Behavior models and verification

Lecture 3

Jan Kofroň, František Plášil
Property specification

- Can take various forms
  - Informal expression (mapped to a formal meaning)
  - Textual description (again mapped)
  - Logical formula
    - LTL
    - CTL
    - ... and many others
Today: **Linear Temporal Logic**

- Also called **Propositional Temporal Logic**
- Able to capture many important properties
- Efficient model checking algorithm exists
  - Linear in the size of model
  - Exponential in the size of property formula
LTL formula has one of the following forms (\( \phi \) and \( \psi \) are formulae):
- \( 0, 1, p, \neg \phi, \phi \land \psi, \phi \lor \psi, \phi \Rightarrow \psi \)
- (for any variable \( p \) in \( AP \))
- \( X \phi \)
- \( G \phi \)
- \( F \phi \)
- \( \phi U \psi \)
LTL semantics

- Let $M = (S, I, R, L)$ be a Kripke structure
- Let $\pi = \pi_0 \rightarrow \pi_1 \rightarrow \pi_2 \ldots$ be an infinite path in $M$
- For an integer $i \geq 0$, $\pi^i$ stands for $i^{th}$ suffix of $\pi$, i.e.
  $\pi^i = \pi_i \rightarrow \pi_{i+1} \rightarrow \pi_{i+2} \ldots$

- Let $\varphi$ be a LTL formula
- $M, \pi \models \varphi$ stands for “a path $\pi$ from $M$ satisfies $\varphi$”
  - $M, \pi \models \varphi$ is defined by induction on the size of $\varphi$
- $M, s \models \varphi$ stands for “a state $s$ from $M$ satisfies $\varphi$”
  - $M, s \models \varphi$ is defined using $M, \pi \models \varphi$
  - **Definition:** $M, s \models \varphi \iff$ for all infinite paths $\pi$ in $M$ s.t. $\pi_0 = s$, $M, \pi \models \varphi$ holds
LTL semantics

- Let $M = (S, I, R, L)$ be a Kripke structure
- Let $\pi = \pi_0 \rightarrow \pi_1 \rightarrow \pi_2 \ldots$ be an infinite path in $M$
- For an integer $i \geq 0$, $\pi^i$ stands for $i^{th}$ suffix of $\pi$, i.e.
  $\pi^i = \pi_i \rightarrow \pi_{i+1} \rightarrow \pi_{i+2} \ldots$

- Let $\varphi$ be a LTL formula
- $M, \pi \models \varphi$ stands for "a path $\pi$ from $M$ satisfies $\varphi$"
  - $M, \pi \models \varphi$ is defined inductively
- $M, s \models \varphi$ stands for "a state $s$ from $M$ satisfies $\varphi$"
  - $M, s \models \varphi$ is defined using $M, \pi \models \varphi$
  - Definition: $M, s \models \varphi \iff$ for all infinite paths $\pi$ in $M$ s.t. $\pi_0 = s$, $M, \pi \models \varphi$ holds

Q: How many of them can this mean?
LTL semantics

- Let $M = (S, I, R, L)$ be a Kripke structure
- Let $\pi = \pi_0 \rightarrow \pi_1 \rightarrow \pi_2 \ldots$ be an infinite path in $M$
- For an integer $i \geq 0$, $\pi^i$ stands for the $i^{th}$ suffix of $\pi$, i.e.
  $\pi^i = \pi_i \rightarrow \pi_{i+1} \rightarrow \pi_{i+2} \ldots$

- Let $\varphi$ be a LTL formula
- $M, \pi \models \varphi$ stands for "a path $\pi$ from $M$ satisfies $\varphi$"
  - $M, \pi \models \varphi$ is defined using $M, \pi_0 \models \varphi$
- $M, s \models \varphi$ stands for "a state $s$ from $M$ satisfies $\varphi$"
  - $M, s \models \varphi$ is defined using $M, \pi \models \varphi$
  - **Definition:** $M, s \models \varphi \iff$ for all infinite paths $\pi$ in $M$ s.t. $\pi_0 = s$, $M, \pi \models \varphi$ holds

**A: Infinitely many.**
LTL semantics

**Definition**

- $M, \pi \models p$ $\iff$ $p \in L(\pi_0)$
- $M, \pi \models \neg \varphi_1$ $\iff$ not $M, \pi \models \varphi_1$
- $M, \pi \models \varphi_1 \lor \varphi_2$ $\iff$ $M, \pi \models \varphi_1$ or $M, \pi \models \varphi_2$
- $M, \pi \models \varphi_1 \land \varphi_2$ $\iff$ $M, \pi \models \varphi_1$ and $M, \pi \models \varphi_2$
- $M, \pi \models X \varphi_1$ $\iff$ $M, \pi^1 \models \varphi_1$
- $M, \pi \models F \varphi_1$ $\iff$ there exists an integer $i \geq 0$ s.t. $M, \pi^i \models \varphi_1$
- $M, \pi \models G \varphi_1$ $\iff$ for all integers $i \geq 0$, $M, \pi^i \models \varphi_1$
- $M, \pi \models \varphi_1 \cup \varphi_2$ $\iff$ there exists an integer $i \geq 0$ s.t.:
  - $M, \pi^i \models \varphi_2$ and
  - for all $j$, $0 \leq j < i$, $M, \pi^j \models \varphi_1$ holds
LTL semantics

- **Definition**
  - $M, \pi \vdash p \iff p \in L(\pi)$
  - $M, \pi \vdash \neg \phi_1 \iff \text{not } M, \pi \vdash \phi_1$
  - $M, \pi \vdash \phi_1 \lor \phi_2 \iff M, \pi \vdash \phi_1 \text{ or } M, \pi \vdash \phi_2$
  - $M, \pi \vdash \phi_1 \land \phi_2 \iff M, \pi \vdash \phi_1 \text{ and } M, \pi \vdash \phi_2$
  - $M, \pi \vdash X \phi_1 \iff M, \pi^1 \vdash \phi_1$
  - $M, \pi \vdash F \phi_1 \iff \text{there exists an integer } i \geq 0 \text{ s.t. } M, \pi^i \vdash \phi_1$
  - $M, \pi \vdash G \phi_1 \iff \text{for all integers } i \geq 0, M, \pi^i \vdash \phi_1$
  - $M, \pi \vdash \phi_1 \lor \phi_2 \iff \text{there exists an integer } i \geq 0 \text{ s.t.:}$
    - $M, \pi^i \vdash \phi_2 \text{ and}$
    - $\text{for all } j, 0 \leq j < i, M, \pi^j \vdash \phi_1 \text{ holds}$
LTL examples

• Mutex specification
  - Safety:  \( G \neg (c_1 \land c_2) \)
  - Liveness:  \( G(t_1 \Rightarrow F c_1) \)
  - Non-blocking:  \( G(n_1 \Rightarrow X t_1) \)
  - No strict sequencing. The following should be false at \( s_0 \):
    \[
    G \left[ c_1 \Rightarrow (\neg c_2 \land (c_1 \land \neg c_2) U (\neg c_1 \land (\neg c_1 \land \neg c_2) U c_2))) \right]
    \land
    G \left[ c_2 \Rightarrow (\neg c_1 \land (c_2 \land \neg c_1) U (\neg c_2 \land (\neg c_2 \land \neg c_1) U c_1))) \right]
    \]

• On all paths, all \( p \) precede \( s \) and \( t \)
  \[
  F p \land F s \land F t \land ((\neg s \land \neg t) U G \neg p)
  \]
• Other examples:
  - G (F ServiceAvailable)
  - G (Request \(\Rightarrow\) F Acknowledgment)
SPIN Model Checker
SPIN - Introduction

- **SPIN** (= Simple Promela Interpreter)
  - is a tool for analysing the logical consistency of concurrent systems, specifically of data communication protocols.
  - state-of-the-art model checker, used by >2000 users
    - Concurrent systems are described in the modelling language called Promela.

- **Promela** (= Protocol/Process Meta Language)
  - allows for the dynamic creation of concurrent processes.
  - communication via message channels can be defined to be
    - synchronous (i.e. rendezvous), or
    - asynchronous (i.e. buffered).
  - resembles the programming language
  - specification language to model finite-state systems

+ features from CSP
Promela Model

- Promela model consist of:
  - type declarations
  - channel declarations
  - variable declarations
  - process declarations
  - \([\textit{init} \text{ process}]\)

- A Promela model corresponds with a (usually very large, but) finite transition system, so
  - no unbounded data
  - no unbounded channels
  - no unbounded processes
  - no unbounded process creation

```c
mtype = \{MSG, ACK\};
chan toS = ...  chan toR = ...  bool flag;

proctype Sender() {
    ...
}
            \textcolor{red}{\text{process body}}

proctype Receiver() {
    ...
}

init {
    ...
} \textcolor{red}{\text{creates processes}}
```
Processes (1)

- A process type (`proctype`) consist of
  - a name
  - a list of formal parameters
  - local variable declarations
  - body

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

The body consist of a sequence of statements.
Processes (2)

- A process
  - is defined by a proctype definition
  - executes concurrently with all other processes, independent of speed of behaviour
  - communicate with other processes
    - using global (shared) variables
    - using channels
- There may be several processes of the same type.
- Each process has its own local state:
  - process counter (location within the proctype)
  - contents of the local variables
Processes (3)

- Process are created using the run statement (which returns the process id).

- Processes can be created at any point in the execution (within any process).

- Processes start executing after the run statement.

- Processes can also be created by adding active in front of the proctype declaration.

```plaintext
proctype Foo(byte x) {
    ...
}

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

```plaintext
active[3] proctype Bar() {
    ...
}
```

number of procs. (opt.)

parameters will be initialised to 0
Hello World!

/* 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);
}

$ spin -n2 hello.pr
init process, my pid is: 1
    last pid was: 2
Hello process, my pid is: 0
    Hello process, my pid is: 2
3 processes created
Variables and Types (1)

- Five different (integer) basic types.
- Arrays
- Records (structs)
- Type conflicts are detected at runtime.
- Default initial value of basic variables (local and global) is 0.

Basic types

bit turn=1; [0..1]
bool flag; [0..1]
byte counter; [0..255]
short s; [-2^{15}-1..2^{15}-1]
int msg; [-2^{31}-1..2^{31}-1]

Arrays

byte a[27];
bit flags[4];

Typedef (records)

typedef Record {
  short f1;
  byte f2;
}
Record rr;
rr.f1 = ..

array indexing start at 0
variable declaration

20
Statements (1)

• The body of a process consists of a sequence of statements. A statement is either
  – executable: the statement can be executed immediately.
  – blocked: the statement cannot be executed.

• An assignment is always executable.

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

  - \( 2 < 3 \)  always executable
  - \( x < 27 \)  only executable if value of \( x \) is smaller 27
  - \( 3 + x \)  executable if \( x \) is not equal to \(-3\)
Statements (2)

- The `skip` statement is always executable.
  - “does nothing”, only changes process’ process counter
- A `run` statement is only executable if a new process can be created (remember: the number of processes is bounded).
- A `printf` statement is always executable (but is not evaluated during verification, of course).

```c
int x;
proctype Aap()
{
  int y=1;
  skip;
  run Noot();
  x=2;
  x>2 && y==1;
  skip;
}
```

Executable if `Noot` can be created...

Can only become executable if a some other process makes `x` greater than 2.
Statements (3)

- `assert(<expr>);`
  - The `assert`-statement is always executable.
  - If `<expr>` evaluates to zero, SPIN will exit with an error, as the `<expr>` “has been violated”.
  - The `assert`-statement is often used within Promela models, to check whether certain properties are valid in a state.

```proctype monitor() {
    assert(n <= 3);
}

proctype receiver() {
...
    toReceiver ? msg;
    assert(msg != ERROR);
...
```
Mutual Exclusion (1)

bit flag; /* signal entering/leaving the section */
byte mutex; /* # procs in the critical section. */

proctype P(bit i) {
    if (flag != 1;)
    flag = 1;
    mutex++;
    printf("MSC: P(%d) has entered section.\n", i);
    mutex--;
    flag = 0;
}

proctype monitor() {
    assert(mutex != 2);
}

init {
    atomic {
        run P(0); run P(1); run monitor();
    }
}

Problem: assertion violation!
Both processes can pass the
flag != 1 "at the same time",
i.e. before flag is set to 1.

starts two instances of process P
Mutual Exclusion (2)

```c
bit x, y;  /* signal entering/leaving the section */
byte mutex; /* # of procs in the critical section. */

active proctype A() {
  x = 1;
  y == 0;
  mutex++;  // Process A waits for
  mutex--;  // process B to end.
  x = 0;
}

active proctype B() {
  y = 1;
  x == 0;
  mutex++;  // Problem: invalid-end-state!
  mutex--;  // Both processes can pass execute
  y = 0;
}

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

Problem: invalid-end-state!
Both processes can pass execute $x = 1$ and $y = 1$ “at the same time”, and will then be waiting for each other.
The if-statement (1) is inspired by Dijkstra's guarded command language.

```
if
:: choice₁ -> stat₁.₁; stat₁.₂; stat₁.₃; ...
:: choice₂ -> stat₂.₁; stat₂.₂; stat₂.₃; ...
:: ...
:: choiceₙ -> statₙ.₁; statₙ.₂; statₙ.₃; ...
fi;
```

- If there is at least one choiceᵢ (guard) executable, the if-statement is executable and SPIN non-deterministically chooses one of the executable choices.
- If no choiceᵢ is executable, the if-statement is blocked.
- The operator “->” is equivalent to “;”. By convention, it is used within if-statements to separate the guards from the statements that follow the guards.
if-statement (2)

if
:: (n % 2 != 0) -> n=1
:: (n >= 0) -> n=n-2
:: (n % 3 == 0) -> n=3
:: else -> skip
fi

- The **else** guard becomes executable if none of the other guards is executable.

**give n a random value**

if
:: skip -> n=0
:: skip -> n=1
:: skip -> n=2
:: skip -> n=3
fi

**non-deterministic branching**

**skips are redundant, because assignments are themselves always executable...**
do-statement (1)

\[
\begin{align*}
\text{do} \\
\text{:: } \text{choice}_1 & \rightarrow \text{stat}_{1.1}; \text{stat}_{1.2}; \text{stat}_{1.3}; ... \\
\text{:: } \text{choice}_2 & \rightarrow \text{stat}_{2.1}; \text{stat}_{2.2}; \text{stat}_{2.3}; ... \\
\text{:: } ... \\
\text{:: } \text{choice}_n & \rightarrow \text{stat}_{n.1}; \text{stat}_{n.2}; \text{stat}_{n.3}; ... \\
\text{od};
\end{align*}
\]

- With respect to the choices, a **do**-statement behaves in the same way as an **if**-statement.

- However, instead of ending the statement at the end of the chosen list of statements, a **do**-statement repeats the choice selection.

- The (always executable) **break** statement exits a **do**-loop statement and transfers control to the end of the loop.
Communication (1)

![Diagram of communication between Sender and Receiver with中标注的MSG和ACK消息的发送和接收]

- $s2r$ is sending
- $r2s$ is receiving

Thursday 11-Apr-2002  Theo C. Ruys - SPIN Beginners' Tutorial
Communication (2)

- Communication between processes is via **channels**:
  - message passing
  - rendez-vous synchronisation (handshake)

- Both are defined as **channels**:

  ```
  chan <name> = [<dim>] of {<t_1>,<t_2>, ... <t_n>};
  ```

  - name of the channel
  - type of the elements that will be transmitted over the channel
  - number of elements in the channel
    - dim==0 is special case: rendez-vous

  ```
  chan c = [1] of {bit};
  chan toR = [2] of {mtype, bit};
  chan line[2] = [1] of {mtype, Record};
  ```

  - array of channels
Communication (3)

- channel = FIFO-buffer (for \texttt{dim}>0)

\textbf{! Sending - putting a message into a channel}

\texttt{\textcolor{red}{ch}} \texttt{! <expr_1>, <expr_2>, \ldots <expr_n>};

- The values of \texttt{<expr_i>} should correspond with the types of the channel declaration.
- A \texttt{send}-statement is executable if the channel is not full.

\textbf{? Receiving - getting a message out of a channel}

\texttt{<var_1> + <const_1> can be mixed}

\texttt{\textcolor{red}{ch}} \texttt{? <var_1>, <var_2>, \ldots <var_n>}; \textcolor{green}{\text{message passing}}

- If the channel is not empty, the message is fetched from the channel and the individual parts of the message are stored into the \texttt{<var_i>}s.

\texttt{\textcolor{red}{ch}} \texttt{? <const_1>, <const_2>, \ldots <const_n>}; \textcolor{green}{\text{message testing}}

- If the channel is not empty and the message at the front of the channel evaluates to the individual \texttt{<const_i>}, the statement is executable and the message is removed from the channel.
Communication (4)

- **Rendez-vous communication**
  
  `<dim> == 0`
  
  The number of elements in the channel is now *zero*.
  
  - If `send ch!` is enabled and if there is a corresponding `receive ch?` that can be executed simultaneously and the constants match, then both statements are enabled.
  
  - Both statements will "handshake" and *together* take the transition.

- **Example:**

  ```
  chan ch = [0] of {bit, byte};
  - P wants to do `ch ! 1, 3+7`
  - Q wants to do `ch ? 1, x`
  - Then after the communication, *x* will have the value *10*.
  ```
Processes in Promela

- Interleaving semantics
  - Each time, a process is selected and its current statement is executed
    - Has to be \textit{enabled}
  - This is repeated
- Number of all possible interleavings may be very high
  - \(\rightarrow\) state space explosion \(\rightarrow\) not verifiable models
- A mechanism to control the interleavings would be handy
proctype P1() { t1a; t1b; t1c }
proctype P2() { t2a; t2b; t2c }
init { run P1(); run P2() }

Not completely correct as each process has an implicit end-transition...
proctype P1() { atomic {t1a; t1b; t1c} }
proctype P2() { t2a; t2b; t2c }
init { run P1(); run P2() }

It is as if P1 has only one transition...

If one of P1's transitions blocks, these transitions may get executed

Although atomic clauses cannot be interleaved, the intermediate states are still constructed.
proctype P1() { d_step {t1a; t1b; t1c} }
proctype P2() { t2a; t2b; t2c }
init { run P1(); run P2() }

It is as if P1 has only one transition...

No intermediate states will be constructed.
Checking for pure atomicity

- Suppose we want to check that none of the atomic clauses in our model are ever blocked (i.e. pure atomicity).

1. Add a global bit variable:
   ```
   bit aflag;
   ```

2. Change all atomic clauses to:
   ```
   atomic {
   stat_1;
   aflag=1;
   stat_2
   ...
   stati
   aflag=0;
   }
   ```

3. Check that `aflag` is always 0.

   e.g.
   ```
   active process monitor {
   assert(!aflag);
   }
   ```
timeout (1)

- Promela does not have real-time features.
  - In Promela we can only specify functional behaviour.
  - Most protocols, however, use timers or a timeout mechanism to resend messages or acknowledgements.

- timeout
  - SPIN’s timeout becomes executable if there is no other process in the system which is executable
  - so, timeout models a global timeout
  - timeout provides an escape from deadlock states
  - beware of statements that are always executable…
timeout (2)

- Example to recover from message loss:

```plaintext
active proctype Receiver()
{
    bit recvbit;
    do
        :: toR ? MSG, recvbit -> toS ! ACK, recvbit;
        :: timeout -> toS ! ACK, recvbit;
    od
}
```

- Premature timeouts can be modelled by replacing the timeout by skip (which is always executable).

One might want to limit the number of premature timeouts (see [Ruys & Langerak 1997]).
**goto**

goto label

- transfers execution to label
- each Promela statement might be labelled
- quite useful in modelling communication protocols

```promela
wait_ack:
  if
    :: B?ACK -> ab=1-ab ; goto success
    :: ChunkTimeout?SHAKE ->
      if
        :: (rc < MAX) -> rc++; F!(i==1),(i==n),ab,d[i];
          goto wait_ack
        :: (rc >= MAX) -> goto error
      fi
  fi ;
```

Timeout modelled by a channel.

Part of model of BRP
unless

{ <stats> } unless { guard; <stats> }

- Statements in <stats> are executed until the first statement (guard) in the escape sequence becomes executable.
- resembles exception handling in languages like Java
- Example:

```plaintext
proctype MicroProcessor() {
    {
        ...
        /* execute normal instructions */
    }
    unless { port ? INTERRUPT; ... }
}
```
macros - **cpp** preprocessor

- Promela uses **cpp**, the C preprocessor to preprocess Promela models. This is useful to define:
  
  - **constants**
    
    ```
    #define MAX 4
    ```
    
  - **macros**
    
    ```
    #define RESET_ARRAY(a) 
    d_step { a[0]=0; a[1]=0; a[2]=0; a[3]=0; }
    ```
    
  - **conditional** Promela model fragments
    
    ```
    #define LOSSY 1
    ...
    ifndef LOSSY
    active proctype Daemon() { /* steal messages */ }
    endif
    ```
inline - poor man's procedures

- Promela also has its own macro-expansion feature using the inline-construct.

```promela
inline init_array(a) {
    d_step {
        i=0;
        do
            :: i<N -> a[i] = 0; i++
            :: else -> break
        od;
        i=0;
    }
}
```

- error messages are more useful than when using #define
- cannot be used as expression
- all variables should be declared somewhere else
(random) Simulation Algorithm

while (!error & !allBlocked) {
    ActionList menu = getCurrentExecutableActions();
    allBlocked = (menu.size() == 0);
    if (!allBlocked) {
        Action act = menu.chooseRandom();
        error = act.execute();
    }
}

act is executed and the system enters a new state

deadlock \equiv allBlocked

interactive simulation: act is chosen by the user

Visit all processes and collect all executable actions.
Verification Algorithm

- **SPIN** uses a depth first search algorithm (DFS) to generate and explore the complete state space.

  ```java
  procedure dfs(s: state) {
    if error(s)
      reportError();
    foreach (successor t of s) {
      if (t not in Statespace)
        dfs(t);
    }
  }
  ```

  States are stored in a hash table.

  Only works for state properties.

  Requires state matching.

  The old states s are stored on a stack, which corresponds with a complete execution path.

- Note that the construction and error checking happens at the same time: SPIN is an on-the-fly model checker.
Properties (1)

• Model checking tools **automatically** verify whether
  \[ M \models \phi \]
  holds, where \( M \) is a (finite-state) **model** of a system and
  **property** \( \phi \) is stated in some formal notation.

• With SPIN one may **check** the following type of properties:
  – deadlocks (invalid endstates)
  – assertions
  – unreachable code
  – LTL formulae
  – **liveness** properties
    • non-progress cycles (livelocks)
    • acceptance cycles
In Spin a subset – $\text{LTL}_{-X}$
- LTL without X operator
- More efficient model checking algorithm
- Still expressive enough

Describing properties of states (or runs), not of transitions between states
Three examples with simple acknowledgment

First example with “perfect lines”
```c
#define MAX 4;
mtype {MSG, ACK};
chan toR = [1] of {mtype, byte, bit};
chan toS = [1] of {mtype, bit};

active proctype Sender()
{
    byte data;
    bit sendb, recvb;
    sendb = 0;
    data = 0;
    do
        :: toR ! MSG(data,sendb) -> toS ? ACK(recvb);
        if
            :: recvb == sendb -> sendb = 1-sendb;
            data = (data+1)%MAX;
            :: else -> skip; /* resend old data */
        fi
    od
}

active proctype Receiver()
{
    byte data, exp_data;
    bit ab, exp_ab;
    exp_ab = 0;
    exp_data = 0;
    do
        :: toR ? MSG(data,ab) ->
            if
                :: (ab == exp_ab) -> assert(data == exp_data);
                exp_ab = 1-exp_ab;
                exp_data = (exp_data+1)%MAX;
                :: else -> skip;
            fi;
        toS ! ACK(ab)
    od
}
Alternating Bit Protocol

• Three examples with simple acknowledgment
  ▪ **Second example** with a stealing daemon modeling lossy channels – the protocol does not work well
Adding a special stealing daemon process:

```proctype
active Daemon()
{
  do
    :: toR ? _, _, _
    :: toS ? _, _
  od
}
```

Q: What happens now?

A: Deadlock!
Alternating Bit Protocol

- Three examples with simple acknowledgment
  - Third example – redemption
Fixing the sender:

```plaintext
do
do
toR ! MSG(data,sendb) ->
  if
toS ? ACK(recv) ->
    if
      recv == sendb -> sendb = 1-sendb;
      data = (data+1)%MAX;
    else /* resend old data */
      fi
    timeout /* message lost */
  
  od
```

Q: What happens now?
A: No error found.
But no data transmitted!
Alternating Bit Protocol

- Three examples with simple acknowledgment
  - Fourth example – does receiver really get data?
Augmenting the receiver:

```plaintext
do
  :: toR ? MSG(data,ab) ->
  if
    :: (ab == exp_ab) -> assert(data == exp_data);
    exp_ab = 1-exp_ab;
    progress: exp_data = (exp_data+1)%MAX;
  :: else -> skip;
  fi;
  toS ! ACK(ab)
od
```

Checking for progress.
The error found.
We should be aware of all possible executions and issues in the model

Model is not implementation!

If there is error due to simplification (abstraction), it can still be ok

- In ABP, for example, we may know that messages can get lost but usually are delivered
- Consider possible errors beyond the ignored one!
Information on Spin

• The homepage: www.spinroot.com
• Tutorials: