
The Spin Model Checker - Advanced Features

Moonzoo Kim
CS Dept. KAIST

Review: 6 Types of Basic Statements

■ Assignment: always executable

✚ Ex. `x=3+x, x=run A()`

■ Print: always executable

✚ Ex. `printf("Process %d is created.\n",_pid);`

■ Assertion: always executable

✚ Ex. `assert(x + y == z)`

■ Expression: depends on its value

✚ Ex. `x+3>0, 0, 1, 2`

✚ Ex. `skip, true`

■ Send: depends on buffer status

✚ Ex. `ch1!m` is executable only if `ch1` is not full

■ Receive: depends on buffer status

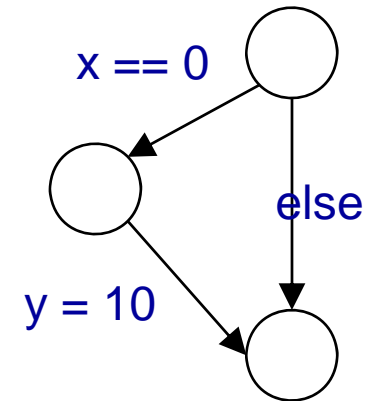
✚ Ex. `ch1?m` is executable only if `ch1` is not empty



Usages of If-statement

```
/* find the max of x and y */  
If  
:: x >= y -> m = x  
:: x <= y -> m = y  
fi
```

```
/* necessity of else */  
/* in C, if(x==0) y=10; */  
If  
:: x == 0 -> y = 10  
:: else /* i.e., x != 0 */  
fi
```



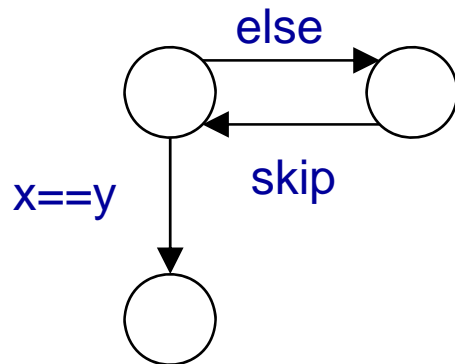
```
/* Random assignment */  
If  
:: n=0  
:: n=1  
:: n=2  
fi
```

```
/* dubious use of else with receive statement */  
If  
:: ch?msg1 -> ...  
:: ch?msg2 ->  
:: else -> ... /* use empty(ch) instead*/  
fi
```

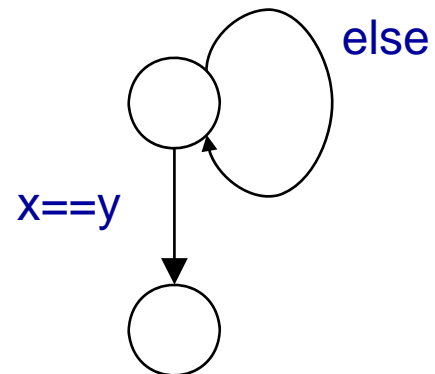


Usages of Do-statement

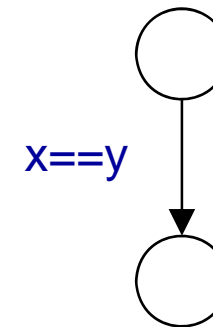
```
do
:: (x == y) -> break
:: else -> skip
od
```



```
Loop: if
:: (x == y) -> skip
:: else -> goto Loop
fi
```



$(x == y)$



Note that `break` or `goto` is **not** a statement, but control-flow modifiers



More Usages of Various Operators

■ More operators

- ✚ The standard C preprocessors can be used
 - #define, #if, #ifdef, #include
- ✚ To overcome limitation of lack of functions
 - #define add(a,b,c) c = a + b
 - inline add(a,b,c) { c = a + b }
 - Note that these two facilities still do not return a value
- ✚ Build multi-dimension array
 - typedef array {byte y[3];}
array x[2];
x[2].y[1] = 10;
- ✚ (cond -> v1: v2) is used as (cond? v1: v2) in C



More Usages of Various Operators

■ Predefined variable

- ✦ **else**: true iff no statement in the current process is executable
- ✦ **timeout** : 1 iff no statement in the model is executable
- ✦ **_**: a scratch variable
- ✦ **_pid**: an ID of current process
- ✦ **_nr_pr**: a total # of active processes
- ✦ **_last**: an ID of the process executed at previous step
- ✦ **STDIN**: a predefined channel used for simulation
- ✦ Remote reference
 - **name[pid]@label_name**
 - name: proctype name
 - **name[pid]:var_name**



■ atomic { g1; s1;s2;s3;s4}

- ✚ A sequence of statements g1;s1;s2;s3;s4 is executed without interleaving with other processes
- ✚ Executable if the guard statement (g1) is executable
 - g1 can be other statement than expression

■ If any statement other than the guard blocks, atomicity is **lost**.

- ✚ Atomicity can be **regained** when the statement becomes executable



- `d_step { g1; s1; s2;s3}`
 - ✦ `g1, s1, s2, and s3` must be deterministic (non-determinism is not allowed)
 - ✦ `g1, s1, s2, and s3` must not be blocked
- Used to perform intermediate computations as a single indivisible step
 - ✦ If non-determinism is present inside of `d_step`, it is resolved in a fixed and deterministic way
 - For instance, by always selecting the first true guard in every selection and repetition structure
 - ✦ Ex. Sorting, or mathematical computation
- Goto-jumps into and out of `d_step` sequences are forbidden

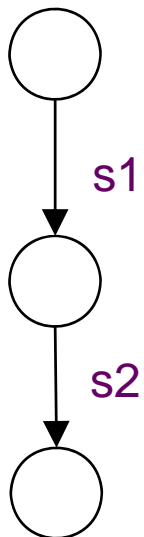


- Atomic and d_step are often used in order to reduce the size of a target model
- Both sequences are executable only when the **guard statement** is executable
 - ✦ **atomic**: if any other statement blocks, atomicity is lost at that point; it can be regained once the statement becomes executable later
 - ✦ **d_step**: it is an error if any statement other than the guard statement blocks
- Other differences:
 - ✦ **d_step**: the entire sequence is executed as **one single transition**.
 - ✦ **atomic**: the sequence is executed **step-by-step**, but without interleaving, it can make non-deterministic choices
- Caution:
 - ✦ infinite loops inside atomic or d_step sequences *are not* detected
 - ✦ the execution of this type of sequence models an indivisible step, which means that it cannot be infinite

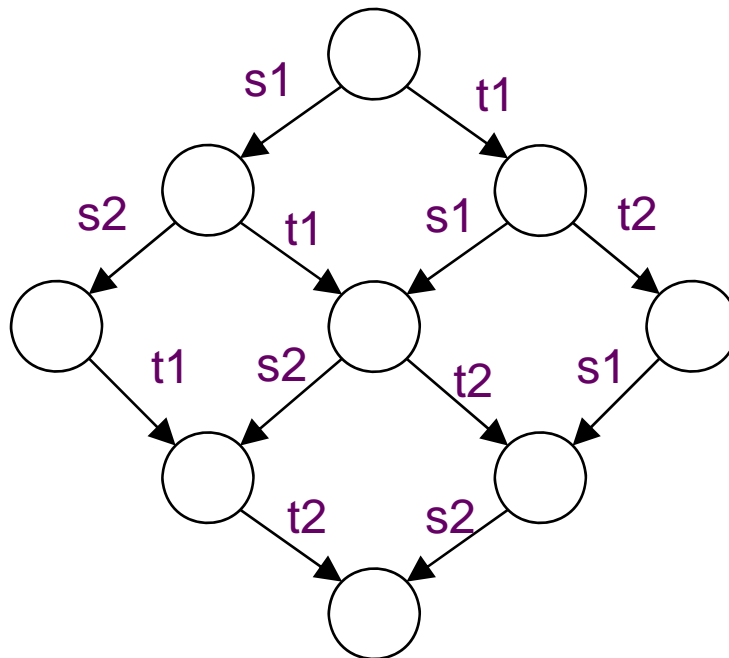
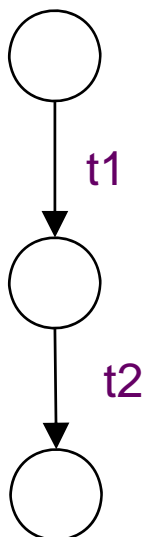


Examples: atomic v.s. d_step

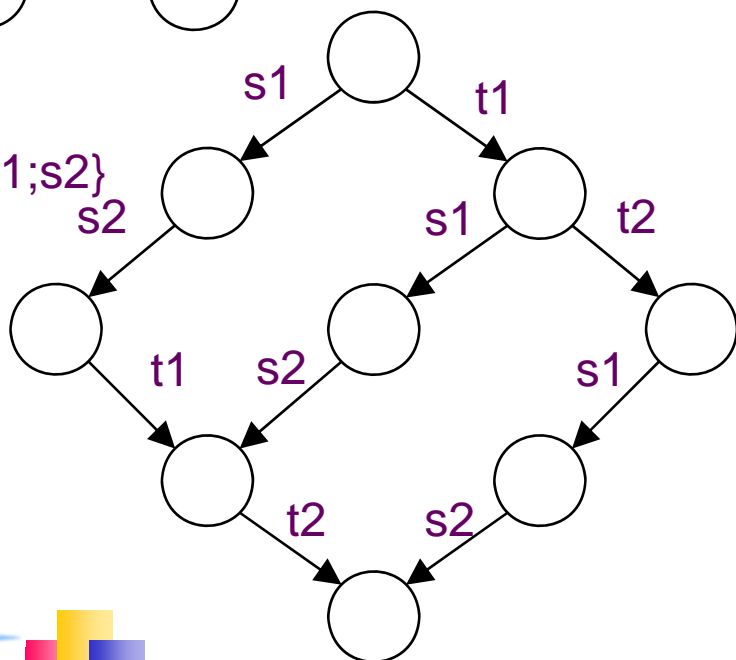
A



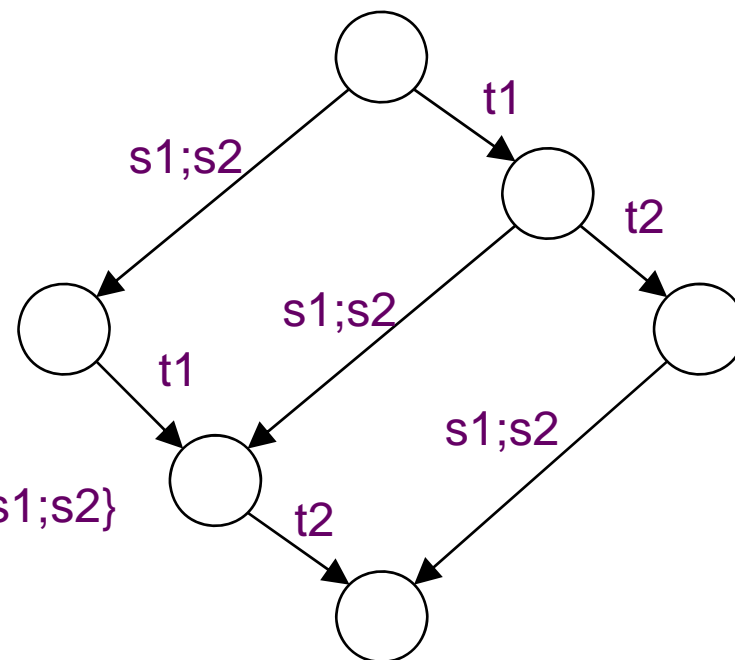
B



$\text{atomic}\{s1;s2\}$



$\text{d_step}\{s1;s2\}$



Rendezvous Comm. within atomic Sequences

- A sender performs a sending operation and a receiver performs a receiving operation **at the same time** for rendezvous communication
- If a sender has `ch!msg` in the atomic clause, after the rendezvous handshake, the sender **loses** its atomicity
- If a receiver has `ch?msg` in the atomic clause, after the rendezvous handshake, the receiver **continues** its atomicity
- Therefore, if both operations are in atomic clauses, atomicity moves from a sender to a receiver in a rendezvous handshake



- {guard1; <stmts1>} unless {guard2; <stmts2>}
 - ✦ To provide exception handling, or preemption capability
- The unless statement is executable if either
 - ✦ the guard statement of the main sequence is executable, or
 - ✦ the guard statement of the escape sequence is executable
- <stmts1> can be executed until guard2 becomes true. If then, <stmts2> becomes executable and <stmts1> is not executable anymore
 - ✦ Unless clause (<stmts2>) **preempts** a main clause (<stmts1>) if guard2 is executable, i.e., main clause is stopped.
 - ✦ Once unless clause becomes executable, no return to the main clause
- Resembles exception handling in languages like Java and ML



- Spin versions 4.0 and later support the inclusion of embedded C code into Promela model
 - ✦ `c_expr` : a user defined boolean guard
 - ✦ `c_code` : a user defined C statement
 - ✦ `c_decl` : declares data types
 - ✦ `c_state`: declares data objects
 - ✦ `c_track`: to guide the verifier whether it should track the value of data object or not
- Embedded C codes are trusted blindly and copied through from the text of the model into the code of `pan.c`



Example 1

```
c_decl {typedef struct Coord {int x, y;} Coord;}
c_state "Coord pt" "Global" /* goes inside state vector */
int z = 3; /* standard global declaration */
active proctype example() {
    c_code { now.pt.x = now.pt.y = 0;};
    do
        :: c_expr {now.pt.x == now.pt.y } ->
            c_code {now.pt.y++}
        :: else -> break
    od;
    c_code {
        printf("values %d:%d,%d,%d\n",
            Pexample-> _pid, now.z, now.pt.x, now.pt.y); };
    assert(false);
}
```



Communication between Embedded C and Promela

- `c_state` primitive introduces a new global data object `pt` of type `Coord` into the state vector
 - ✦ The object is initialized to zero according to the convention of Promela
- A global data object in a Promela model can be accessed through `now.<var>` in embedded C codes
- A local data object in a Promela model can be accessed through `P<procname>-><var>`



Example 2

```
c_decl {typedef struct Coord {int x, y;} Coord;}
c_code {Coord pt;} /* Embedded declaration goes inside
state vector */
int z = 3; /* standard global declaration */
active proctype example() {
    c_code { now.pt.x = now.pt.y = 0;};
    do
        :: c_expr {now.pt.x == now.pt.y } ->
            c_code {now.pt.y++;}
        :: else -> break
    od;
    c_code {
        printf("values %d:%d,%d,%d\n",
            Pexample-> _pid, now.z, now.pt.x, now.pt.y); };
    assert(false);
}
```



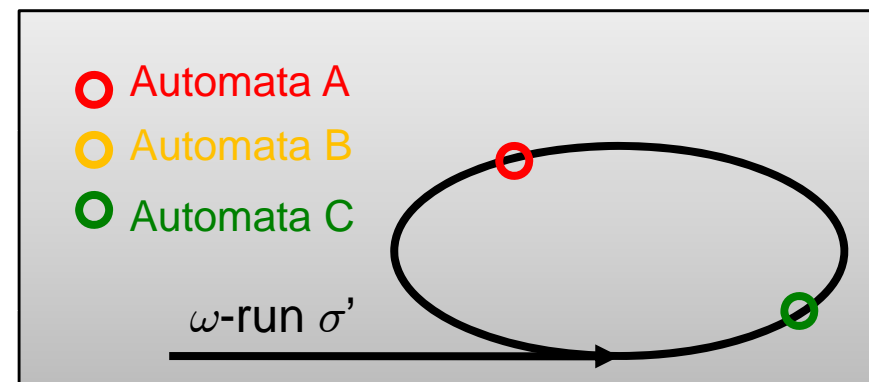
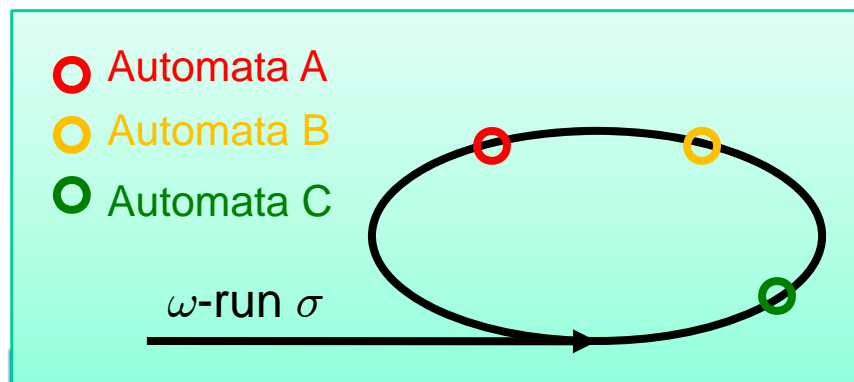
Weak Fairness v.s. Strong Fairness

Strong fairness

- ⊕ An ω -run σ satisfies the **strong fairness** requirement if it contains infinitely many transitions from **every** component automaton that is enabled **infinitely often** in σ
 - FAIRNESS running in NuSMV

Weak fairness

- ⊕ An ω -run σ satisfies the **weak fairness** requirement if it contains infinitely many transitions from **every** component automaton that is enabled **infinitely long** in σ



Examples

```
byte x;
active proctype A() {
do
:: x=2;
:: x=3;
od;}
/* [] <> x==2
F: no fairness
F: weak fairness */
```

```
byte x;
active proctype A() {
do
:: x=2;
od;}
active proctype B() {
do
:: atomic{x==2 -> x=1;}
od;}
/* [] <> (x==1)
F: no fairness
T: weak fairness, thus T
with strong fairness */
```

```
byte x;
active proctype A() {
do
:: x=2;
:: x=3;
od;}

active proctype B() {
do
:: atomic{x==2 -> x=1;}
od;}

/* [] <> (x==1)
F: if weak fairness is
applied
*/
```

