Behavior models and verification

Lecture 3

Jan Kofroň, František Plášil
Model checker: Spin

Markov chains
Timed automata
Labelled transition system
Kripke structure

Model

Property specification

AG(start → AF heat)

Model checker

Property satisfied
Property violated

Jan Kofroň, František Plášil, Lecture 3
SPIN - Introduction

- **SPIN** (= Simple Promela Interpreter)
  - is a tool for analysing the logical consisteny 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 C
  - specification language to model finite-state systems
Promela Model

• **Promela model** consist of:
  – type declarations
  – channel declarations
  – variable declarations
  – process declarations
  – [init 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

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

proctype Sender() {
  ...
}
  \[ process body \]

proctype Receiver() {
  ...
}

init {
  ...
}
  \[ creates processes \]
```
Processes (1)

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

```latex
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

- Processes 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);
}

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

- number of procs. (opt.)
- parameters will be initialised to 0
/* 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;  
- `bool` flag;  
- `byte` counter;  
- `short` s;  
- `int` msg;

**Arrays**

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

Array indexing start at 0

**Typedef (records)**

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

Variable declaration
Statements

• 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.
  
  \[
  \begin{align*}
  2 & < 3 \quad \text{always executable} \\
  x & < 27 \quad \text{only executable if value of } x \text{ is smaller than } 27 \\
  3 + x & \quad \text{executable if } x \text{ is not equal to } -3 
  \end{align*}
  \]
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;

proc 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)

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

proctype P(bit i) {
    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++;  
    mutex--;  
    x = 0;
}

active proctype B() {
    y = 1;
    x == 0;
    mutex++;  
    mutex--;  
    y = 0;
}

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

Process A waits for process B to end.

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.
if-statement (1)

```
if
  :: choice₁ → stat₁₁; stat₁₂; stat₁₃; ...
  :: choice₂ → stat₂₁; stat₂₂; stat₂₃; ...
  :: ...
  :: choiceₙ → statₙ₁; statₙ₂; statₙ₃; ...
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)

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

- 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 choosen 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)

Sender

! is sending

? is receiving

Receiver

s2r

r2s

s2r!MSG

MSG

s2r?MSG

ACK

r2s!ACK

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**:

  \[
  \text{chan } <\text{name}> = [<\text{dim}>] \text{ of } \{<t_1>, <t_2>, \ldots, <t_n>\};
  \]

  - 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**

  \[
  \begin{align*}
  \text{chan c} &= [1] \text{ of } \{\text{bit}\}; \\
  \text{chan toR} &= [2] \text{ of } \{\text{mtype, bit}\}; \\
  \text{chan line[2]} &= [1] \text{ of } \{\text{mtype, Record}\};
  \end{align*}
  \]
Communication (3)

- channel = FIFO-buffer (for dim>0)

**Sending - putting a message into a channel**

\[ \text{ch} \; ! \; \langle \text{expr}_1 \rangle, \; \langle \text{expr}_2 \rangle, \; \ldots \; \langle \text{expr}_n \rangle; \]
- The values of \( \langle \text{expr}_i \rangle \) should correspond with the types of the channel declaration.
- A send-statement is executable if the channel is not full.

**Receiving - getting a message out of a channel**

\[ \text{ch} \; ? \; \langle \text{var}_1 \rangle, \; \langle \text{var}_2 \rangle, \; \ldots \; \langle \text{var}_n \rangle; \]
- If the channel is not empty, the message is fetched from the channel and the individual parts of the message are stored into the \( \langle \text{var}_i \rangle \)s.

\[ \text{ch} \; ? \; \langle \text{const}_1 \rangle, \; \langle \text{const}_2 \rangle, \; \ldots \; \langle \text{const}_n \rangle; \]
- If the channel is not empty and the message at the front of the channel evaluates to the individual \( \langle \text{const}_i \rangle \), the statement is executable and the message is removed from the channel.
Rendez-vous communication

\[ \text{dim} \] == 0

The number of elements in the channel is now zero.

- If send \texttt{ch!} is enabled and if there is a corresponding receive \texttt{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:**

```plaintext
chan ch = [0] of {bit, byte};
- P wants to do \texttt{ch! 1, 3+7}
- Q wants to do \texttt{ch? 1, x}
- Then after the communication, \texttt{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 *enabled*
  - This is repeated
- Number of all possible interleavings may be very high
  - → state space explosion → 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
   ...
   stat_n
   aflag=0;
   }
   ```

3. Check that `aflag` is always 0.

   ```
   []!aflag
   ```

   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 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:

```c
prototype 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
    ...
    #ifdef LOSSY
    active proctype Daemon() { /* steal messages */ }
    #endif
    ```
inline - poor man's procedures

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

```c
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

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

- `deadlock = allBlocked`
- `act is chosen by the user`
- `act is executed and the system enters a new state`
- `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.

```plaintext
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.
- 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.

**Thursday 11-Apr-2002**

**Theo C. Ruys - SPIN Beginners' Tutorial**
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 – LTL_{\neg X}
- LTL without X operator
- More efficient model checking algorithm
- Still expressive enough

Describing properties of **states** (or runs), not of **transitions** between states
Alternating Bit Protocol

- Three examples with simple acknowledgment
  - First example with “perfect lines”
#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:

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

Q: What happens now?
A: Deadlock!
Three examples with simple acknowledgment

Third example – redemption
Fixing the sender:

```plaintext
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 */
      fi
    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:

\[
\text{do} \\
\text{:: toR ? MSG(data,ab) ->} \\
\text{if} \\
\text{:: (ab == exp_ab) -> assert(data == exp_data);} \\
\text{exp_ab = 1-exp_ab;} \\
\text{progress: exp_data = (exp_data+1)\%MAX;} \\
\text{:: else -> skip;} \\
\text{fi;} \\
\text{toS ! ACK(ab)} \\
\text{od}
\]
Alternating Bit Protocol – Summary

- 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: