Test and Verification Lecture 14

SPIN and promela

Ulrik Nyman
ulrik@cs.aau.dk
Plan for today

- Promela
  - Constructs
  - Examples
- LTL properties
- Installation
- SPIN demo
Promela

- Programming Meta Language
- A modeling language for verification and simulation
- Restricted set of constructs and datatypes
Model parts

- Processes
- Message Channels
- Variables
No difference between conditions and statements

- This might seem strange at first

Boolean conditions can be executed when they are true

Else they block until they become true

Statement are always executable
Executability

- No need for busy loops

```c
while (a != b)
    skip   /* wait for a==b */
```

- Can be replaced with

```c
(a == b)
```
Variables

- Global and local variables

```c
bool flag;
int state;
byte msg;
```

- Array variables

- Message types

```c
mtype = {ack, nack, err}
```
## Datatypes

<table>
<thead>
<tr>
<th>Typename</th>
<th>C-equivalent</th>
<th>Macro in limits.h</th>
<th>Typical Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit or bool</td>
<td>bit-field</td>
<td>-</td>
<td>0..1</td>
</tr>
<tr>
<td>byte</td>
<td>uchar</td>
<td>CHAR_BIT (width in bits)</td>
<td>0..255</td>
</tr>
<tr>
<td>short</td>
<td>short</td>
<td>SHRT_MIN..SHRT_MAX</td>
<td>-2(^{15}) - 1 .. (2^{15} - 1)</td>
</tr>
<tr>
<td>int</td>
<td>int</td>
<td>INT_MIN..INT_MAX</td>
<td>-2(^{31}) - 1 .. (2^{31} - 1)</td>
</tr>
</tbody>
</table>
One local variable

```java
proctype A()
{
    byte state;
    state = 3
}
```
; is only a separator

-> is equivalent

byte state = 2;
proctype A()
{
    (state == 1) -> state = 3
}
proctype B()
{
    state = state - 1
}
Process Instantiation

- Special init process

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

- Processes can be started from anywhere
proctype A(byte state; short foo)
{
    (state == 1) -> state = foo
}
init
{
    run A(1, 3)
}

Mutual exclusion example

#define true    1
#define false   0
#define Aturn   false
#define Bturn   true
bool x, y, t;
proctype A()
{
    x = true;
    t = Bturn;
    (y == false || t == Aturn);
    /* critical section */
    x = false
}
proctype B()
{
    y = true;
    t = Aturn;
    (x == false || t == Bturn);
    /* critical section */
    y = false
}
init
{
    run A(); run B()
}
Atomic sequences

- Runtime error if anything but the first statement blocks

```plaintext
byte state = 1;
proctype A()
{
    atomic {
        (state==1) -> state = state+1
    }
}
proctype B()
{
    atomic {
        (state==1) -> state = state-1
    }
}
init
{
    run A(); run B()
}
```
Message passing

- Used to model transfer of data
- Global or local
- Channels can send channel names

```
chan qname = [16] of { short }
chan qname = [16] of { byte, int, chan, byte }
```

- Synchronous communication

```
chan qname = [0] of { short }
```
Message passing

- **Sending**
  
  ```
  chan qname = [16] of { byte, int, chan, byte }
  qname!v,y,myChan,a
  ```

- **Receiving**
  
  ```
  qname?var,x,ch,b
  ```

- **Receiving with constants**
  
  ```
  qname?var,cons1,ch,cons2
  ```
proctype A(chan q1)
{
    chan q2;
    q1?q2;
    q2!123
}
proctype B(chan qforb)
{
    int x;
    qforb?x;
    printf("x = %d\n", x)
}
init {
    chan qname = [1] of { chan };
    chan qforb = [1] of { int };
    run A(qname);
    run B(qforb);
    qname!qforb
}
Testing for messages

- Length – built in function
  
  ```
  len(qname)
  ```

- Testing for reception
  
  ```
  qname?[var,cons1,ch,cons2]
  ```

- True if the message can be received

- Remember to use atomic
  
  ```
  (len(qname) < MAX) -> qname!msgtype
  qname?[msgtype] -> qname?msgtype
  ```
Control Flow

- Case selection

```plaintext
if
:: (a != b) -> option1
:: (a == b) -> option2
fi
```

- Guards
- Does not need to be mutually exclusive
- Keyword else
proctype counter()
{
  do
  :: (count != 0) ->
    if
    :: count = count + 1
    :: count = count - 1
    fi
  :: (count == 0) -> break
  od
}

Unconditional Jumps

```proctype Euclid(int x, y)
{
    do
    :: (x > y)  -> x = x - y
    :: (x < y)  -> y = y - x
    :: (x == y) -> goto done
    od;

    done: skip
}
```

- Extra skip at the end
proctype fact(int n; chan p)
{       chan child = [1] of { int };
    int result;
    if
        :: (n <= 1) -> p!1
        :: (n >= 2) ->
            run fact(n-1, child);
            child?result;
            p!n*result
    fi
}

init
{       chan child = [1] of { int };
    int result;
    run fact(7, child);
    child?result;
    printf("result: %d\n", result)
Timeout

- **Modeling trick**

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

- **Cannot be implemented**
Assertions

- Produces errors during simulation or verification

```python
assert(any_boolean_condition)
```
Labels

- End state labels
  - end, end1, end_here, ...

- Progress
  - progress, progress2, ...

- After having compiled
  - ./pan -l
  - Search for non progress loops
- spin -m -a ex.1a
- gcc -o pan pan.c
- ./pan
Bitstate hashing

- Coverage
- Not precise analysis
- -DBITSTATE
Propositional formulas defined separately
- Evaluated over computations
- [] Always
- <> Eventually
- U (strong) until (p U q)
- V !(!p U !q) (Also known as release)
Examples

- Nested properties
- \[ p \]
- \! ( <> !q )
- \( p U q \)
- \( p U ([] (q U r)) \)