Computer Lab 1:
Model Checking and Logic Synthesis using Spin (lab)

Richard M. Murray
Nok Wongpiromsarn    Ufuk Topcu
California Institute of Technology

AFRL, 25 April 2012

Outline

• Spin model checker: modeling concurrent systems and describing system requirements in Promela
• Model-checking with Spin
• Logic synthesis with Spin

The Spin Model Checker
Gerard J. Holzmann
Addison-Wesley, 2003
http://spinroot.com
The process flow of model checking

Efficient model checking tools automate the process: SPIN, nuSMV, TLC,...
Spin Verification Models

System design (behavior specification)
• *Promela* (Process Meta Language) is a non-deterministic, guarded command language for specifying possible system behavior of a distributed system in Spin
• There are 3 types of objects in Spin verification model
  - asynchronous processes
  - global and local data objects
  - message channels

System requirements (correctness claims)
• default properties
  - absence of system deadlock
  - absence of unreachable code
• assertions
• end-state labels
• acceptance
• progress
• fairness
• never claim
• LTL formulas
• trace assertions

\[
\Phi = \square g_1 \land \square g_2
\]
Running Spin

Typical sequence of commands

$ spin -u100 model # non-verbose simulation for 100 steps
$ spin -a model # generate C code for analysis (pan.c)
$ gcc -o pan pan.c # generate executable verifier from pan.c
$ ./pan -a -N P1 # perform verification of specification P1
$ spin -t -p model # show error trail

Note: spin -- and ./pan -- list available command-line and un-time options, resp.
• A keyword `proctype` is used to declare process behavior
• 2 ways to instantiate a process
  - Add the prefix `active` to a `proctype` declaration. The process will be instantiated in the initial system state.
  - Use `run` operator to instantiate a process in any reachable system state

Extra process `init` needs to be created
Promela Objects: Data Objects

- 2 levels of scope: global and process local
- No intermediate levels of scope
- The default initial value of all data objects is zero
- All objects must be declared before they can first be referenced
- User-defined type can be declared using keyword `typedef`

<table>
<thead>
<tr>
<th>Type</th>
<th>Typical Range</th>
<th>Sample Declaration</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit</td>
<td>0, 1</td>
<td>bit turn = 1</td>
</tr>
<tr>
<td>bool</td>
<td><code>false, true</code></td>
<td>bool flag = true</td>
</tr>
<tr>
<td>byte</td>
<td>0...255</td>
<td>byte a[12]</td>
</tr>
<tr>
<td>chan</td>
<td>1...255</td>
<td>chan m</td>
</tr>
<tr>
<td>mtype</td>
<td>1...255</td>
<td>mtype n</td>
</tr>
<tr>
<td>pid</td>
<td>0...255</td>
<td>pid p</td>
</tr>
<tr>
<td>short</td>
<td>$-2^{15} ... 2^{15} - 1$</td>
<td>short b[4] = 89</td>
</tr>
<tr>
<td>int</td>
<td>$-2^{31} ... 2^{31} - 1$</td>
<td>int cnt = 67</td>
</tr>
<tr>
<td>unsigned</td>
<td>0...$2^n - 1$</td>
<td>unsigned w : 3 = 5</td>
</tr>
</tbody>
</table>

All elements initialized to 0

All elements initialized to 89

Unsigned stored in 3 bits (range 0...7)
Basic Statements

• Assignment
  - valid assignment: \( c++ \), \( c-- \), \( c = c+1 \), \( c = c-1 \)
  - invalid assignment: \( ++c \), \( --c \)

• Expressions
  - must be side effect free
  - the only exception is the run operator, which can have a side effect

• Print: \( \text{printf("x = \%d\n", x)} \)

• Assertion: \( \text{assert}(x+y == z), \text{assert}(x <= y) \)
  - always executable and has no effect on the state of the system when executed
  - can be used to check safety property: Spin reports an error if the expression can evaluate to zero (false)

```c
int n;
active proctype invariant() {
    assert(n <= 3)
}
```

The assertion statement can be executed at any time. This can be used to check a system invariant condition: it should hold no matter when the assertion is checked.

• send
• receive } message passing between processes (won’t cover)
A statement in a Spin model is either executable or blocked

- A statement is executable iff it evaluates to true or non-zero integer value

  - print statements and assignments are always unconditionally executable
  - If a process reaches a point where there is no executable statements left to execute, it simply blocks

<table>
<thead>
<tr>
<th>Expression</th>
<th>Executability</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 &lt; 3</td>
<td>always executable</td>
</tr>
<tr>
<td>x &lt; 27</td>
<td>executable iff x &lt; 27</td>
</tr>
<tr>
<td>3 + x</td>
<td>executable iff x ≠ 3</td>
</tr>
</tbody>
</table>

while (a != b)
{
   skip;
}

\[ \text{do} \]
:: (a == b) -> break
:: else -> skip
\[ \text{od} \]

\[ a == b; \]

block until a==b

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

2 levels of nondeterminism

- System level: processes execute concurrently and asynchronously
  - Process scheduling decisions are non-deterministic
  - Statement executions from different processes are arbitrarily interleaved in time
    - Basic statements execute atomically
- Process level: local choice within processes can also be non-deterministic

```plaintext
byte x = 2, y = 2;
active proctype A() {
    do
        :: x = 3-x
        :: y = 3-y
    od
}
active proctype B() {
    do
        :: x = 3-y
        :: y = 3-x
    od
}
```

At any point in an execution, any of these statements can be executed.
Control Flow

- Semicolons, gotos and labels
- Atomic sequences: `atomic{ ... }`
  - Define an indivisible sequence of actions
  - No other process can execute statements from the moment that the first statement of this sequence begins to execute until the last one has completed
- Deterministic steps: `d_step{ ... }`
  - Similar to atomic sequence but more restrictive, e.g., no nondeterminism, goto jumps, or unexecutable statements is allowed
- Nondeterministic selection:
  ```
  if
  :: guard1 -> stmt11; stmt12; ...
  :: guard2 -> stmt21; stmt22; ...
  :: ...
  fi
  ```
- Nondeterministic repetition:
  ```
  do
  :: guard1 -> stmt11; stmt12; ...
  :: guard2 -> stmt21; stmt22; ...
  :: ...
  do
  ```
- Escape sequences: `{ P } unless { E }`
- Inline definitions: `inline{ ... }`

```
atomic {
  tmp = b;
  b = a;
  a = tmp
}
d_step {
  tmp = b;
  b = a;
  a = tmp
}
```

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

```
do
:: x++
:: x--
:: break
od
```

```
swap the values of a and b
```

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

```
without the else clause, the if-statement would block until other guards becomes true.
```

```
transfers control to the end of the loop
```
Nondeterministic Selection and Repetition

• If at least one guard is executable, the if/do statement is executable
• If more than one guard is executable, one is selected non-deterministically
• If none of the guard statements is executable, the if/do statement blocks
• Any type of basic or compound statement can be used as a guard
• ‘if’ statement checks once and continues; ‘do’ statement re-executes code until a break is reached
Defining Correctness Claims

- **default properties**
  - absence of system deadlock
  - absence of unreachable code
- **assertions**
  - local process assertions
  - system invariants
- **end-state labels**
  - define proper termination points of processes

- **accept-state labels**
  - when looking for acceptance cycles
- **progress-state labels**
  - when looking for non-progress cycles
- **fairness**
- **never claims**
- **LTL formulas**
- **trace assertions**

**safety**
- “nothing bad ever happens”
- properties of reachable states

**liveness**
- “something good eventually happens”
- properties of infinite sequences of states
Progress and Acceptance

Progress
• Search for reachable non-progress cycles (infinite executions that do not pass through any progress state)
• Progress states are specified using progress label
• Enforced by gcc -DNP and pan -l

Acceptance
• Search for acceptance cycles (infinite executions that do pass through a specially marked state)
• Acceptance states are specified using accept label
• Enforced by pan -a

```c
byte x = 2, y = 2;
active proctype A()
{
    do
        :: x = 3-x
        :: y = 3-y; progress: skip
    od
}
```

A non-progress cycle is an infinite execution sequence that does not pass through any progress state.
Fairness

Weak fairness

- If a statement is executable infinitely long, it will eventually be executed
- Process-level weak-fairness can be enforced by run-time option `pan -f`
  - if a process contains at least one statement that remains executable infinitely long, that process will eventually execute a step
  - does not apply to non-deterministic transition choices within a process

Strong fairness

- If a statement is executable infinitely often, it will eventually be executed

Enforcing fairness increases the cost of verification

- **Weak fairness**: complexity is linear in the number of active processes
- **Strong fairness**: complexity is quadratic in the number of active processes
Never Claims

Define an observer process that executes synchronously with the system

- Intended to monitor system behavior; do not contribute to system behavior
- Can be either deterministic or non-deterministic
- Contain only side-effect free expressions
- Abort when they block
- Reports a violation when
  - closing curly brace of never claim is reached
  - an acceptance cycle is found (infinite execution passing through accept label)

Typically used to enforce LTL property

- Old style: `spin -f '!spec'` generates never claim
- New style: use `ltl label { spec }`
- Make sure to run `pan -a` when you have never claims

Example: `[]<>g1`

- To make sure this is always true, need to make sure that `!spec` is never true (same inversion as usual)
LTL Formulas

- Never-claims can define all omega-regular word-automata
  - Properties that can be stated in LTL can be converted mechanically into never claims
- Negate the LTL formula, and generate the claim from the negated form
- All accepting runs of the resulting omega-automaton correspond to violations of the original (non-negated) property

```bash
$ spin -f '!<>[]p'
never { /* !<>[]p */
T0_init:
  if
    :: (! ((p))) -> goto accept_S9
    :: (1) -> goto T0_init
  fi;
accept_S9:
  if
    :: (1) -> goto T0_init
  fi;
}
```

```bash
$ ltl2ba -f '!<>[]p'
never { /* !<>[]p */
T0_init:
  if
    :: (!p) -> goto accept_S1
    :: (1) -> goto T0_init
  fi;
accept_S1:
  if
    :: (!p) -> goto accept_S1
    :: (1) -> goto T0_init
  fi;
}
```
Spin Commands

Generate model-specific ANSI C code `pan.c`

$ spin -a model.pml

Generate verifier from `pan.c`

- Typical command
  $ gcc -o pan pan.c
- Enforcing progress
  $ gcc -DNP -o pan pan.c

Perform verification

- Typical command
  $ ./pan -a -N P1 model.pml
- Enforcing progress: add -l
- Enforcing acceptance: add -a
- Enforcing fairness: add -f

Note: `spin --` and `.pan --` list available command-line and run-time options, resp

Relay error trail

$ spin -t -p -g model.pml

- follow error trail
- print all statements
- print all global variables

Correctness proof

correctness proof
Example 1: traffic lights (property verified)

**System** $TS$: composition of two traffic lights and a controller

![Diagram of the system](image)

**Property verified:**

$TS \models P_1$

**Specification** $P_1$:

“The light are never green simultaneously.”

$\square \neg (g_1 \land g_2)$

$\neg \square (g_1 \land g_2)$

**SPIN code:**

```plaintext
active prototype $P()$
{
    do
        :: atomic{ (g1==0 & g2==0) -> g1=1; g2=0 }
        :: atomic{ (g1==0 & g2==0) -> g1=0; g2=1 }
        :: atomic{ (g1==1 & g2==0) -> g1=0; g2=0 }
        :: atomic{ (g1==0 & g2==1) -> g1=0; g2=0 }
    od
}

never { /* F(g1 & g2) */
    T0_init : /* init */
        if
            :: (1) -> goto T0_init
            :: (g1 & g2) -> goto accept_all
        fi;
    accept_all : /* 1 */
        skip
}
```

```
lit2ba -f "<\!(g1 &\! g2)" > lights_safety.pml
spin -a -N lights_safety.pml lights.pml
gcc -o pan pan.c
./pan -a
```
Example 2: traffic lights
(counterexample found $\rightarrow$ property not verified)

**System** $TS$: composition of two traffic lights and a controller

$$
\begin{align*}
q_1 &\xrightarrow{\alpha} q_2, \{g_1\} & q_2 &\xrightarrow{\beta} q_1, \{g_1\} \\
q_2 &\xrightarrow{\alpha} q_1, \{g_1\} & q_1 &\xrightarrow{\alpha} q_2, \{g_1\} \\
\end{align*}
$$

traffic light 1  traffic light 2  controller

**Property not verified:** $TS \not\models P_2$

Counterexample:

$$(\langle q_1, s_1, c_1, 1 \rangle \langle q_1, s_2, c_3, 1 \rangle)^\omega$$

**Specification** $P_2$:

“The first light is infinitely often green.”

$[]<>g_1$

$$\text{ltl2ba -f "! []<>g1" > lights_progress.pml}
\text{spin -a -N lights_progress.pml lights.pml}
\text{gcc -o pan pan.c}
\text{./pan -a}$$

$$\text{spin -a -N lights_simple.pml}
\text{gcc -o pan pan.c}
\text{./pan -a -N P2}$$
Example 3: traffic lights  
(counterexample used to modify the controller)

**System** $TS$: composition of two traffic lights and a modified controller

new controller: $\beta^\omega$ is not a valid control signal anymore

**Specification** $P_2$:  
“The first light is infinitely often green.”

Property verified:  
$TS \models P_2$
Example: traffic lights - find “good” sequence

\[ P = \quad || \quad \Phi = \quad \square \neg (g_1 \land g_2) \land \square \Diamond g_1 \land \square \Diamond g_2 \]

\[ L_\omega (A) = \text{Words}(\Phi) \]

\[ A \]

\[
\begin{align*}
\text{bool } g_1 = 0, g_2 = 0; \\
\text{active proctype } TL1() \{ \\
\quad \text{do} \quad \text{atomic} \{ g_1 == 0 \rightarrow g_1 = 1 \} \quad \text{atomic} \{ g_1 == 1 \rightarrow g_1 = 0 \} \quad \text{od} \\
\} \\
\text{active proctype } TL2() \{ \\
\quad \text{do} \quad \text{atomic} \{ g_2 == 0 \rightarrow g_2 = 1 \} \quad \text{atomic} \{ g_2 == 1 \rightarrow g_2 = 0 \} \quad \text{od} \\
\}
\]

\[
\text{never} \{ \\
\quad \text{T0.init:} \quad \text{if} \quad \text{atomic} \{ !g_1 \land !g_2 \rightarrow \text{goto T0.init} \} \\
\quad \text{fi}; \\
\quad \text{T1.S1:} \quad \text{if} \quad \text{atomic} \{ !g_1 \land !g_2 \rightarrow \text{goto T1.S1} \} \\
\quad \text{fi}; \\
\quad \text{accept.S1:} \quad \text{if} \quad \text{atomic} \{ !g_1 \land !g_2 \rightarrow \text{goto T0.init} \} \\
\quad \text{fi}; \\
\}
\]
Example: farmer puzzle

A farmer wants to cross a river in a little boat with a wolf, a goat and a cabbage.

Constraints:

- The boat is only big enough to carry the farmer plus one other animal or object.
- The wolf will eat the goat if the farmer is not present.
- The goat will eat the cabbage if the farmer is not present.

How can the farmer get all both animals and the cabbage safely across the river?

\[
\Phi = \Diamond (f = w = g = c = 1) \land \\
\Box (w \neq g \lor f = g) \land \\
\Box (g \neq c \lor f = g)
\]

\[
L_\omega (A) = \text{Words}(\Phi)
\]