PROMELA LANGUAGE REPORT C

This appendix is a reference manual for PROMELA, the language for describing protocol validation models introduced in this book. It gives a terse overview of the main syntax requirements of the language. Semantics and usage is more fully explained in Chapters 5 and 6. This manual describes the language proper. It does not cover possible restrictions or extensions of specific implementations. In case of doubt, for instance when you have to find out what the precise effect is of an expression such as `(-10)%(-9)` or `(-10)<<(-2)` on your machine, the quickest way to learn is to execute a little PROMELA test program, like

```
init { printf("%d\t%d\n", (-10)%(-9), (-10)<<(-2)) }
```

using the PROMELA simulator (Chapter 12). The meaning of all binary, arithmetic, and relational operators matches that of ANSI standard C.

C.1 LEXICAL CONVENTIONS

There are five classes of tokens: identifiers, keywords, constants, operators and statement separators. Blanks, tabs, newlines, and comments serve only to separate tokens. If more than one interpretation is possible, a token is taken to be the longest string of characters that can constitute a token.

C.2 COMMENTS

Any string started with /* and terminated with */ is a comment. Comments cannot be nested.

C.3 IDENTIFIERS

An identifier is a single letter or underscore, followed by zero or more letters, digits, or underscores.

C.4 KEYWORDS

The following identifiers are reserved for use as keywords:

```
assert          atomic          bit             bool
break           byte            chan            do
```

```
fi              goto            if              init
int             len             mtype           never
od              of              printf          proctype
run             short           skip            timeout
```

C.5 CONSTANTS

There are three types of constants.

- String constants
- Enumeration constants
- Integer constants

String constants can only be used in `printf` statements.

Enumeration constants can be used to define symbolic names for message types. They can be defined in `mtype` declarations of the type

```
mtype = { namelist }
```

where `namelist` is a comma separated list of symbolic names. Only one `mtype` declaration per program can be used.

An integer constant is a sequence of digits representing a decimal integer. There are no floating point numbers in PROMELA.

C.6 EXPRESSIONS

The evaluation of expressions is defined in integer arithmetic. Unsigned data, that is all variables declared with type `bit`, `byte`, or `bool`, are cast to signed integers before being used in expressions. For example, the value of expression `(p-1)`, with `p` a variable of type byte (unsigned char) and value zero, is the signed value `-1` in PROMELA, and not the unsigned equivalent 255. On assignments, however, the type of the destination always prevails. The value `-1` is cast to 255 when it is stored in a unsigned variable, but it remains `-1` when stored in a signed variable.

The following operators can be used to build expressions.

`+, -, *, /, %,`	arithmetic operators
`>, >=, <, <=, ==, !=,`	relational operators
`&&, ||, !`	logical AND, OR, NOT
`&, |, ~, >>, <<`	C-style bit operators
`!, ?`	send and receive operators
`(), []`	grouping, indexing
`len, run`	special operators

The syntax, semantics, side effects, and machine dependencies of all operators match ANSI standard C. Table C.1 defines the precedence levels. The operators on the first line in the table have the highest precedence.

Most operators, including assignment =, take two operands. The boolean negation `!` and the unary minus – operator can be both unary and binary, depending on context. The assignment operator takes an expression on the right, and a variable reference on the left:

Table C.1 — Precedence and Associativity

Operators	Associativity
`()  []`	left to right
`~  -` *(unary minus)* `!` *(boolean negation)*	left to right
`*  /  %`	left to right
`+  -`	left to right
`>>  <<`	left to right
`>  <  >=  <=`	left to right
`==  !=`	left to right
`&`	left to right
`|`	left to right
`&&`	left to right
`||`	left to right
`!` *(send)* `?` *(receive)*	left to right
`len  run`	left to right
`=`	right to left

```
varref = expression
```

Unlike C, the assignment operator cannot be used in expressions in PROMELA. The unary operator, `len`, applies to message channels only, and the unary operator `run` applies to process types. Informally, we talk about `len`- or `run`- statements, and similarly about send- and receive-statements, for statements that contain these operators.

REMOTE REFERENCING

Global variables and local variables declared within the same process type can be referred to by name. For instance

```
byte glob;

proctype same()
{       bool loc;

here:   (loc+glob)
}
```

Local variables of other processes can be referred to as follows:

```
proctype other()
{
        assert(same[2].loc > 3) /* NB!: no longer valid in current version */
}
```

Here a process of type `other` refer to the local variable `loc` of the process with `pid` two, i.e., the second process that was instantiated. It is a run-time error if the type of that process is different from the specified type `same`.

The process state of a remote process can be tested with boolean colon expressions. For instance, the condition

```
same[2]:here    /* NB! the current syntax is: same[2]@here */
```

is true if and only if the process referred to is currently in the state that was labeled `here`. Remote referencing of variables and control flow states is intended to be used only in assertions and in temporal claims. The language definition, however, does not prevent other applications.

C.7 DECLARATIONS

Processes and variables must be declared before they can be used. Variables can be declared either locally, within a process type, or globally. A process can only be declared globally in a `proctype` declaration. `Proctype` declarations cannot be nested. Local declarations may appear anywhere in a process body. The scope of a local variable is the complete process body, irrespective where its declaration is placed. It is not accessible, though, until execution has passed the point of declaration at least once. There are six data types:

```
bit, bool, byte, chan, short, int
```

VARIABLES

A variable declaration begins with a keyword indicating the data type of the variable followed by a list of identifier names, each one optionally followed by an initializer.

```
byte name1, name2 = 4, name3;
chan qname; chan a = [3] of { byte };
```

The initializer must be an expression for a variable of a basic data type, and a channel specification for variables of type `chan`. By default, variables of all types except `chan` are initialized to zero. Variables of type `chan` must be initialized explicitly before they can be used for message passing. It is undefined what the result is of using an uninitialized channel variable. Most likely, it causes a fatal runtime error.

Table C.2 summarizes the width and attributes of the five basic data types.

Table C.2 — Basic Data Types

Name	Size (bits)	Usage
bit	1	unsigned
bool	1	unsigned
byte	8	unsigned
short	16	signed
int	32	signed

The names `bit` and `bool` are synonyms for a single bit of information. A `byte` is an unsigned quantity that can store a value between 0 and 255. `Short`s and `int`s are signed quantities that differ only in the range of values they can hold.

An array of variables is declared as follows:

```
int name1[N];
chan q[M];
```

where `N` and `M` are constants. An array declaration may have an initializer, which initializes all elements of the array. If the array is a channel, one message channel of the given type per array element is created. In the channel initializer

```
chan q[M] = [x] of { types }
```

`M` is a constant, `x` is an expression that specifies the size of the channel, and `types` is a comma separated list of one or more data types that defines the format of each message that can be passed through the channel. All channels are initialized to be empty. Initialized channel identifiers can be passed from one process to another in messages or in `run` statements.

C.8 PROCESSES AND TEMPORAL CLAIMS

A process declaration starts with the keyword `proctype` followed by a name, a list of formal parameters enclosed in round braces, and a sequence of statements and local variable declarations. The body of a process declaration is enclosed in parentheses.

```
proctype name( /* parameter declarations */ )
{
        /* declarations and statements */
}
```

The parameter declarations cannot have initializers. One process declaration is required in every PROMELA model: the initial process. It is declared without the keyword `proctype` and without a parameter list.

```
init {
        /* declarations and statements */
}
```

It is the first process running and it has `pid` zero.

A temporal claim starts with the keyword `never` and can contain any PROMELA text

```
never {
        /* declarations and statements */
}
```

There can be at most one temporal claim per PROMELA model. It is used to specify a correctness requirement about the executions of the system specified. The temporal claim specifies a behavior that is claimed to be impossible. The claim will normally only contain conditions, though it is valid to allow the temporal claim to contain variable declarations, atomic sequences, and send and receive statements. To violate a correctness claim, it must be possible to execute one statement, or one atomic sequence of statements, for every statement that is executed by any of the other processes in the model. By using the temporal claim in combination with acceptance-state labels, any linear-time propositional temporal logic formula on the

system behavior can be expressed (see Chapter 6).

C.9 STATEMENTS

There are twelve types of statements:

```
assertion       assignment      atomic          break
expression      goto            printf          receive
selection       repetition      send            timeout
```

Any statement can be preceded by one or more declarations. A statement can only be passed if it is executable. To determine its executability the statement can be evaluated: if evaluation returns a zero value the statement is blocked. In all other cases the statement is executable and can be passed. The evaluation of a compound expression is always indivisible. This means that the statement

```
(a == b && a != b)
```

will always be unexecutable, but the sequence

```
(a == b); (a != b)
```

may be executable in that order.

The act of passing the statement after a successful evaluation is called the ‘‘execution’’ of the statement. There is one *pseudo* statement, `skip`, which is syntactically equivalent to the condition `(1)`. `Skip`, is a null statement; it is always executable and has no effect when executed. It may be needed to satisfy syntax requirements.

Goto statements can be used to transfer control to any labeled statement within the same process or procedure. They are always executable. Assignments and declarations are also always executable. Expressions are only executable if they return a non-zero value. That is, the expression 0 (zero) is never executable, and similarly 1 is always executable.

Each statement may be preceded by a label: a name followed by a colon. Each label may be used as the destination of a `goto`. Three types of labels have predefined meanings in validations: end-state labels, progress-state labels, and acceptance-state labels. The semantics are explained in Chapter 6.

The remaining statements, selection, repetition, send, receive, break, timeout, and atomic sequences, are discussed below.

SELECTION

A selection statement begins with the keyword `if`, is followed by a list of one or more options and ends with the keyword `fi`. Every option begins with the flag `::` followed by any sequence of statements. One and only one option from a selection statement will be selected for execution. The first statement of an option determines whether the option can be selected or not. If more than one option is executable, one will be selected at random. Thus the language defines nondeterministic machines as defined on page 164.

REPETITION AND BREAK

A repetition or `do` statement is similar to a selection statement, but is executed repeatedly until either a `break` statement is executed or a `goto` jump transfers control outside the cycle. The keywords of the repetition statement are `do` and `od` instead of `if` and `fi`. The `break` statement will terminate the innermost repetition statement in which it is executed. The use of a break statement outside a repetition statement is illegal.

ATOMIC SEQUENCE

The keyword `atomic` introduces an atomic sequence of statements that is to be executed as one indivisible step. The syntax is as follows:

```
atomic { sequence };
```

Logically the sequence of statements is now equivalent to one single statement. It is a run-time error if any statement in an atomic sequence other than the first one is found to be unexecutable. The first statement is called the *guard* of the sequence. If it is executable, so should be the rest of the sequence. In general, therefore, the guard of an atomic sequence is followed only with local assignments and local conditions, but not with any send or receive statements.

SEND

The syntax of a send statement is

```
q!expr
```

where `q` is the name of a channel, and the evaluation of expression `expr` returns a value to be appended to the channel. The send statement is not executable (blocks) if the channel is full or does not exist. If more than one value is to be passed from sender to receiver, the expressions are written in a comma-separated list:

```
q!expr1,expr2,expr3
```

Equivalently, this may be written

```
q!expr1(expr2,expr3)
```

RECEIVE

The syntax of the receive statement is

```
q?name
```

where `q` is the name of a channel and `name` is a variable or a constant. If a constant is specified the receive statement is only executable if the channel exists and the oldest message stored in the channel contains the same value. If a variable is specified, the receive statement is executable if the channel exists and contains any message at all. The variable in that case will receive the value of the message that is retrieved. If more than one value is sent per message, the receive statement also take a comma-separated list of variables and constants,

```
q?name1,name2,...
```

which again is syntactically equivalent to

```
q?name1(name2,...)
```

Each constant in this list puts an extra condition on the executability of the receive: it must be matched by the value of the corresponding message field of the message to be retrieved. The variable fields retrieve the values of the corresponding message fields on a receive. It is an error to attempt to receive a value when none was transferred, and vice versa.

Any receive statement can be used as a side-effect free condition by enclosing its parameter list in square braces:

```
q?[name1,name2,...]
q?[name1(name2,...)]
```

The statement is executable (returns a non-zero result) only if the corresponding receive operation is executable, but it has no effect on the variables or the channel.

The only other type of operation allowed on channels is

```
len(varref)
```

where `varref` identifies an instantiated channel. The operation returns the number of messages in the channel specified, or zero if the channel does not exist.

TIMEOUT

The keyword `timeout` represents a condition that becomes true if and only if no other statement in the system is executable. A timeout statement has no effect when executed. Timeouts can be included in expressions.

C.10 MACROS AND INCLUDE FILES

The source text of a specification is processed by the C preprocessor for macro-expansion and file inclusions, Kernighan and Ritchie [1978].

C.11 PROMELA GRAMMAR

The grammar is listed in BNF-style. Parenthesis are used for grouping. A plus indicates a repetition of one or more times of the last syntactical unit; a star indicates a repetition of zero or more times. Square brackets are used to indicate optional elements. A vertical bar separates options. Literals are quoted. Terminals are written in upper-case, non-terminals in lower-case.

```
program    ::= { unit } +
```

```
unit       ::=  PROCTYPE NAME '(' [ decl_lst ] ')' body
           | CLAIM body
           | INIT body
           | one_decl
           | MTYPE ASGN '{' NAME { ',' NAME } * '}'
           | ';'

body       ::=  '{' sequence '}'

sequence   ::=  step { ';' step } *

step       ::=  [ decl_lst ] stmnt

one_decl   ::=  [ TYPE ivar { ',' ivar } * ]

decl_lst   ::=  one_decl { ';' one_decl } *

ivar       ::=  var_dcl | var_dcl ASGN expr | var_dcl ASGN ch_init

ch_init    ::=  '[' CONST ']' OF '{' TYPE { ',' TYPE } * '}'

var_dcl    ::=  NAME [ '[' CONST ']' ]

var_ref    ::=  NAME [ '[' expr ']' ]

stmnt      ::=  var_ref ASGN expr
           | var_ref RCV margs
           | var_ref SND margs
           | PRINT '(' STRING { ',' expr } * ')'
           | ASSERT expr
           | GOTO NAME
           | expr
           | NAME ':' stmnt
           | IF options FI
           | DO options OD
           | BREAK
           | ATOMIC '{' sequence '}'

options    ::=  { SEP sequence } +

binop      ::=  '+' | '-' | '*' | '/' | '%' | '&' | '|' | '>' | '<'
           | GE  | LE  | EQ  | NE  | AND | OR  | LSHIFT | RSHIFT

unop       ::=  '~' | '-' | SND
```

```
expr       ::= '(' expr ')'
           | expr binop expr
           | unop expr
           | RUN NAME '(' [ arg_lst ] ')'
           | LEN '(' var_ref ')'
           | var_ref RCV '[' margs ']'
           | var_ref
           | CONST
           | TIMEOUT
           | var_ref '.' var_ref
           | var_ref ':' NAME

arg_lst    ::=  expr { ',' expr } *

margs      ::=  arg_lst | expr '(' arg_lst ')'
```