

Grammaire de Promela

Dans cette annexe nous présentons la grammaire de la langue Promela. Cette grammaire peut être trouvée dans le livre de Holzmann [18] et sur la page d'accueil SPIN [16].

The following list defines the grammar of the input language for the SPIN model checker version 6. The notational conventions are as follows.

Choices are separated by vertical bars: |

Optional parts are included in square brackets: [...]

A Kleene star * indicates zero or more repetitions of the immediately preceding fragment

Literals are enclosed in single quotes: ' ... '

Uppercase names refer to tokens (i.e., terminals) representing keywords from the language. In Spin models, the same keywords are spelled just like these token names, but are always in lower- instead of upper-case.

Lowercase names refer to grammar rules from this list.

The name `any_ascii_char` refers to any printable ASCII character except the double quote character: `"`.

The statement separator used in this list is the semi-colon `';`'. In most cases the semi-colon can be replaced with a two-character arrow symbol: `'>'`, without change of meaning.

In the specification that follows non-terminals are hyper-linked to the production rule where they are defined, and terminals are hyper-linked to the online man-pages for the defining elements of the language.

Annexe B

Grammar Rules

```
spec      : module [ module ] *

module    : proctype          /* proctype declaration */
           | init              /* init process      */
           | never             /* never claim     */
           | trace             /* event trace     */
           | utype            /* user defined types */
           | mtype            /* mtype declaration */
           | decl_lst         /* global vars, chans */

proctype: [ active ] PROCTYPE name '(' [ decl_lst ] ')'
           [ priority ] [ enabler ] '{' sequence '}'

init      : INIT [ priority ] '{' sequence '}'

never     : NEVER '{' sequence '}'

trace     : TRACE '{' sequence '}'

utype     : TYPEDEF name '{' decl_lst '}'

mtype     : MTYPE [ '=' ] '{' name [ ',' name ] * '}'

decl_lst: one decl [ ';' one decl ] *

one decl: [ visible ] typename ivar [ ',' ivar ] *

typename: BIT | BOOL | BYTE | SHORT | INT | MTYPE | CHAN
           | uname /* user defined type names (see utype) */

active    : ACTIVE [ '[' const ']' ] /* instantiation */

priority: PRIORITY const /* simulation priority */

enabler   : PROVIDED '(' expr ')' /* execution constraint */

visible   : HIDDEN | SHOW

sequence: step [ ';' step ] *

step      : stmnt [ UNLESS stmnt ]
           | decl_lst
           | XR varref [ ',' varref ] *
           | XS varref [ ',' varref ] *

ivar      : name [ '[' const ']' ] [ '=' any_expr | '=' ch_init ]

ch_init : '[' const ']' OF '{' typename [ ',' typename ] * '}'

varref    : name [ '[' any_expr ']' ] [ '.' varref ]

send     : varref '!' send_args /* normal fifo send */
           | varref '!' '!' send_args /* sorted send */

receive : varref '?' recv_args /* normal receive */
           | varref '?' '?' recv_args /* random receive */
           | varref '?' '<' recv_args '>' /* poll with side-effect */
           | varref '?' '?' '<' recv_args '>' /* ditto */

poll     : varref '?' '[' recv_args ']' /* poll without side-effect */
           | varref '?' '?' '[' recv_args ']' /* ditto */

send_args: arg_lst | any_expr '(' arg_lst ')'

arg_lst  : any_expr [ ',' any_expr ] *
```

Annexe B

```
recv_args: recv_arg [ ', recv_arg ] * | recv_arg '(' recv_args ' '

recv_arg : varref | EVAL '(' varref ' ' | [ '-' ] const

assign : varref '=' any_expr /* standard assignment */
| varref '+' ++ /* increment */
| varref '-' -- /* decrement */

stmtnt : IF options FI /* selection */
| DO options OD /* iteration */
| FOR '(' range ' ' '{' sequence '}' /* iteration */
| ATOMIC '{' sequence '}' /* atomic sequence */
| D_STEP '{' sequence '}' /* deterministic atomic */
| SELECT '(' range ' ' /* non-deterministic value selection */
| '{' sequence '}' /* normal sequence */
| send
| receive
| assign
| ELSE /* used inside options */
| BREAK /* used inside iterations */
| GOTO name
| name ':' stmtnt /* labeled statement */
| PRINT '(' string [ ', arg_lst ] ' '
| ASSERT expr
| expr /* condition */
| c_code '{' ... '}' /* embedded C code */
| c_expr '{' ... '}'
| c_decl '{' ... '}'
| c_track '{' ... '}'
| c_state '{' ... '}'

range : varref ':' expr ..' expr
| varref IN varref

options : ':' ':' sequence [ ':' ':' sequence ] *

andor : '&' '&' | '|' '|'

binarop : '+' | '-' | '*' | '/' | '%' | '&' | '^' | '|'
| '>' | '<' | '>' '=' | '<' '=' | '=' '=' | '!' '='
| '<' '<' | '>' '>' | andor

unarop : '~' | '-' | '!'

any_expr: '(' any_expr ' '
| any_expr binarop any_expr
| unarop any_expr
| '(' any_expr '-' '>' any_expr ':' any_expr ' '
| LEN '(' varref ' ' /* nr of messages in chan */
| poll
| varref
| const
| TIMEOUT
| NP /* non-progress system state */
| ENABLED '(' any_expr ' ' /* refers to a pid */
| PC_VALUE '(' any_expr ' ' /* refers to a pid */
| name '[' any_expr ']' '@' name /* refers to a pid */
| RUN name '(' [ arg_lst ] ' ' [ priority ]
| get_priority( expr ) /* expr refers to a pid */
| set_priority( expr , expr ) /* first expr refers to a pid */

expr : any_expr
| '(' expr ' '
| expr andor expr
| chanpoll '(' varref ' ' /* may not be negated */

chanpoll: FULL | EMPTY | NFULL | NEMPTY
```

Annexe B

```
string  : ''' [ any_ascii_char ] * '''
uname   : name
name     : alpha [ alpha | number ] *
const    : TRUE | FALSE | SKIP | number [ number ] *
alpha    : 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j'  
          | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't'  
          | 'u' | 'v' | 'w' | 'x' | 'y' | 'z'  
          | 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J'  
          | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T'  
          | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z'  
          | '_'
number  : '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```