(j)Spin Cheat Sheet

Ver. 1

Comments

Everything between /* and */ is ignored (C-style comments).

```
/* This is a comment */
proctype P()
{
}
```

Data types

Type	Values	Size (bits)
bit, bool	0, 1 , **false, true**	1
byte	[0, 255]	8
short	[-32768, 32767]	16
int	$[-2^{31}, 2^{31}-1]$	32
unsigned	$[0, 2^{n}-1]$	≤ 32
chan	Reference to a channel	
pid	[0, 255]	8
mtype	Symbolic names (max. 255)	8

bool is the same as **bit**, and **true/false** is the same as 1/0.

Predefined variables

Variable	Meaning
_nr_pr	Number of running processes
_pid	Process ID of current process

```
proctype P()
{
  printf("This process has pid = %d\n",_pid)
}

init {
  run P();
  /* Wait for P to terminate */
  (_nr_pr == 1);
  printf("Init is now last process running\n")
}
```

Operators

Precedence	Operator	Associativity	Description
14	**()**	left	Parenthesis
14	**[]**	left	Array indexing
14	**.**	left	Field selection
13	**!**	right	Logical negation
13	**~**	right	Bitwise complementation
13	**++, --**	right	Increment, decrement
12	***, /, %**	left	Multiply, divide, modulo
11	**+, -**	left	Addition, subtraction
10	**<<, >>**	left	Left/right bitwise shift
9	**<, <=, >, >=**	left	Arithmetic relational operators
8	**==, !=**	left	Equality, inequality
7	**&**	left	Bitwise AND
6	**^**	left	Bitwise exclusive OR
5	**|**	left	Bitwise inclusive OR
4	**&&**	left	Logical AND
3	**||**	left	Logical OR
2	**(-> :)**	right	Conditional expression
1	**=**	right	Assignment

Symbolic names

```
/* Preprocessor macro */
#define N 10

/* Enumerated names */
mtype = {red, blue, green};
mtype = {yellow, orange}; /* Merged, note 1 */
mtype light1 = green;
mtype light2 = yellow;
```

A maximum of 255 symbolic names can be defined.
Note 1: Multiple definitions are merged. There can only be one set of mtype-names.

Statements and flow control

Expressions, statements and guards
Expressions are statements. Some statements (such as assignments, printf) are always executable. Some statements (logical expressions) are only executable when they are true and are known as *guards*.

Sequence
Semicolon '**;**' is the separator between statements executed in sequence.

```
x = 17;        /* Always executable        */
x + y > 20;    /* Guard: blocks until true  */
printf("hi"); /* Not executed until x+y>20 */
```

Selection - if

```
if
:: x == 20 -> printf("large") /* Note 1 */
:: x == 10 ;  printf("small")
:: else ->    printf("???")   /* Note 2 */
fi
```

Note 1: '**->**' is just syntactic sugar for '**;**', used to emphasize the causal relation (if->then).
Note 2: The else guard is only executable when *all* other guards are *false*.

Selection - select
To choose a value non-deterministically an if-statement can be used:

```
int x;
if  /* x = value between 1..3 */
:: x = 1
:: x = 2
:: x = 3
fi
```

The *select* statement has the same effect:

```
int x;
select(x : 1..3); /* x = value between 1..3 */
```

Repetition - do
The do-loop repeats until a *break* or *goto* statement is used break out of it.

```
do
:: x == 20 -> printf("large") /* Note 1 */
:: x == 10 ;  printf("small)
:: else ->                    /* Note 2 */
     break                    /* Note 3 */
od
```

Note 1: '**->**' is just syntactic sugar for '**;**', used to emphasize the causal relation (if->then).
Note 2: The else guard is only executable when *all* other guards are *false*.
Note 3: The *break* statement causes the loop to terminate.

Aarhus School of Engineering

Repetition – for

```
byte i;
for (i : 1..10) {
  /* Body of loop */
}
```

The bounds can be expressions:

```
for (i : (a*2)..(n+4)) {
}
```

Jump – goto

The *goto*-statement causes control to jump to a *label*. *Goto* can be used instead of *break* in a loop:

```
 do
 :: i > n -> goto exitloop
 :: else -> . . .
 od;
exitloop:
 printf(". . .");
```

Processes

All work in Spin is done in processes. A process "type" must be defined before it can run.

```
proctype p1()
{
  /* Process body */
}
```

A process type may take arguments.

```
proctype p2(byte id; byte num)
{
  printf("id = %d, num = %d",id,num);
}
```

To run the processes above, use:

```
run p1();    /* p1 takes no arguments  */
run p2(1,7); /* p2 takes two arguments */
```

If the declaration is preceded by *active*, a running instance of the process is created automatically.

```
active proctype p3()
{
  . . .
}
```

Several instances may be created in one go (3 in this example).

```
active [3] proctype p4()
{
  . . .
}
```

If a process called *init* is defined, it will run automatically. It can then start other processes as needed.

```
init {
  atomic {  /* Note 1 */
    run p1();
    run p2(2,5);
    run p3();
  }
}
```

Note 1: By convention run statements are enclosed in *atomic* to ensure that all processes have been instantiated before any of them begins execution.

Verification

Use *assert*- statements at selected points in the code to verify correct model behaviour/state:

```
assert(x >= 4 && x <= 10);
```

Synchronisation

Atomic

When statements are enclosed in *atomic*, they are executed until completion, without interference from other processes. The first statement may be a guard.

```
atomic {
  !ready;        /* Atomic sequence will block */
  temp = n + 1; /* until !ready becomes true, */
  n = temp      /* but will then run to com-  */
}               /* pletion w.o. interference. */
```

d_step

d_step (for deterministic step) can also be used.

```
d_step {
  !ready;
  temp = n + 1;
  n = temp
}
```

d_step is more efficient than *atomic*, but is subject to 3 limitations:

- Except for the first statement (the guard), statements may not block.
- It is illegal to move in or out of the sequence with *goto* or *break*.
- Non-determinism is always resolved by choosing the *first* *true* alternative (no real non-determinism).

LTL

A Spin LTL formula implicitly refers to *all* computations of the model. So if a correctness property is specified as an LTL formula, the property only holds if it is *true* in all computations – so Spin only needs to provide a single counterexample to disprove the property

Operators

Operator	*Math*	*Spin*
not	$\neg$	!
and	$\wedge$	&&
or	$\vee$	||
implies	$\Rightarrow$	->
equivalent	$\Leftrightarrow$, $\equiv$	<->
always	$\Box$	[]
eventually	$\Diamond$	<>
until	$\mathcal{U}$	U

Duality: $\neg\Box p \equiv \Diamond\neg p$, $\neg\Diamond p \equiv \neg\Box p$

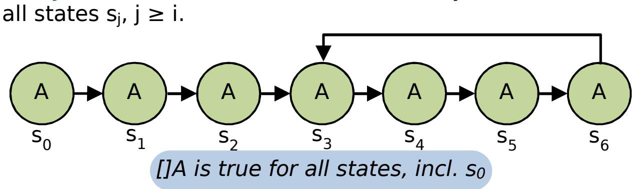
If *good* and *bad* are atomic propositions such that *good* is equivalent to *!bad*, then we have the following equivalences:

$$\neg\Box good \equiv \Diamond\neg good \equiv \Diamond\neg\neg bad \equiv \Diamond bad,$$
$$\neg\Diamond good \equiv \Box\neg good \equiv \Box\neg\neg bad \equiv \Box bad$$

Safety properties

A counterexample consists of one state where the formula is false. Choose "Safety" in jSpin drop-down menu.

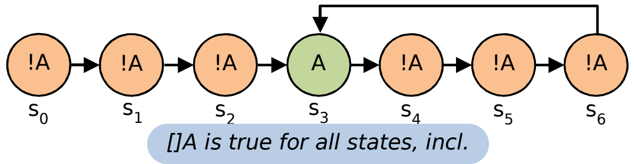
Always A, ***[] A***, is true in state s_i if and only if A is true for all states s_j, $j \geq i$.



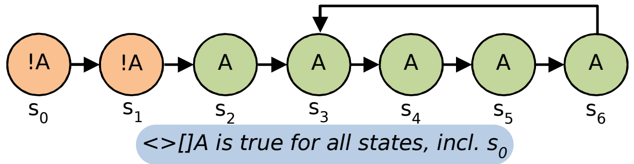
Liveness properties

A counterexample is an *infinite* computation where the formula never becomes *true*. Use "Acceptance" in jSpin dropdown menu (and tick of "Weak fairness").

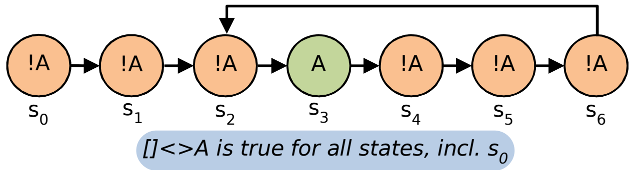
Eventually A, ***<> A***, is true in state s_i if and only if A is true for some state s_j , j ≥ i.



Latching: <>[]A (eventually always)



Indefinitely often: []<>A (always eventually)

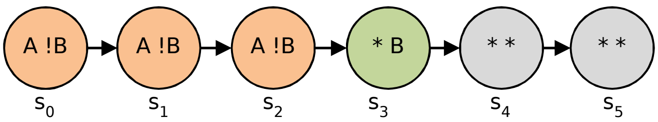


Precedence

The [] and <> operators are unary and cannot express properties relating two points in time, such as "A must become true before B becomes true". The binary operator U (*strong until*) must be used for such purposes.

A until B, ***(A) U (B)***, is true in state s_i if and only if:

- B is true in some state s_k, k ≥ i
- A is true for all states s_j, i ≤ j < k



* = don't care

Data and Program Structures

Arrays

```
int a[5];
a[0] = 1; a[1] = 3; a[2] = 5; a[3] = 7; a[4] = 9;
```

Type definitions

Compound types are defined with *typedef*, and are primarily used for defining the structure of messages to be sent over channels:

```
typedef MESSAGE {
  mtype messagetype;
  byte source;
  byte destination;
}
```

Inline

The inline construct is almost the same as the preprocessor macro feature, but with a more friendly syntax. The parameters of the *inline* sequence (if any) are replaced by the actual values and the sequence is inserted at the point of call.

```
inline swap(a,b) {
  int tmp;
  tmp = a; a = b; b = tmp;
}
proctype p() {
  int j,k;
  j = 2; k = 9;
  swap(j,k);
}
```

Preprocessor

```
/* Inclusion of external file */
#include "filename.h"

/* Define simple symbols */
#define N 4
#define mutex (critical <= 1)

/* Define macros */
#define increment(x) (x = x + 1)
#define swap(a,b) \
  int tmp;        \
  tmp = a;        \
  a = b;          \
  b = tmp;
```

Channels

A channel is datatype with 2 operations, *send* and *receive*. Every channel is associated with a specific message type. At most 255 channels can be created. It is possible to create an array of channels.

```
chan reply[2] = [4] of { byte };
chan ch = [capacity] of { typename[,typename] };
```

Send/receive

```
channel ! var1[,var2 ...]; /* Send    */
channel ? var1[,var2 ...]; /* Receive */
```

Channel capacity

A channel with capacity 0 is called a *rendez-vous channel*. Send and receive operations on a rendez-vous channel blocks until the peer process is ready, at which point the send and receive operation is executed synchronously and atomically.

A channel with a capacity larger than 0 is called a *buffered channel*. They behave like a FIFO with a specified capacity. Send and receive statements are executable if there is room in the channel, or messages in the channel, respectively. Otherwise they block until space or a message becomes available.

Special syntax for send/receive

Defining messages type(s):

```
mtype {open, close, reset};
chan ch = [1] of {mtype,byte,byte};
```

The message definition above allows send statements like this:

```
ch ! open, 2, 3; /* Send open message  */
ch ! close(4,7); /* Send close message */
```

A receiver might look like this:

```
proctype Receiver() {
  mtype request;
  byte parm1;
  byte parm2;
  ch ? request,parm1,parm2;
}
```

Checking contents of a channel

These functions are only allowed for buffered channels.

Predefined function	*Description*
full(channel)	True if channel is full.
nfull(channel)	True if channel is not full.
empty(channel)	True if channel is empty.
nempty(channel)	True if channel is not empty.
len(channel)	Return number of messages.

These functions must be used - !full and !empty are not allowed. *Warning:* do not use *else* alternatives in *if/do* that have channel expressions as guards; instead use the pairs *full/nfull* and *empty/nempty*.