

INF5140: Specification and Verification of Parallel Systems

Lecture 8 – LTL into Automata and Introduction to Promela

Gerardo Schneider

Department of Informatics
University of Oslo

INF5140, Spring 2006



Credits:

- Many slides (all the figures with blue background and few others) were taken from Holzmann's slides on "Logical Model Checking", course given at Caltech
(<http://spinroot.com/spin/Doc/course/index.html>)



1 Translating LTL into Automata

2 Spin and Promela

- Spin and Promela
- A First Example

3 Overview of Promela

- General Concepts
- Processes
- Data Objects
- Message Channels



Translating LTL into Automata

Preliminaries

- Rewrite the *eventually* and *always* operators
 - $\diamond\psi \equiv \top U \psi$
 - $\square\psi \equiv \perp R \psi$
- Write the formulas in negation normal form
 - $\neg\neg\psi \equiv \psi$
 - De Morgan laws for \wedge and \vee
 - $\neg(\phi U \psi) \equiv (\neg\phi) R (\neg\psi)$ ¹
 - $\neg(\phi R \psi) \equiv (\neg\phi) U (\neg\psi)$
- Make use of the following recurrence equations
 - $\phi U \psi \equiv \psi \vee (\phi \wedge \bigcirc(\phi U \psi))$
 - $\phi R \psi \equiv \psi \wedge (\phi \vee \bigcirc(\phi R \psi))$

¹The release operator R is sometimes denoted as V



Translating LTL into Automata

- Algorithm: Chapter 6, section 6.8 of Peled's book "Software Reliability Methods"
- We will show (no slides though) how the algorithm works by translating the LTL formula $\diamond p$ into a Büchi automaton



Translating LTL into Automata

Example: $\diamond p$



Translating LTL into Automata

Example: $\diamond p$

- The core algorithm gives as a result *Nodes_Set*:

Name: 1
Incoming: $\{init, 1\}$
Old: $\{\top, \top U p\}$
New: \emptyset
Next: $\{\top U p\}$

Name: 5
Incoming: $\{init, 1\}$
Old: $\{p, \top U p\}$
New: \emptyset
Next: \emptyset

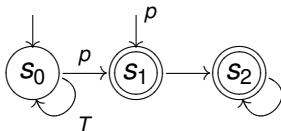
Name: 6
Incoming: $\{5, 6\}$
Old: \emptyset
New: \emptyset
Next: \emptyset



Translating LTL into Automata

Example: $\diamond p$

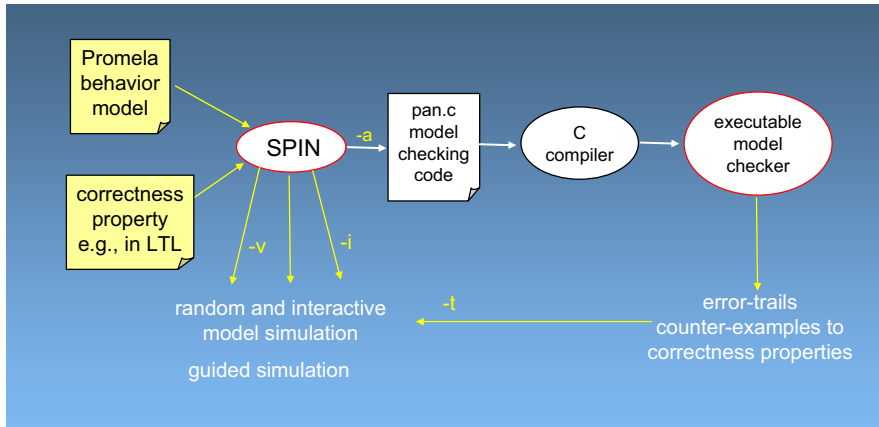
- We build the automaton as follows:
 - The set of labels L are conjunctions of propositions appearing in ψ
 - The set of states S consists of nodes in $Nodes_Set$
 - There is a transition from s to s' if $s \in Incoming(s')$
 - If p is in $Old(s)$, then p will be a label of every transition into s
 - The set of initial states s_0 are the nodes containing the incoming *init* edge
 - The generalized Büchi acceptance condition contains a separate set of states $f \in F$ for each subformula of the form $\phi U \psi$
 - f contains all the states s such that either $p \in Old(s)$ or $\top U p \notin Old(s)$



- **PROMELA** is an acronym for **PRO**cess **ME**ta **LA**nguage
 - It's a *system description language* (**not** a programming lang.)
 - Its emphasis is on modeling of process synchronization and coordination, not computation
 - Targeted to the description of software systems, rather than hardware circuits
- **SPIN** is an acronym for **S**imple **P**romela **I**nterpreter
 - It can be used as a *simulator* and as a *verifier*
 - Its basic building blocks are
 - Asynchronous processes
 - Buffered and unbuffered message channels
 - Synchronizing statements
 - Structured data
 - There are no floating points, no notion of time nor of a clock



The Tool



Producer and Consumers

```
mtype = { P, C };
mtype turn = P;

active proctype producer()
{
    do
        :: (turn == P) ->
            printf("Produce\n");
            turn = C
    od
}

active proctype consumer()
{
    do
        :: (turn == C) ->
            printf("Consume\n");
            turn = P
    od
}
```



Producers and Consumers

- P and C are two *symbolic variables* (enumerated type)
- `mtype turn = P` declares a global variable `turn`
- “producer” and “consumer” don’t identify a *process* but a *process type* (`proctype`)
- `active` means one process will be instantiated from each `proctype` declaration
- `do` is the Promela loop. Each option sequence is preceded by “::” (here there is only one, with *guard* (`turn==P`))
- The execution of a loop may be stopped with a `goto` or a `break` (not present in the example)
- If all guard conditions evaluate to *false* the process **blocks** until at least one guard becomes true
 - Good for modeling interprocess synchronization
- If more than one condition evaluate to *true* one is chosen **non-deterministically**



Producer and Consumers

We can simulate an execution of the above program (limited to 14 steps)

```
$ spin -u14 3-prodcons.pml
    Produce
      Consume
    Produce
      Consume
-----
depth-limit (-u14 steps) reached
...
```



Extending Producers and Consumers

- We extend the previous model to have 2 producers and 2 consumers
- To force a strict alternation of producers and consumers we need more structure
 - `turn` must be neither `P` nor `C`: we add `N`
 - We associate an identity for the process: `who` of type `pid`

We add, then:

```
mtype = { P, C, N };
```

```
mtype turn = P;
```

```
pid   who;
```



Extending Producers and Consumers

The producer becomes:

```
active [2] proctype producer()
{
    do
        :: request(turn, P, N) ->
            printf("P%d\n", _pid);
            assert(who == _pid);
            release(turn, C)
    od
}
```

- `request(turn, P, N)` is not a function, it's an inline definition
- Also `release(turn, C)` is an inline definition



Extending Producers and Consumers

`request (_, _, _)` may be written as follows:

```
inline request(x, y, z) {  
    atomic { x == y -> x = z; who = _pid }  
}
```

- Formal parameters have no type designation; they are only *place holders*

- If the `inline` is invoked as `request(turn, P, N)` then the following code is inserted into the model at the invocation point:

```
atomic { (turn == P) -> turn = N; who = _pid }
```

- Notice that the scope of local variables of an `inline` definition depends on the invocation point of the `inline` and it's not restricted to its body
- `atomic` means **all** steps in the sequence will complete before another process may execute (like Manna&Pnueli's "`< >`" constructor)
 - It will execute only if `(turn==P)`



Extending Producers and Consumers

After the printing there is an assertion `assert (who==_pid)`

- The assertion verifies that the value of `who` matches the last assigned process
- The Spin verifier will be able to prove whether the assertion may be satisfied at this point or not

Following the assertion, there is another `inline`, defined as follows:

```
inline release(x, y) {  
    atomic { x = y; who = 0 }  
}
```

The call `release(turn, C)` will produce the following inlined code:

```
atomic { turn = C; who = 0 }
```

Executing this will pass the control to a consumer process...



The Extended Producer and Consumers Model

```
mtype = { P, C, N };
mtype turn = P;
pid    who;

inline request(x, y, z) {
    atomic { x == y -> x = z; who = _pid }
}

inline release(x, y) {
    atomic { x = y; who = 0 }
}

active [2] proctype producer()
{
    do
    :: request(turn, P, N) ->
        printf("P%d\n", _pid);
        assert(who == _pid);
        release(turn, C)
    od
}

active [2] proctype consumer()
{
    do
    :: request(turn, C, N) ->
        printf("C%d\n", _pid);
        assert(who == _pid);
        release(turn, P)
    od
}
```



Simulating Producers and Consumers

We can simulate the model with Spin:

```
$ spin prodcons2.pml | more
  P1
    C3
  P1
    C3
  P1
    C3
  P1
    C2
...
```

- The simulation *seems* to confirm the alternation of producer and consumers
- It seems no violation of the assertion has been found



Verifying Producers and Consumers

We can invoke the Spin verifier:

```
$ spin -a prodcons2.pml # generate a verifier
$ cc -o pan pan.c       # compile the verifier
$ ./pan                 # perform the verification
(Spin Version 4.2.6 -- 27 October 2005) + Partial Order Reduction
```

```
Full statespace search for:
  never claim           - (none specified)
  assertion violations  +
  acceptance cycles    - (not selected)
  invalid end states    +
```

```
State-vector 28 byte, depth reached 5, errors: 0
  10 states, stored
   3 states, matched
  13 transitions (= stored+matched)
   0 atomic steps
```

...

- Only 10 states have been search in order to verify the model
- There are no errors
- No assertion violations



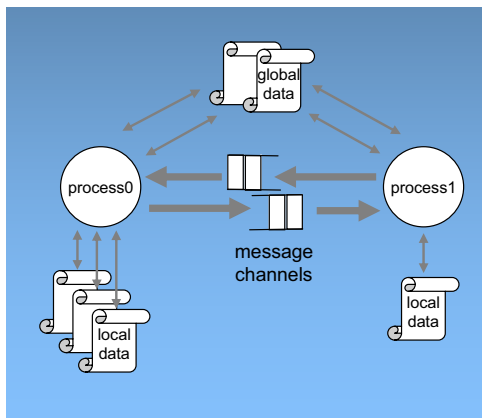
- **Finite-state models only:** Promela models are always bounded
 - Boundedness guarantees decidability
 - Finite state models can still permit infinite executions
- **Asynchronous behavior**
 - No hidden global system clock
 - No implied synchronization between processes
- **Non-deterministic control structures**
 - To support abstraction from implementation level detail
- **Executability as a core part of the semantics**
 - Every basic and compound statement is defined by a precondition and an effect
 - A statement can be executed, producing the effect, only when its precondition is satisfied; otherwise, the statement is blocked
 - Example: $q_1?m$ when channel q_1 is non-empty, retrieve message m



Central Concepts

A Promela model consists of three basic types of objects:

- Processes
- Global and local data objects
- Message channels



- Processes can synchronize their behavior in 2 ways
 - Through the use of global (shared) variables
 - Via message passing through channels
 - Buffered channels or rendezvous channels
 - There is no global “clock” that could be used for synchronization
- Each process has its own **local state**
 - Process “program-counter” (i.e., control-flow point)
 - Values of all locally declared variables
- The model as a whole has a **global state**
 - The value of all globally declared variables
 - The contents of all message channels
 - The set of all currently active processes



- **Processes** are instances of `proctype`s
- They can be instantiated in different ways
 - By prefixing a `proctype`s with the keyword `active`:
`active proctype my_process() {...}`
 - By using `run`, from any running process (see next slide)
- Multiple instances of process is possible:
`active [4] proctype my_process() {...}`
- Process types are always declared globally
- The predefined variable `_pid` denotes the process *id*
- The number of active processes is bound to 255
- Process **termination** is different from process **death**
 - A process terminates when it reaches the end of its code
 - A process can “die” and be removed only if all processes that were instantiated from it (directly or indirectly) have died
 - Processes can terminate in any order, but they die in the reverse order of their creation



Creating Processes with run

run

no 'active' prefix used in this case

init is a predefined initial process (optional)

two assignments with run expressions on the right

print statement

```
proctype irun(byte x)
{
    printf("it is me %d, %d\n", x, _pid)
}

init {
    pid a, b;
    a = run irun(1);
    b = run irun(2);
    printf("I created %d and %d\n", a, b)
}
```

two local variables, invisible outside init
x is a local variable inside irun, initialized at process initialization

'pid' and byte are data-types
'_pid' is a predefined local variable

parameter passing

```
$ spin irun.pml
    it is me 1, 1
    I created 1 and 2
        it is me 2, 2
3 processes created
$
```

interleaving of statement executions
3 asynchronous processes running.
1 of 6 possible interleavings...

init plus two copies of irun

default indentation of output
(output of process *i* gets *i+1* tab-stops)
suppressed with spin -T option

assignments and print statements are unconditionally executable
expressions are only executable when they evaluate to true

- Parameter passing is by *value*
- Parameter values cannot be passed to the `init` process nor to processes created as `active`
 - If a process created with `active` has parameters, they are treated as local variables and instantiated to zero
- A newly created process may, **but need not**, to start executing immediately after it is instantiated
- Each process defines an asynchronous thread of execution
 - It can interleave its statement executions in arbitrary way with other processes
- `run` it's an *operator*, hence `run irun(1)` is an *expression*
- The `run` expression is the only one which can have a side effect when it evaluates to non-zero (when it instantiates a process) -but not when it evaluates to zero (when it fails to instantiate a process)
 - Evaluating a `run` expression produces a value of type `pid`



There are only two levels of scope in Promela

- **Global**
 - Global to **all** processes
 - Not possible to define global variables to a subset of processes
- **Process local**
 - Local variables can be referenced from its point of declaration onwards inside the `proctype` body
 - Not possible to define local variables restricted to specific blocks of statements



Basic Data Types

| Type | Typical Range | Sample Declaration |
|----------|---------------------|--------------------|
| bit | 0..1 | bit turn = 1; |
| bool | false..true | bool flag = true; |
| byte | 0..255 | byte cnt; |
| chan | 1..255 | chan q; |
| mtype | 1..255 | mtype msg; |
| pid | 0..255 | pid p; |
| short | $-2^{15}..2^{15}-1$ | short s = 100; |
| int | $-2^{31}..2^{31}-1$ | int x = 1; |
| unsigned | $0..2^n-1$ | unsigned u : 3; |

3 bits of storage
range 0..7

the default initial value of *all* data objects (global *and* local) is zero

all variables (local and global) must be declared before they are used
a variable declaration can appear anywhere...

note: there are no reals, floats, or pointers
deliberately: verification models are meant to
model *coordination* not *computation*



Data Structures

- In Promela is possible to define record structures
- Example

```
typedef Field {
    short f = 3;
    byte  g      };

typedef Record {
    byte a[3];
    int  fld1;
    Field fld2;
    chan p[3];
    bit  b      };

proctype me(Field z) {
    z.g = 12      }

init { Record goo;
      Field foo;
      run me(foo)
    }
```



- User-defined variables are defined with the `mtype` declaration
 - They hold symbolic values which cannot match Promela reserved words
- Only one-dimensional arrays are possible
 - It is possible, however, to declare multidimensional arrays indirectly
- A user-defined type variable may be passed as argument in `run` statements, provided it does not have arrays



Message Channels

- Processes may exchange data through **message channels**
 - Asynchronously** (buffered channels)
 - Synchronously** –*rendez-vous* handshake– (unbuffered channels)
- Declaration:** `chan qname = [8] of {short, chan, record}`
 - The channel named `qname` can contain at most 8 messages, each message consist of three fields: a `short`, a `chan` (channel) and a `record` (user-defined type)
- The name of a channel can be local or global, but the channel itself is always global

```
chan x = [3] of { chan }; /* global handle, visible to both A and B */

active proctype A()
{
    chan a;          /* uninitialized local channel */

    x?a;            /* get channel id, provided by process B */
    a!x             /* and start using b's channel! */
}

active proctype B()
{
    chan b = [2] of { chan }; /* initialized local channel */

    x!b;           /* make channel b available to A */
    b?x;          /* value of x doesn't really change */
    0             /* avoid death of B, or else b disappears */
}
```



Sending and Receiving

- **Sending** a message: `qname!expr1, expr2, expr3`
 - Only executable if the channel is not full
- **Retrieving/receiving** a message: `qname?var1, var2, var3`
 - Only executable if the channel is not empty
 - If some of the parameters (e.g., `var2`) is a constant, the receiving operation is executable only if the constant parameters match the value of the corresponding fields in the message to be received
 - We can use `eval` for imposing constraints on incoming messages:
`qname?eval(var1), var2, var3`
 - `var1` is evaluated and the receive operation is executed only if the first field of the incoming messages matches the current value of `var1`
- Send and receive operations are **not** expressions; they are *i/o statements*
 - `(a>b && qname?msg0)` **not valid!**
 - `(a>b && qname?[msg0])` **valid!**
 - Expression `qname?[msg0]` is *true* when `qname?msg0` would be executed at this point (but the actual receive is not executed)

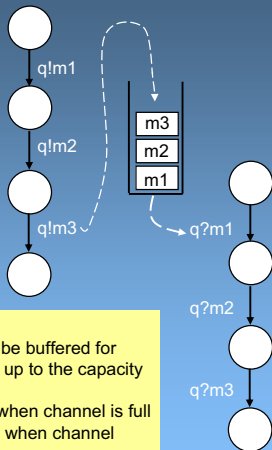


Send and Receive Variants

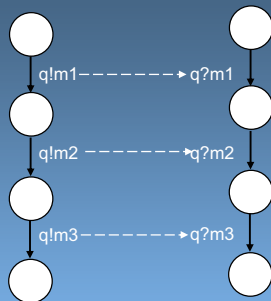
- **Sorted send:** $q! !n, m, p$
 - Like $q! n, m, p$ but adds the message n, m, p to q in numerical order (rather than in FIFO order)
- **Random receive:** $q??n, m, p$
 - Like $q?n, m, p$ but can match any message in q (it need not be the first message)
- **“Brackets”:** $q? [n, m, p]$
 - It is a side-effect free Boolean expression
 - It evaluates to true precisely when $q?n, m, p$ is executable, but has no effect on n, m, p and does not change the contents of q
- **“Braces”:** $q?n (m, p)$
 - Alternative notation for standard receive; same as $q?n, m, p$
 - Sometimes useful for separating type from arguments
- **Channel polls:** $q? <n, m, p>$
 - It is executable iff $q?n, m, p$ is executable; has the same effect on n, m, p as $q?n, m, p$, but does not change the contents of q



Asynchronous vs. synchronous



asynchronous
messages can be buffered for
later retrieval – up to the capacity
of the channel
sender blocks when channel is full
receiver blocks when channel
is empty



synchronous
channel capacity is 0
can only perform an rv handshake
not store messages
sender blocks until matching receiver
is available and vice versa



Further Reading and Final Remarks

- The LTL to Automata was taken from Chap. 6 of Peled's book
- The rest was based on Chapters 2 and 3 of Holzmann's book "The Spin Model Checker"
- For more details on where to find more on Promela syntax and Spin see "Pensum/laringskrav" at the course homepage
- Next lecture we'll continue with Chap. 3 of Holzmann's book and Chap. 7

