;
fi
```

- Statement `else` is executable if no other option is executable (may occur at most in one option)

# Promela Statements: Repetition

```
do
  :: s1    /* option 1 */
  :: ...
  :: sn    /* option n */
od
```

- Executable if at least one of its options is executable
- Chooses **repeatedly** an option **non-deterministically** and executes it
- Terminates when a break or goto is executed

```
/* compute factorial of n */

int r = 1;

do
  :: n > 1 -> r = r*n; n = n-1;
  :: else -> break
od
```

```
/* deadlock detection */
active proctype watchdog() {
  do
    :: timeout ->
      /* reset the state */
  od
}
```

# Promela Statements: Atomic

- Basic statements are executed **atomically**
  - No interleaving during execution of statement
  - skip, timeout, assert, assignment, expression statement
- `atomic { s }` executes `s` atomically
  - If any statement within `s` blocks, atomicity is lost, and other processes are then allowed to execute statements
- Example: Binary semaphores (locks)

```
bit locked; /* global */
```

```
/* lock */  
locked == 0;  
locked = 1;  
  
/* critical section */  
locked = 0; /* unlock */
```

```
/* lock */  
atomic {  
    locked == 0;  
    locked = 1;  
}  
/* critical section */  
locked = 0; /* unlock */
```

# Promela Macros

- Promela does not contain procedures
- Effect can often be achieved using **macros**

```
inline lock() {  
    atomic {  
        locked == 0;  
        locked = 1  
    }  
}
```

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

- A macro just defines a replacement text for a symbolic name, possibly with parameters
  - The inline call `lock()` is replaced by the body of the definition
  - No new variable scope
  - No recursion
  - No return values
- Define macro globally before its first use

# Motivation: Deadlock

- Threads are synchronized via locks
- Interleaved execution of `a.transfer(b,n)` and `b.transfer(a,m)` might **deadlock**
- Multi-threaded programs are **extremely hard to test**

```
class Account {  
    int balance;  
  
    void transfer(Account to, int amount) {  
        acquire this;  
        acquire to;  
        this.balance -= amount;  
        to.balance += amount;  
        release this;  
        release to;  
    }  
}
```



# Promela Model: Account

- We need to model accounts and clients
  - General approach: omit all irrelevant details to reduce complexity
- Account
  - Balance is not relevant for potential deadlocks
  - Only model the locks of accounts

```
#define N 5

bit Account_locks[N];

inline lock(n) {
    atomic {
        Account_locks[n] == 0;
        Account_locks[n] = 1;
    }
}
```

# Promela Model: Client

- Idea: model the most generic client and run several instances in parallel
  - Pick two arbitrary accounts **non-deterministically**
  - Lock both accounts
  - Unlock both accounts
- Choosing accounts

```
inline choose(a, l, u) {  
    a = l;  
    do  
        :: (a < u) -> a++  
        :: break  
    od  
}
```

```
inline chooseAccounts(f, t) {  
    do  
        :: (f != t) -> break  
        :: (f == t) -> choose(f, 0, N-1);  
                           choose(t, 0, N-1)  
    od  
}
```

# Promela Model: Client Process

```
active [C] proctype transfer() {
    byte from, to;

    /* choose accounts non-deterministically */
    chooseAccounts(from, to);

    /* acquire locks */
    lock(from);
    lock(to);

    /* actual transfer omitted */

    /* release locks */
    Account_locks[from] = 0;
    Account_locks[to] = 0;
}
```

# Alternative Account Selection

- Idea: instead of looping until two different accounts are found, restrict range for second choice

```
do
  :: (f != t) -> break
  :: (f == t) -> choose(f, 0, N-1);
                  choose(t, 0, N-1)
od
```

```
choose(f, 0, N-2);
choose(t, f+1, N-1)
```

# Alternative Account Selection

- Idea: instead of looping until two different accounts are found, restrict range for second choice

```
do
:: (f != t) -> break
:: (f == t) -> choose(f, 0, N-1);
               choose(t, 0, N-1)
od
```

```
choose(f, 0, N-2);
choose(t, f+1, N-1)
```

- Alternative model is **less general**
  - It guarantees  $\text{from} < \text{to}$
  - So locks are acquired in order and **deadlock is prevented!**

# Alternative Account Selection

- Idea: instead of looping until two different accounts are found, restrict range for second choice

```
do
:: (f != t) -> break
:: (f == t) -> choose(f, 0, N-1);
               choose(t, 0, N-1)
od
```

```
choose(f, 0, N-2);
choose(t, f+1, N-1)
```

- Alternative model is **less general**
  - It guarantees from < to
  - So locks are acquired in order and **deadlock is prevented!**
- General strategy
  - Start with most general model
  - If model contains errors that cannot occur in real system (**spurious error**), revise model

# Promela Channels

- `chan ch = [d] of { t1, ..., tn }` declares a channel
- Channel can buffer up to `d` messages
  - `d > 0`: buffered channel (FIFO)
  - `d = 0`: unbuffered channel (rendez-vous)
- Each message is a tuple whose elements have types `t1, ..., tn`
- Example

```
mtype = { req, ack, err };  
  
chan ch = [5] of { mtype, int }
```

# Send and Receive: Buffered Channels

```
chan ch = [5] of { mtype, int }
```

- `ch ! e1, ..., en` sends message
  - Type of `ei` must correspond to `ti` in channel declaration
  - Send is executable iff buffer is not full
- `ch ? a1, ..., an` receives message
  - `ai` is a variable or constant of type `ti`
  - Receive is executable iff buffer is not empty **and** the oldest message in the buffer **matches the constants** `ai`
  - Variables `ai` are assigned values of the message

```
ch ! req, 7;  
ch ! ack, 1
```

```
int n;  
ch ? req, n;  
printf("Received: %d\n", n);  
ch ? req, n;  
printf("Received: %d\n", n);
```



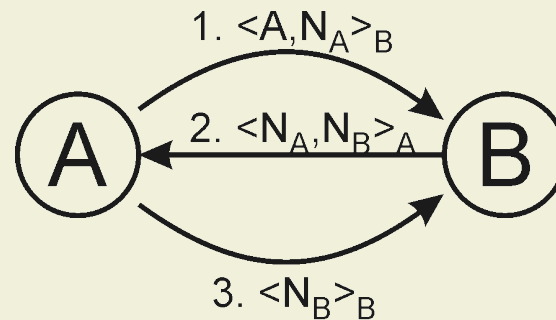
# Send and Receive: Unbuffered Channels

```
chan ch = [0] of { int };
```

- $ch ! e_1, \dots, e_n$  sends message
  - Send is executable if there is a receive operation that can be **executed simultaneously**
- $ch ? a_1, \dots, a_n$  receives message
  - Receive is executable if there is a send operation that can be **executed simultaneously**
- Unbuffered channels model **synchronous communication** (rendez-vous)

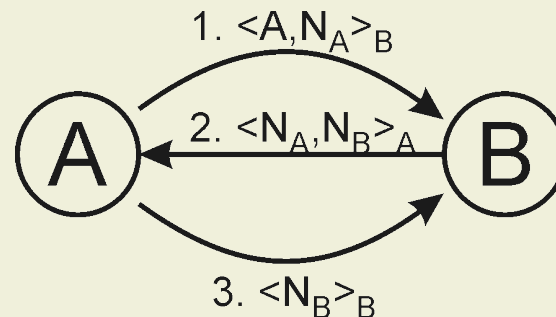
# Motivation: Needham-Schroeder Protocol

- Establish a common secret over an insecure channel
  - Alice sends random number  $N_A$  to Bob, encrypted with Bob's public key:  $\langle A, N_A \rangle_B$
  - Bob sends random number  $N_B$  to Alice, encrypted with Alice's public key:  $\langle N_A, N_B \rangle_A$
  - Alice responds with  $\langle N_B \rangle_B$



- Intruders may:
  - Intercept, store, and replay messages
  - Initiate or participate in runs of the protocol
  - Decrypt messages only if encrypted with intruder's public key
- Error: intruder can pretend to be another party

# Promela Model: Network



- We model the protocol for two agents plus intruder
- Agents communicate synchronously

```
chan network = [0] of {  
  mtype, /* tag:                msg1, msg2, msg3      */  
  mtype, /* intended receiver: agentA, agentB, agentI */  
  Crypt /* message              */  
};
```

- We use enumeration type `mtype` for all constants
  - Spin treats `mtype` constants as symbols, not values
  - Speeds up model checking

# Promela Model: Messages

- Message consists of key and up to two contents

```
typedef Crypt {  
    mtype key,          /* public key used to encrypt */  
        content1,      /* agent or nonce          */  
        content2       /* nonce or don't care    */  
};
```

- We model encryption by putting the public key into the message
  - Agent *a* will only look at message content if message key is *a*'s public key
  - No need to model private keys and encryption
- Constants for message tag, public keys, agents, and nonces

```
mtype = { msg1, msg2, msg3,  
          keyA, keyB, keyI,  
          agentA, agentB, agentI,  
          nonceA, nonceB, nonceI };
```

# Promela Model: Alice

- Protocol runs are always started by Alice

```
mtype partnerA;
bit    statusA; /* 1 = success */

active proctype Alice() {
    mtype pkey;      /* the partner's public key          */
    mtype pnonce;    /* nonce that we receive from partner */
    Crypt message;   /* our message to the partner        */
    Crypt data;      /* received message                  */

    if /* choose a partner for this run */
    :: partnerA = agentB; pkey = keyB;
    :: partnerA = agentI; pkey = keyI;
    fi;

    /* Protocol run below */
    statusA = 1; /* Success */
}
```

# Promela Model: Alice's Protocol Run

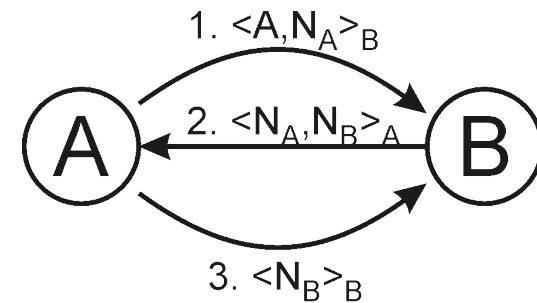
```
/* Prepare and send first message */  
build(message, pkey, agentA, nonceA);  
network ! msg1, partnerA, message;
```

```
/* Wait for answer */  
network ? (msg2, agentA, data);
```

```
/* Proceed only if the key matches keyA and the  
   nonce is the one that we have sent earlier */  
(data.key == keyA) && (data.content1 == nonceA);
```

```
/* Obtain partner's nonce */  
pnonce = data.content2;
```

```
/* Prepare and send the last message */  
build(message, pkey, pnonce, 0);  
network ! msg3, partnerA, message;
```



# Intruder

- Intruders may:
  - Intercept messages
  - Store one message
  - Replay messages
  - Initiate or participate in runs of the protocol
  - Decrypt messages only if encrypted with intruder's public key
- How can we model the most powerful attack using these capabilities?
- Solution: Model intruder **fully non-deterministically**
  - Intruder has no intelligence whatsoever
  - Model checker will **explore all possible** behaviors of intruder

# Promela Model: Intruder

```
bool knows_nonceA, knows_nonceB;

active proctype Intruder() {
    mtype tag;           /* message tag           */
    mtype recpt;         /* recipient for our message */
    Crypt data           /* received message         */
    Crypt intercepted;    /* stored message           */

    do
        :: /* Receive and learn */

        :: /* Replay or send      */
    od
}
```



# Promela Model: Intruder Receives

```
do
  :: network ? (tag, _, data) ->
    if /* perhaps store the message */
      :: copy(data, intercepted);
      :: skip;
    fi;
    if /* record newly learnt nonces */
      :: (data.key == keyI) ->
        knows_nonceA = knows_nonceA ||
          (data.content1 == nonceA) ||
          (data.content2 == nonceA);
        knows_nonceB = knows_nonceB ||
          (data.content1 == nonceB) ||
          (data.content2 == nonceB);

      :: else -> skip;
    fi;
  :: /* Replay or send */
od
```

# Promela Model: Intruder Sends

```
do
  :: /* Receive and learn */
  :: /* Replay or send */
  if /* choose message type */
    :: tag = msg1;
    :: tag = msg2;
    :: tag = msg3;
  fi;
  if /* choose recipient */
    :: recpt = agentA;
    :: recpt = agentB;
  fi;
  if /* replay intercepted message or assemble it */
    :: copy(intercepted, data);
    :: /* assemble new message */
  fi;
  network ! tag, recpt, data;
od
```

# Promela Model: Intruder Sends (cont'd)

```
:: /* assemble new message */
```

```
if
```

```
:: data.key = keyA;
```

```
:: data.key = keyB;
```

```
fi;
```

```
if
```

```
:: data.content1 = agentA;
```

```
:: data.content1 = agentB;
```

```
:: data.content1 = agentI;
```

```
:: knows_nonceA -> data.content1 = nonceA;
```

```
:: knows_nonceB -> data.content1 = nonceB;
```

```
:: data.content1 = nonceI;
```

```
fi;
```

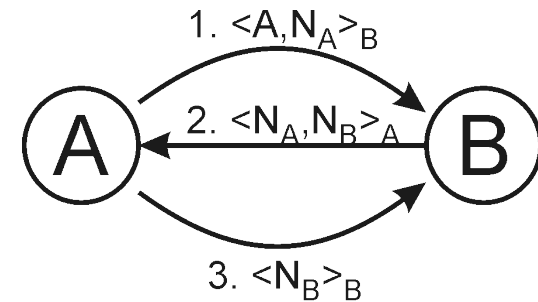
```
if
```

```
:: knows_nonceA -> data.content2 = nonceA;
```

```
:: knows_nonceB -> data.content2 = nonceB;
```

```
:: data.content2 = nonceI;
```

```
fi;
```



# Summary

- Models are **abstractions** of the real world
- **Omit irrelevant details** to reduce complexity
  - Example: balance in account example
- **Keep model small** to avoid state space explosion
  - As few process as possible
  - As little data as possible
- **Non-determinism** is a powerful modelling tool
  - Let model checker explore all options
- Typical sources of non-determinism are:
  - Abstraction
  - Modelling of the environment