ALBERT-LUDWIGS-UNIVERSITÄT FREIBURG

Network Protocol Design and Evaluation

05 - Validation, Part I

Stefan Rührup

University of Freiburg
Computer Networks and Telematics

Summer 2009

CoNe Freiburg

IIF INSTITUT FÜR INFORMATIK FREIBURG

Overview

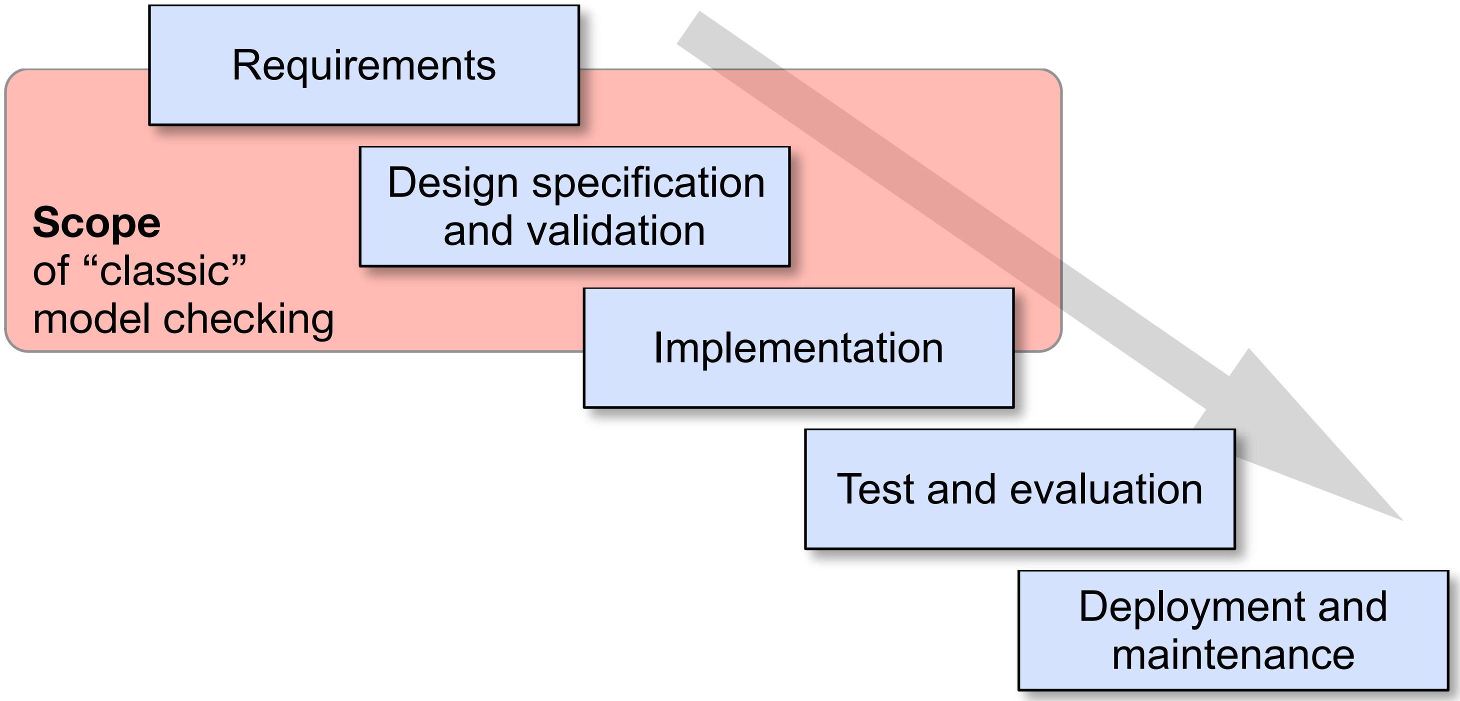
- **In the last lectures:**
  - Specification of protocols and data/message formats

- **In this chapter:**
  - Building a validation model
  - Verification with SPIN
  - Example: Validation of the Alternating Bit Protocol

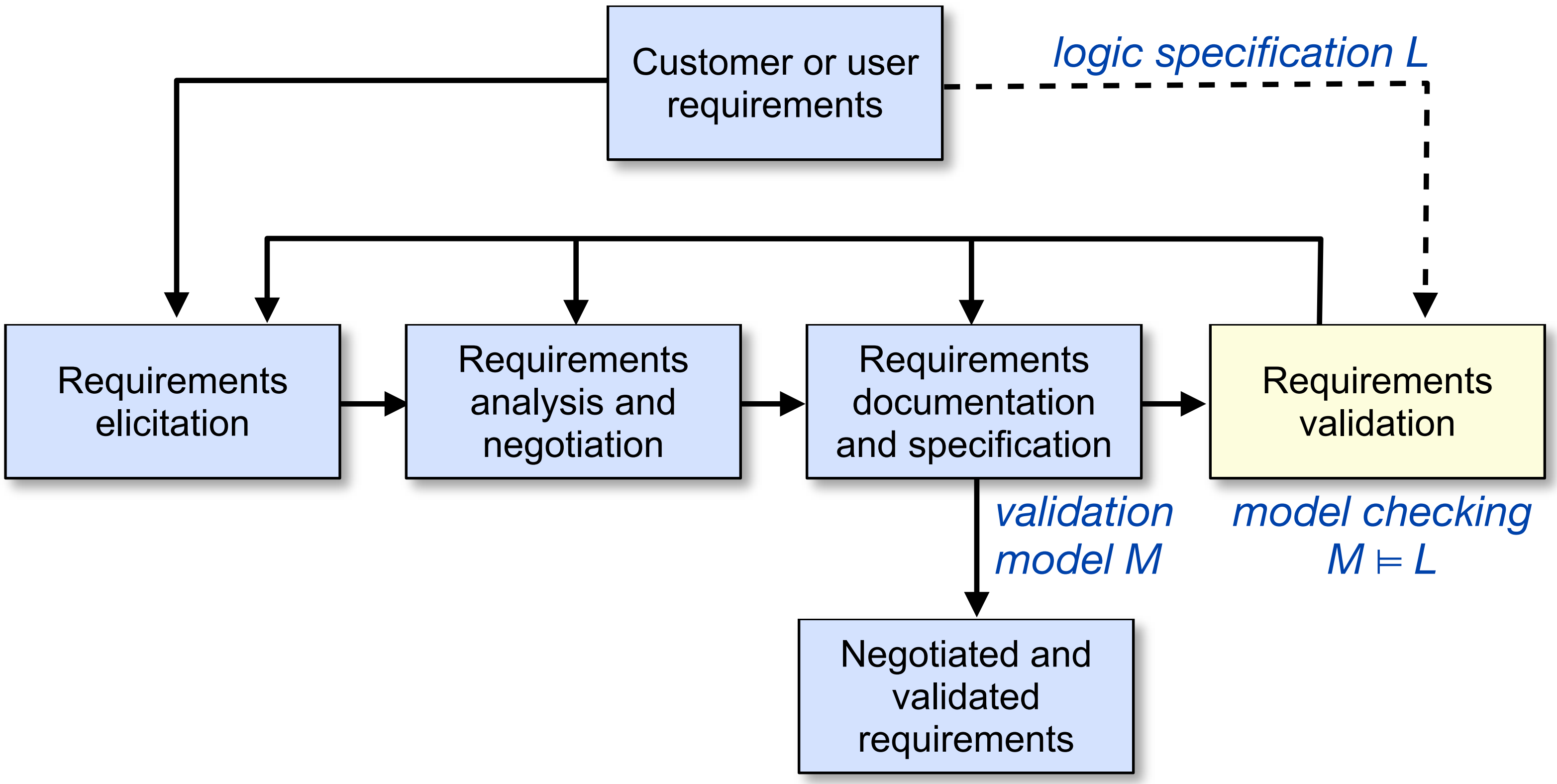
Validation and Model Checking

- **Validation models** for protocols:
  - Description of procedure rules (partial description)
  - Finite state model
  - Prototype of an implementation
- **Model checking**
  - Automated verification technique
  - Does a protocol satisfy some predefined logical properties?

Model Checking



Model checking



[S. Leue, Design of Reactive Systems, Lecture Notes, 2001]

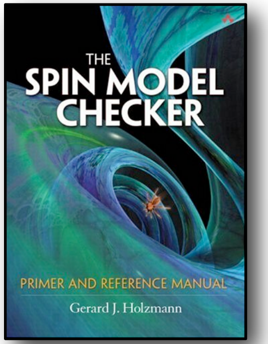
Model checking with SPIN

- **Outline**
  - Describing validation models in PROMELA
    (Protocol / Process Meta Language)
  - Simulation with SPIN
    (Simple Promela Interpreter)
  - Adding correctness properties
    (assertions, temporal claims)
  - Validation with SPIN: Building and executing a verifier

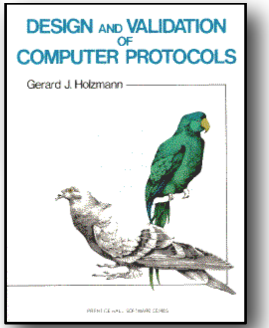
Promela & SPIN References

- **Online resources**
  - Lot's of documents on www.spinroot.com, e.g.
  - Tutorial: spinroot.com/spin/Doc/SpinTutorial.pdf
  - Manual: spinroot.com/spin/Man/Manual.html

- **Books:**
  - G. J. Holzmann: The SPIN Model Checker: Primer and Reference Manual, Addison-Wesley, 2003



  - G. J. Holzmann, Design and Validation of Computer Protocols, Prentice Hall, 1991

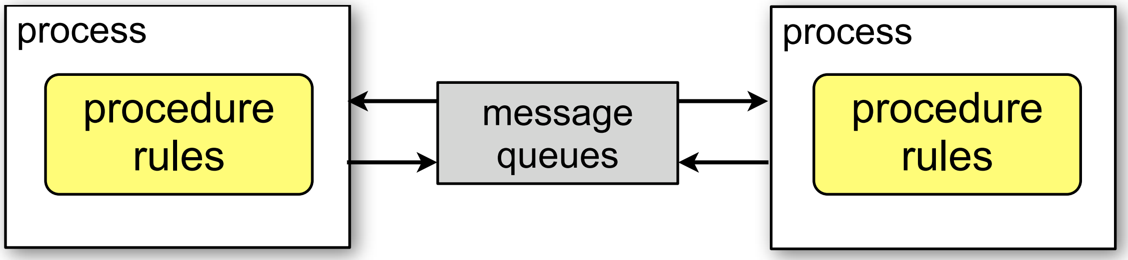


PROMELA

- **Pro**cess or **Pro**tocol **Me**ta **La**nguage
- Description Language for describing validation models
- Application in reactive systems design (not only communication protocols)
- Basis for model checking with SPIN

Promela Model

- Abstract model focusing on procedure rules (i.e. the behavior of the protocol)
- based on the communicating finite state machine model



- simplified data messages and channels
- not an implementation language, but a system description language

Promela Model

- **Building blocks:**
  - Processes (asynchronous)
  - Message channels (buffered and unbuffered)
  - Synchronizing statements
  - Structured data

- No clock, no floating point numbers, limited arithmetic functions

[Holzmann 2003]

Example

```
mtype = { msg0, msg1, ack0, ack1 };

chan to_sender = [2] of { mtype };
chan to_receiver = [2] of { mtype };

proctype Sender()
{
 again:
 to_receiver!msg1;
 to_sender?ack1;
 to_receiver!msg0;
 to_sender?ack0;
 goto again
 }

proctype Receiver()
{
 again:
 to_receiver?msg1;
 to_sender!ack1;
 to_receiver?msg0;
 to_sender!ack0;
 goto again
}

init{ run Sender(); run Receiver(); }
```

type declaration

channel declaration

process declaration

send statement

receive statement

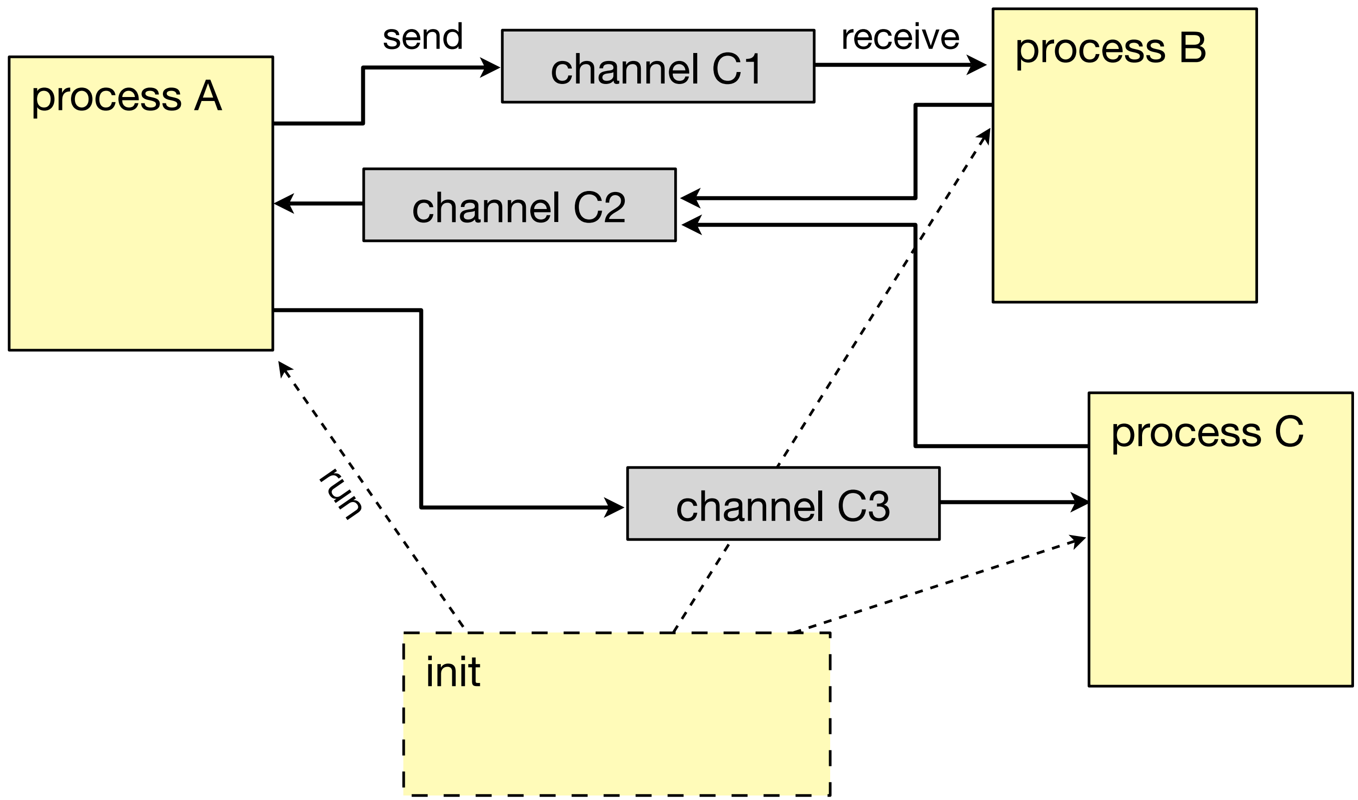
init process

Elements of a PROMELA Model

- Type declarations
- Channel declarations
- Variable declarations
- Process declarations
- The init process (optional)

```
mtype = { msg, ack }

chan StoR = ...
chan RtoS = ...

proctype Sender(chan in; chan out)
{
  bit sendBit, rcvBit;
  ...
}

init
{
  run Sender(RtoS, StoR);
  ...
}
```

Elements of a PROMELA Model



Processes (1)

- Building block of a Promela Model
- defined by a `proctype` definition
- Processes contain a list of statements
- ... and communicate via channels or via global variables

parameters

local variables

body

```
proctype Sender(chan in, out) {
   byte o,i;
   in?next(o);
   do
   ::in?msg(i) -> out!ack(o)
   ::in?err(i) -> out!nack(o)
   od
}
```

I/O statement

Processes (2)

- Processes run concurrently
- They are created by the `run` statement at any point (within the init process or any other process)
- ... or automatically by putting the `active` keyword in front of the proctype definition
- Several instances of the same type may be created

number of instances to be created

initial process (similar to the main function in C)

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

init {
  int pid = run Receiver(Rin, Rout)
}
```

run returns the process ID

Execution constraints

- Optional: Priorities and Constraints
- **Priorities** change the probability of execution in random simulations (default = 1; higher number = higher priority).
  - specified in proctype declarations or run-statements
- The **provided** clause constrains the execution with a global expression

```
byte a;

active proctype Sender(...) priority 2 provided (a > 1)
{
  ...
}
```

Data types and variables

- Basic data types: see table
- Arrays (one-dimensional)

```
byte a[16];
```

- Records

```
typedef Msg {
  int n1; int n2
}

Msg m;
```

- Variables are declared as in C
- Default initialization: 0

Type	Range
bit	0,1
bool	false,true
byte	0..255
chan	1..255
mtype	1..255
pid	0..255
short	$-2^{15}..2^{15}-1$
int	$-2^{31}..2^{31}-1$
unsigned	$0..2^{n}-1$ ($1 \leq n \leq 32$)

[Holzmann 2003]

Statements and Executability

- A process contains a sequence of statements, which are
  - assignments, e.g. `a = b`, or
  - expressions, e.g. `(a==b)`
- Statements are either *executable (enabled)* or *blocked*.
- Assignments are always executable
- An expression is executable, if its evaluation is non-zero

  Examples:

```
x >= 0    /* executable, if x is non-negative */
3 < 2     /* always blocked */
x - 1     /* executable, if x != 1 */
```

Special Statements

- `skip` statement: do nothing, always executable
- `run` statement: only executable if a new process can be created
- `goto` statement: jump to a label, always executable
- `assert` statement: used to check certain properties, always executable

Control Flow

- Statements are separated by “;” or “->”
- Case selection

```
if
:: (choice1) -> statement1a; statement1b
:: (choice2) -> statement2a; statement2b
fi
```

- Repetition

```
do
:: statement1;
:: (condition) -> break
od
```

- Jumps: `goto` *`label`*

Case selection (1)

guard statement

```
if
:: (choice1) -> statement1a; statement1b
:: (choice2) -> statement2a; statement2b
fi
```

- Only one sequence is executed required that the first statement is executable
- If more than one choice is executable, one sequence is chosen randomly and executed
- If no choice is executable, then the if-statement is blocked
- Here, the separator `->` is used to separate guards from the rest of the statement sequence

Case selection (2)

guard statement

```
if
:: (choice1) -> statement1a; statement1b
:: (choice2) -> statement2a; statement2b
:: else -> statement3
fi
```

- The else statement becomes executable, if all other guards are blocked.

Repetitions

```
do
:: (condition1) -> statement1a; statement1b
:: (condition2) -> statement2a; statement2b
:: (condition3) -> break
od
```

- do-statements behave like if-statements, but with *repeating* the choice
- The do-statement (do-loop) is ended by `break`

```
do
:: (condition) -> statement
:: else -> break
od
```

Jumps and Labels

- Statements and control flow constructs can be preceded by a label
- Labels can be the destination of `goto` jumps
- As labels have to precede a statement, a jump to the end of the program can be realized by

```
goto lastlabel;
...
lastlabel: skip
```

- There are special labels used in verification with the prefixes `accept`, `end`, and `progress`

Escape sequences

```
{
  statement_sequence_1;
}
unless { guard; statement_sequence_2 }
```

- Statements of the first sequence are repeated until the first statement in the unless-block (guard statement) becomes executable

Timeouts

- The `timeout` statement becomes executable, if all other statements are blocked.

Example: A process that sends a reset signal to a guard channel in case of a timeout [Holzmann 1991]

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

Channels

- Communication is modeled by sending and receving messages to and from *channels*
- Channels are FIFO message queues
- Declaration (with Initializer):

  ```
  chan name = [capacity] of {list of types}
  ```

  Examples:

  ```
  chan a;                     /* basic declaration */
  chan b[3];                  /* array of channels */
  chan c = [4] of {byte,int}
  ```

- Channels have to be initialized before they can be used

Channels and message fields

- Channel initialization with message fields:

```
chan c = [4] of {byte,int}
chan d = [1] of {mtype, short, bit}
```

- ... and the corresponding I/O statements:

```
c!expr1,expr2
d!msg,var1,var2
d!msg(var1,var2)  /* alternative notation */
```

- By convention, the first field should specify the message type.

Message type definitions

- Messages types are declared using `mtype`:

```
mtype  = {msg, ack, error}
```

- This defines and enumeration of three symbolic constants, which can be used later, e.g.:

```
mtype n = msg;
```

- Messages can carry variables (if the channel allows it)

```
byte data;
out!msg(data)
```

Message passing

- **Send statement**: The statement
  `ch!expr`
  sends the value of the expression `expr` to the channel `ch`. The expression can be a message variable. It is executable, if the channel is not full.

- **Receive statement**: The statement
  `ch?msg`
  receives a message from a channel and stores it into a the variable `msg`. It is executable, if the channel is not empty.

Conditional receive

- The receive statement with constant expressions

  ```
  ch?const1,const2
  ```

  removes the first message from the channel if the constants are matching with the message content.

  It is allowed to mix constant and variables:

  ```
  ch?const1,var
  ```

Sorted Send and Random Receive

- **Sorted Send** - Inserting messages in sorted numerical order (instead of FIFO):

  ```
  channel!!msg
  ```

- **Random Receive** - Retrieving random messages from a queue (instead of taking the first elemement out):

  ```
  channel??msg
  ```

  Random receive yields the first matching message

Channel polling

- The receive statement
  `ch!<x,y>`
  writes the message fields into the local variables x and y, but does not remove the message from the channel

- Testing without receiving: The statement
  `ch![msg]`
  is executed if there is a matching message, but the message is not removed from the channel.

Operations on Channels

- Operations on channels

```
len(ch)   /* number of messages stored in ch */
empty(ch)
full(ch)
```

- Example: testing if there is space in the channel before sending a message:

```
!full(ch) -> ch!msg
```

Race conditions

- Potential side-effects when using conditions, e.g.

```
(len(ch) > 0) -> ch?msg
ch?[msg] -> ch?msg
```

- In both cases, the second statement `ch?msg` is not necessarily executable after the first one! Other processes might access the channel in between.
- Solution:

```
atomic { ch?[msg] -> ch?msg }
```

Atomic sequences

- Sequences can be declared as atomic:
  - Examples:
    ```
    atomic{  run A; run B }
    atomic { ch?[msg] -> ch?msg }
    ```
  - The sequence may be non-deterministic

- Efficient alternative: `d_step {` *`sequence`* `}`
  - Deterministic indivisible sequence
  - No jumps into or from this sequence

Example: Test and Set (1)

- Problem: What is the resulting state?

```
byte state = 1;

proctype A()
{
        (state==1) -> state = state+1
}

proctype B()
{
        (state==1) -> state = state-1
}

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

Example: Test and Set (2)

- Solution: atomic statements

```
byte state = 1;

proctype A()
{    atomic {
        (state==1) -> state = state+1
     }
}

proctype B()
{    atomic {
        (state==1) -> state = state-1
     }
}

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

Rendezvous Communication

- Rendezvous port (instead of asynchronous communication)

  `chan port = [0] of {byte}`

- Zero-capacity channel, messages cannot be stored
- Example:

```
#define msgtype 33

chan port = [0] of { byte, byte };

active proctype A()
{   port!msgtype(101);
    port!msgtype(102)  /* not executable */
}

active proctype B()
{   byte state;
    port?msgtype(state)
}
```

Macros

- Promela models are processed by the C preprocessor, this allows to define
  - Constants
    ```
    #define MAX 16
    ```
  - Macros
    ```
    #define dummy(a,b)   (a+b)
    ```
  - (De-)activation of code fragments
    ```
    #define ACTIVATED 1

    #ifdef ACTIVATED
    #else
    #endif
    ```

Inline definitions

- Textual replacement
- Similar to macro definitions
- Cannot be used as an expression
- Inline sequence should not contain variable definitions

```
inline swap(x,y) {
  c = x;
  x = y;
  y = c
}

init {
  int a,b,c;

  swap(a,b)
}
```

```
init {
  int a,b,c;

  c = a;
  a = b;
  b = c
}
```

Assertions

- Assertions are inserted into the program code
- Basic assertion:

  ```
  assert(expression)
  ```

- (there are also trace assertions)

- Assertions = correctness properties
- can be checked during simulation
  (other types of correctness properties require to run SPIN in validation mode)

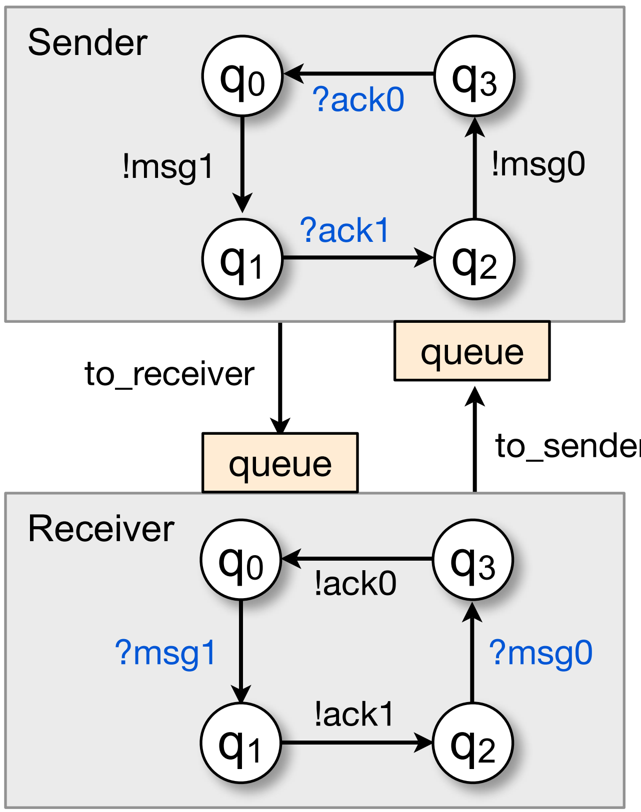
Input and Output

- Output can be generated using `printf()` as in C.

- Input is possible by reading integer numbers from STDIN.
  - Possibility of user-guided simulations
  - Usually, the model should be closed

Example: ABP in Promela

A simplified version of the Alternating Bit Protocol in Promela




```
mtype = { msg0, msg1, ack0, ack1 };

chan to_sender = [2] of { mtype };
chan to_receiver = [2] of { mtype };

active proctype Sender()
{
 again:
 to_receiver!msg1;
 to_sender?ack1;
 to_receiver!msg0;
 to_sender?ack0;
 goto again
}

active proctype Receiver()
{
 again:
 to_receiver?msg1;
 to_sender!ack1;
 to_receiver?msg0;
 to_sender!ack0;
 goto again
}
```

[Holzmann 2003]

What is this good for?

- Promela models can be simulated and automatically validated by the SPIN model checker

- **SPIN (Simple Promela Interpreter)**
  - developed by Gerard J. Holzmann, Bell Labs
  - open source
  - Command line or Tcl/Tk GUI (XSpin)

- Download: http://spinroot.com/spin/Src/

Example: Simulating ABP with SPIN

```
mtype = { msg0, msg1, ack0, ack1 };

chan to_sender = [2] of { mtype };
chan to_receiver = [2] of { mtype };

active proctype Sender()
{
 again:
 to_receiver!msg1;
 to_sender?ack1;
 to_receiver!msg0;
 to_sender?ack0;
 goto again
}

active proctype Receiver()
{
 again:
 to_receiver?msg1;
 to_sender!ack1;
 to_receiver?msg0;
 to_sender!ack0;
 goto again
}
```

[Holzmann 2003]

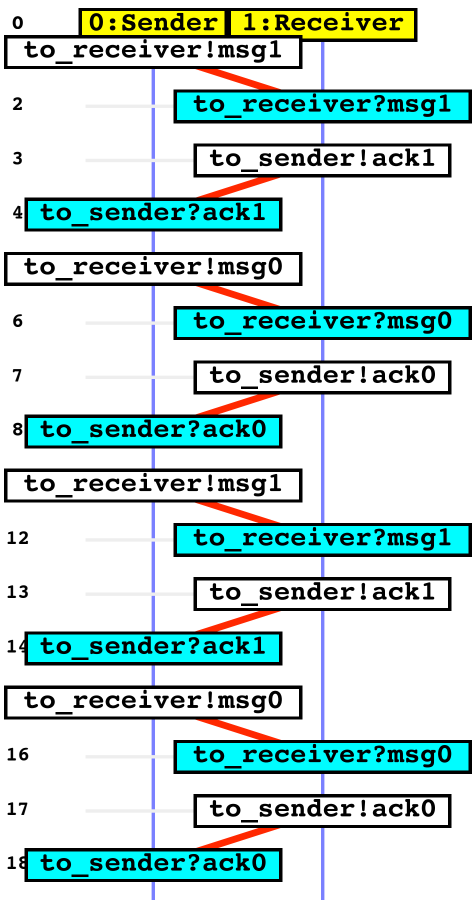
```
> spin -c -u14 abp.pml
proc 0 = Sender
proc 1 = Receiver
q\p   0   1
  1   to_receiver!msg1
  1   .   to_receiver?msg1
  2   .   to_sender!ack1
  2   to_sender?ack1
  1   to_receiver!msg0
  1   .   to_receiver?msg0
  2   .   to_sender!ack0
  2   to_sender?ack0
  1   to_receiver!msg1
  1   .   to_receiver?msg1
  2   .   to_sender!ack1
  2   to_sender?ack1
-------------
depth-limit reached
-------------
final state:
-------------
#processes: 2
          queue 2 (to_sender):
          queue 1 (to_receiver):
[msg0]
 15: proc  1 (Receiver) line  21
"abp.pml" (state 3)
 15: proc  0 (Sender) line  12
"abp.pml" (state 4)
2 processes created
```

Goodies: Generating MSCs

```
mtype = { msg0, msg1, ack0, ack1 };

chan to_sender = [2] of { mtype };
chan to_receiver = [2] of { mtype };

active proctype Sender()
{
 again:
 to_receiver!msg1;
 to_sender?ack1;
 to_receiver!msg0;
 to_sender?ack0;
 goto again
}

active proctype Receiver()
{
 again:
 to_receiver?msg1;
 to_sender!ack1;
 to_receiver?msg0;
 to_sender!ack0;
 goto again
}
```

[Holzmann 2003]

```
> spin -M -u steps
```

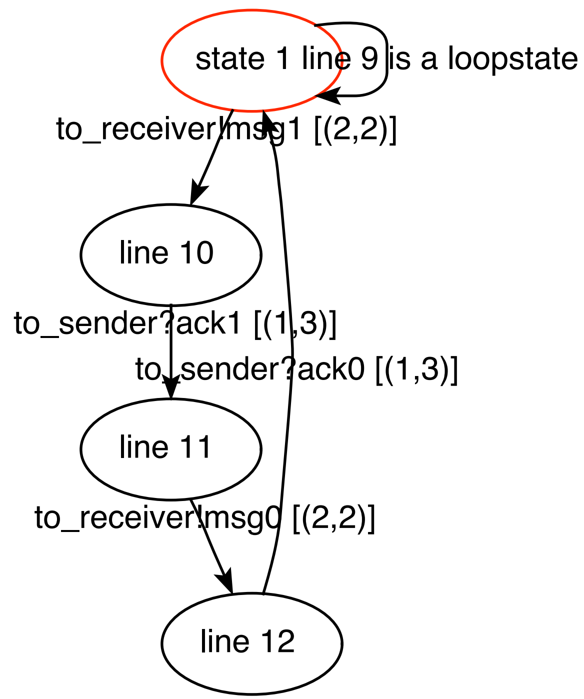


Goodies: Generating a state chart

```
mtype = { msg0, msg1, ack0, ack1 };

chan to_sender = [2] of { mtype };
chan to_receiver = [2] of { mtype };

active proctype Sender()
{
 again:
 to_receiver!msg1;
 to_sender?ack1;
 to_receiver!msg0;
 to_sender?ack0;
 goto again
}

active proctype Receiver()
{
 again:
 to_receiver?msg1;
 to_sender!ack1;
 to_receiver?msg0;
 to_sender!ack0;
 goto again
}
```

[Holzmann 2003]

XSPIN:
1. Run -> View state automaton
2. Select process



Lessons learned

- A validation model is an abstract system model
- Models are no timed. Any possible sequence of process interaction will be checked.
- We describe validation models in Promela, based on communicating (extended) finite state machines
- Special constructs in Promela: Statements and their executability