Chapter 16. PROMELA Language Reference

> "The infinite multitude of things is incomprehensible, and more than a man may be able to contemplate."
>
> —(Giambattista della Porta, 1535–1615, Natural Magick)

The PROMELA manual pages that are included in this book can be grouped into seven main sections. The first five of these sections, plus the grammar description given here, describe the language proper. The entries from the sixth section cover those things that are deliberately not in the language, and contain a brief explanation of why they were left out. The entries from the seventh and last section cover the more recent extensions to the PROMELA language to support the use of embedded C code statements and data declarations. The main sections are:

1. Meta Terms (translated by preprocessors into vanilla PROMELA)
2. Declarators (for defining process, channel, and data objects)
3. Control Flow Constructors (separators, compound statements, jumps, labels, etc.)
4. Basic Statements (such as send, receive, assignment, etc.)
5. Predefined Functions and Operators (such as len, run, nempty, etc.)
6. Omissions (such as floating point, probabilities, etc.)
7. Extensions (for embedded C code)

This chapter contains the manual pages for the first six of these sections, listed in alphabetical order with the section name indicated at the top of each page. Chapter 17 separately introduces the extensions for embedded C code and contains the corresponding manual pages from the last section in our list.

In the tradition of the classic UNIX manuals, each manual page contains some or all of the following eight defining elements.

Grammar Rules

The following list defines the basic grammar of PROMELA. 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 grammar fragment; literals are enclosed in single quotes; uppercase names are keywords; lowercase names refer to the grammar rules from this list. The name `any_ascii_char` appears once, and is used to refer to any printable ASCII character except `'"'`. PROMELA keywords are spelled like the token−names in the grammar, but in lowercase instead of uppercase.

The statement separator used in this list is the semicolon ';'. In all cases, the semicolon can be replaced with the two−character arrow symbol: '−>' without change of meaning.

We will not attempt to include a full grammar description for the language C, as it can appear inside the embedded C code statements. Where it appears, we have abbreviated this as `... C ...` in the grammar rules that follow.

```
spec    : module [ module ] *

module  : utype      /* user defined types   */
        | mtype      /* mtype declaration    */
        | decl_lst   /* global vars, chans   */
        | proctype   /* proctype declaration */
        | init       /* init process - max 1 per model */
        | never      /* never claim  - max 1 per model */
        | trace      /* event trace  - max 1 per model */
        | c_code '{' ... C ... '}'
        | c_decl '{' ... C ... '}'
        | c_state string string [ string ]
        | c_track string string

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

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

never   : NEVER '{' sequence '}'

trace   : TRACE '{' sequence '}'
        | NOTRACE '{' 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  | PID
        | SHORT | INT  | MTYPE | CHAN
        | uname /* user defined typenames (see utype)  */

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

```
priority: PRIORITY const            /* simulation only */

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

visible : HIDDEN
        | SHOW

sequence: step [ ';' step ] *

step    : decl_lst
        | stmnt [ UNLESS stmnt ]
        | 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         /* fifo send   */
        | varref  '!' '!' send_args     /* sorted send */

receive : varref  '?' recv_args         /* fifo receive   */
        | varref  '?' '?' recv_args     /* random receive */
        | varref  '?' '<' recv_args '>' /* poll */
        | varref  '?' '?' '<' recv_args '>'

recv_poll: varref '?' '[' recv_args ']' /* test */
        |  varref '?' '?' '[' recv_args ']'

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

arg_lst : any_expr [ ',' any_expr ] *

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

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

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

stmnt   : IF options FI         /* selection  */
        | DO options OD         /* iteration  */
        | ATOMIC '{' sequence '}'
        | D_STEP '{' sequence '}'
        | '{' sequence '}'
        | send
        | receive
        | assign
        | ELSE                  /* guard statement   */
        | BREAK                 /* only inside loops */
        | GOTO name             /* anywhere          */
```

```
        | name ':' stmnt        /* labeled statement */
        | PRINT '(' string [ ',' arg_lst ] ')'
        | ASSERT expr
        | expr                  /* condition */
        | c_code [ c_assert ] '{' ... C ... '}'
        | c_expr [ c_assert ] '{' ... C ... '}'

c_assert: '[' ... C ... ']'     /* see p. 505 */

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 */
        | recv_poll
        | varref
        | const
        | TIMEOUT       /* hang system state */
        | NP_           /* non-progress system state */
        | ENABLED '(' any_expr ')'
        | PC_VALUE '(' any_expr ')'
        | name '[' any_expr ']' '@' name
        | RUN name '(' [ arg_lst ] ')' [ priority ]

expr    : any_expr
        | '(' expr ')'
        | expr andor expr
        | chanop '(' varref ')'

chanop  : FULL | EMPTY | NFULL | NEMPTY

string  : '"' [ any_ascii_char ] * '"'

uname   : name

name    : alpha [ alpha | const | '_' ] *

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'
```

Main Sections

The manual pages that follow are in alphabetical order, with the section name indicated. The pages can be grouped per section as follows:

Meta Terms

Declarators

Control Flow

Basic Statements

Predefined

Embedded C Code

Omissions

Reference

Table 16.1 gives an overview of all the manual pages that describe the PROMELA language, together with the corresponding page numbers. Five of the primitives are discussed in Chapter 17, with the corresponding manual pages following on pages 505 to 511.

Special Cases

Several language features apply only in special cases. Two types of special cases include those features that only affect the specific way in which either a simulation or a verification run is performed. Other types of special case include features that are either incompatible with the enforcement of SPIN's partial order reduction method or with the breadth−first search option, and features that are mutually incompatible. We summarize all these special cases next.

Table 16.1. Index of All Manual Pages

Simulation Only

A small number of language features apply only to simulations, and are ignored in verification runs. They are:

priority (p. 453), show (p. 477), and STDIN (p. 480).

The use of some special keywords inside print statements, such as `MSC:` and `BREAK` are only interpreted by the graphical user interface XSPIN. An explanation of these special keywords can be found in the manpage for the print statement on p. 451, and in Chapter 12 on p.272.

Verification Only

Some language features apply only to verifications, and are ignored in simulation runs. They include the special labels:

accept (p. 379), progress (p. 459), and end (p. 413),

as well as the verification related features:

ltl (434), never (p. 441), trace (p. 483), notrace (p, 483), xr (p. 493), and xs (p. 493).

Partial Order Reduction

Two PROMELA language features are incompatible with the enforcement of SPIN's partial order reduction algorithm. They are:

_last (p. 374), enabled (p. 412), and provided (p. 461).

This means that if these constructs appear in a verification model, the use of the partial order reduction algorithm cannot be considered safe and may cause an incompleteness of the search. If an error is found, the error report remains valid, but if no error is found this no longer implies that errors are impossible. The verifier will issue a warning when it detects the presence of one or both of the above two constructs, and the user did not disable the partial order reduction algorithm. To avoid the warning, and the problem, it suffices to compile the `pan.c` source with the extra directive `-DNOREDUCE`. As a result, the time and memory requirements may increase, but the accuracy of the search process will be secure.

Rendezvous: Rendezvous communication is incompatible with partial order reduction in a small number of cases. The partial order reduction algorithm can produce an invalid reduction when rendezvous send operations can appear in the guard of an escape clause of a PROMELA `unless` statement. When the verifier detects that a model contains both `unless` statements and rendezvous message passing operations, it will therefore always issue a warning, recommending the use of directive `-DNOREDUCE` to disable partial order reduction. If the warning is ignored and an error trace is found, it will nonetheless be accurate, so this mode of search may still be of some use.

Breadth−First Search: The situation is less favorable when a breadth−first search is performed for the same type of model. In this case false error reports would become possible, even in the absence of partial order reduction. If, therefore, the verifier detects the use of a rendezvous send operation as the guard statement of the escape clause of an `unless` statement, the verifier will abort the run in breadth−first search mode with an error message. The use of a rendezvous receive operation in the escape clause of an `unless` statement can be considered safe in both cases.

The LTL Next Operator: If SPIN is compiled from its sources with the additional compiler directive `-DNXT`, the use of the LTL 'next' operator, which is written `X`, is enabled. The use of this operator can conflict with SPIN's partial order reduction if the LTL formula that is specified is not stutter invariant. If you are not sure about stutter invariance, it is always best to disable partial order reduction whenever the `X` operator is used.

Fairness: In models that use rendezvous message passing, the weak fairness option is also not compatible with the use of partial order reduction. If this case is detected, the verifier will issue a warning. To suppress it, either omit the weak fairness option, or disable partial order reduction with compile−time directive `-DNOREDUCE`.

Remote References: Partial order reduction is incompatible with the use of remote referencing operations. The verifier will issue a warning if this is detected.

There are a few other types of incompatibility.

Channel Assertions and Buffered Channels: The channel assertions `xr` and `xs` can only be applied to buffered message channels; they cannot be used on rendezvous ports (i.e., on channels with a zero capacity).

Breadth−First Search and Rendezvous: The breadth−first search algorithm cannot be used on models that contain rendezvous statements in the escape clause of an `unless` statement. The verifier will issue a warning when it encounters this case.

Breadth−First Search and _last: Breadth−first search, finally, is incompatible with the use of the predefined variable `_last`. The verifier will issue a warning also in this case.

_

Name

_ – a predefined, global, write–only, integer variable.

Syntax

_

Description

The underscore symbol _ refers to a global, predefined, write–only, integer variable that can be used to store scratch values. It is an error to attempt to use or reference the value of this variable in any context.

Examples

The following example uses a `do`–loop to flush the contents of a channel with two message fields of arbitrary type, while ignoring the values of the retrieved messages:

```
do
:: q?_,_
:: empty(q) -> break
od
```

See Also

_nr_pr, _last, _pid, np_, hidden

_last

Name

`_last` – a predefined, global, read–only variable of type `pid`.

Syntax

`_last`

Description

`_last` is a predefined, global, read–only variable of type `pid` that holds the instantiation number of the process that performed the last step in the current execution sequence. The initial value of `_last` is zero.

The `_last` variable can only be used inside `never` claims. It is an error to assign a value to this variable in any context.

Examples

The following sample `never` claim attempts to match an infinite run in which the process with process initialization number one executes every other step, once it starts executing.

```
never {
        do
        :: (_last != 1)
        :: else -> break
        od;
accept:
        do
        :: (_last != 1) -> (_last == 1)
        od
}
```

Because the initial value of variable `_last` is zero, the first guard in the first `do` loop is always true in the initial state. This first loop is designed to allow the claim automaton to execute dummy steps (passing through its `else` clause) until the process with instantiation number one executes its first step, and the value of `_last` becomes one. Immediately after this happens, the claim automaton moves from into its second state, which is accepting. The remainder of the run can only be accepted, and reported through SPIN's acceptance cycle detection method, if the process with instantiation number one continues to execute every other step. The system as a whole may very well allow other executions, of course. The `never` claim is designed, though, to intercept just those runs that match the property of interest.

Notes

During verifications, this variable is not part of the state descriptor unless it is referred to at least once. The additional state information that is recorded in this variable will generally cause an increase of the number of reachable states. The most serious side effect of the use of the variable `_last` in a model is, though, that it prevents the use of both partial order reduction and of the breadth−first search option.

See Also

_, _nr_pr, _pid, never, np_

_nr_pr

Name

`_nr_pr` – a predefined, global, read–only, integer variable.

Syntax

`_nr_pr`

Description

The predefined, global, read–only variable `_nr_pr` records the number of processes that are currently running (i.e., active processes). It is an error to attempt to assign a value to this variable in any context.

Examples

The variable can be used to delay a parent process until all of the child processes that it created have terminated. The following example illustrates this type of use:

```
proctype child()
{
        printf("child %d\n", _pid)
}
active proctype parent()
{
        do
        :: (_nr_pr == 1) ->
                run child()
        od
}
```

The use of the precondition on the creation of a new child process in the `parent` process guarantees that each child process will have process instantiation number one: one higher than the parent process. There can never be more than two processes running simultaneously in this system. Without the condition, a new child process could be created before the last one terminates and dies. This means that, in principle, an infinite number of processes could result. The verifier puts the limit on the number of processes that can effectively be created at 256, so in practice, if this was attempted, the 256th attempt to create a child process would fail, and the `run` statement from this example would then block.

See Also

_, _last, _pid, active, procedures, run

_pid

Name

_pid – a predefined, local, read–only variable of type pid that stores the instantiation number of the executing process.

Syntax

_pid

Description

Process instantiation numbers begin at zero for the first process created and count up for every new process added. The first process, with instantiation number zero, is always created by the system. Processes are created in order of declaration in the model. In the initial system state only process are created for active proctype declarations, and for an init declaration, if present. There must be at least one active proctype or init declaration in the model.

When a process terminates, it can only die and make its _pid number available for the creation of another process, if and when it has the highest _pid number in the system. This means that processes can only die in the reverse order of their creation (in stack order).

The value of the process instantiation number for a process that is created with the run operator is returned by that operator.

Instantiation numbers can be referred to locally by the executing process, through the predefined local _pid variable, and globally in never claims through remote references.

It is an error to attempt to assign a new value to this variable.

Examples

The following example shows a way to discover the _pid number of a process, and gives a possible use for a process instantiation number in a remote reference inside a never claim.

```
active [3] proctype A()
{
	printf("this is process: %d\n", _pid);
L:	printf("it terminates after two steps\n")
}

never {
	do
```

```
        :: A[0]@L -> break
        od
}
```

The remote reference in the claim automaton checks whether the process with instantiation number zero has reached the statement that was marked with the label `L`. As soon as it does, the claim automaton reaches its end state by executing the `break` statement, and reports a match. The three processes that are instantiated in the `active proctype` declaration can execute in any order, so it is quite possible for the processes with instantiation numbers one and two to terminate before the first process reaches label `L`.

Notes

A `never` claim, if present, is internally also represented by the verifier as a running process. This claim process has no visible instantiation number, and therefore cannot be referred to from within the model. From the user's point of view, the process instantiation numbers are independent of the use of a `never` claim.

See Also

_, _last, _nr_pr, active, init, never, proctype, remoterefs, run

accept

Name

accept – label–name prefix used for specifying liveness properties.

Syntax

accept [a-zA-Z0-9_] *: stmnt

Description

An accept label is any label name that starts with the six–character sequence accept. It can appear anywhere a label can appear as a prefix to a PROMELA statement.

Accept labels are used to formalize Büchi acceptance conditions. They are most often used inside never claims, but their special meaning is also recognized when they are used inside trace assertions, or in the body of a proctype declaration. There can be any number of accept labels in a model, subject to the naming restrictions that apply to all labels (i.e., a given label name cannot appear more than once within the same defining scope).

A local process statement that is marked with an accept label can also mark a set of global system states. This set includes all states where the marked statement has been reached in the process considered, but where the statement has not yet been executed. The SPIN generated verifiers can prove either the absence or presence of infinite runs that traverse at least one accept state in the global system state space infinitely often. The mechanism can be used, for instance, to prove LTL liveness properties.

Examples

The following proctype declaration translates into an automaton with precisely three local states: the initial state, the state in between the send and the receive, and the (unreachable) final state at the closing curly brace of the declaration.

The accept label in this model formalizes the requirement that the second state cannot persist forever, and cannot be revisited infinitely often either. In the given program this would imply that the execution should eventually always stop at the initial state, just before the execution of sema!p.

```
active proctype dijkstra()
{       do
        :: sema!p ->
accept:         sema?v
        od
}
```

Notes

When a `never` claim is generated from an LTL formula, it already includes all required accept labels. As an example, consider the following SPIN generated `never` claim:

```
dell: spin -f '[]<>(p U q)'
never {    /* []<>(p U q) */
T0_init:
        if
        :: (q) -> goto accept_S9
        :: (1) -> goto T0_init
        fi;
accept_S9:
        if
        :: (1) -> goto T0_init
        fi;
}
```

In this example, the second state of the claim automaton was marked as an accepting state.

Since in most cases the `accept` labels are automatically generated from LTL formula, it should rarely be needed to manually add additional labels of this type elswhere in a verification model.

See Also

end, labels, ltl, never, progress, trace

active

Name

`active` – prefix for `proctype` declarations to instantiate an initial set of processes.

Syntax

`active proctype` name ([decl_lst]) { sequence}

`active` '[' const ']' `proctype` name ([decl_lst]) { sequence }

Description

The keyword `active` can be prefixed to any `proctype` declaration to define a set of processes that are required to be active (i.e., running) in the initial system state. At least one active process must always exist in the initial system state. Such a process can also be declared with the help of the keyword `init`.

Multiple instantiations of the same `proctype` can be specified with an optional array suffix of the `active` prefix. The instantiation of a `proctype` requires the allocation of a process state and the instantiation of all associated local variables. At the time of instantiation, a unique process instantiation number is assigned. The maximum number of simultaneously running processes is 255. Specifying a constant greater than 255 in the suffix of an `active` keyword would result in a warning from the SPIN parser, and the creation of only the first 255 processes.

Processes that are instantiated through an `active` prefix cannot be passed arguments. It is, nonetheless, legal to declare a list of formal parameters for such processes to allow for argument passing in additional instantiations with a `run` operator. In this case, copies of the processes instantiated through the `active` prefix have all formal parameters initialized to zero. Each active process is guaranteed to have a unique `_pid` within the system.

Examples

```
active proctype A(int a) { ... }
active [4] proctype B() { run A(_pid) }
```

One instance of `proctype A` is created in the initial system state with a parameter value for `a` of zero. In this case, the variable `a` is indistinguishable from a locally declared variable. Four instances of `proctype B` are also created. Each of these four instances will create one additional copy of `proctype A`, and each of these has a parameter value equal to the process instantiation number of the executing process of type `B`. If the process of type `A` is assigned `_pid` zero, then the four process of type `B` will be assigned `_pid` numbers one to three. All five processes that are declared through the use of the two `active` prefixes are guaranteed to be created and instantiated before any of these processes starts executing.

Notes

In many PROMELA models, the `init` process is used exclusively to initialize other processes with the `run` operator. By using `active` prefixes instead, the `init` process becomes superfluous and can be omitted, which reduces the amount of memory needed to store global states.

If the total number of active processes specified with `active` prefixes is larger than 255, only the first 255 processes (in the order of declaration) will be created.

See Also

_pid, init, proctype, remoterefs, run

arrays

Name

arrays – syntax for declaring and initializing a one–dimensional array of variables.

Syntax

typename name '[' const ']' [= any_expr]

Description

An object of any predefined or user–defined datatype can be declared either as a scalar or as an array. The array elements are distinguished from one another by their array index. As in the C language, the first element in an array always has index zero. The number of elements in an array must be specified in the array declaration with an integer constant (i.e., it cannot be specified with an expression). If an initializer is present, the initializing expression is evaluated once, and all array elements are initialized to the same resulting value.

In the absence of an explicit initializer, all array elements are initialized to zero.

Data initialization for global variables happens in the initial system state. All process local variables are initialized at process instantiation. The moment of creation and initialization of a local variable is independent of the precise place within the `proctype` body where the variable declaration is placed.

Examples

The declaration

```
byte state[N]
```

with `N` a constant declares an array of `N` bytes, all initialized to zero by default. The array elements can be assigned to and referred to in statements such as

```
state[0] = state[3] + 5 * state[3*2/n]
```

where `n` is a constant or a variable declared elsewhere. An array index in a variable reference can be any valid (i.e., side–effect free) PROMELA expression. The valid range of indices for the array `state`, as declared here, is `0..N-1`.

Notes

Scalar objects are treated as shorthands for array objects with just one element. This means that references to scalar objects can always be suffixed with `[0]` without triggering a complaint from the SPIN parser. Be warned, therefore, that if two arrays are declared as

```
byte a[N], b[N];
```

then the assignment

```
a = b;
```

will have the same effect as

```
a[0] = b[0];
```

and will not copy all the elements of the arrays.

An array of `bit` or `bool` variables is stored by the verifier as an array of `unsigned char` variable, and therefore saves no memory over a `byte` array. It can be better, therefore, to use integers in combination with bit−masking operations to simulate operations on a bit−array when memory is tight. The same rules apply here as would apply for the use of bit−arrays in C programs.

Multidimensional arrays can be constructed indirectly with the use of `typedef` definitions.

The use of an array index value outside the declared range triggers a run−time error in SPIN. This default array−index bound checking can be turned off during verifications, if desired, for increased performance. This can be done by compiling the `pan.c` source with the additional directive `−DNOBOUNDCHECK`.

See Also

chan, datatypes, mtype, typedef

assert

Name

`assert` – for stating simple safety properties.

Syntax

`assert` (expr)

Executability

true

EFFECT

`none`

Description

An `assert` statement is similar to `skip` in the sense that it is always executable and has no other effect on the state of the system than to change the local control state of the process that executes it. A very desirable side effect of the execution of this statement is, however, that it can trap violations of simple safety properties during verification and simulation runs with SPIN.

The `assert` statement takes any valid PROMELA expression as its argument. The expression is evaluated each time the statement is executed. If the expression evaluates to false (or, equivalently, to the integer value zero), an assertion violation is reported.

Assertion violations can be ignored in a verification run, by invoking the SPIN generated verifier with run–time option `-A`, as in:

```
$ ./pan -A
```

Examples

The most common type of assertion statement is one that contains just a simple boolean expression on global or local variable values, for instance, as in:

```
assert(a > b)
```

A second common use of the assertion is to mark locations in a `proctype` body that are required, or assumed, to be unreachable, as in:

```
assert(false)
```

If the statement is reached nonetheless, it will be reported as an assertion violation. A statement of this type is comparable to the infamous

```
printf("this cannot happen\n");
```

from C programs.

If more than one such assertion is needed, tracking can be made easier by using slight variations of expressions that necessarily will evaluate to `false`, such as:

```
assert(1+1 != 2)
assert(1>2)
assert(2>3)
```

The `assert` statement can also be used to formalize general system invariants, that is, boolean conditions that are required to be invariantly true in all reachable system states. To express this, we can place the system invariant in an independently executed process, as in:

```
active proctype monitor()
{
        assert(invariant)
}
```

where the name of the proctype is immaterial. Since the process instance is executed independently from the rest of the system, the assertion may be evaluated at any time: immediately after process instantiation in the initial system state, or at any time later in the system execution.

Several observations can be made about this example. First note that the process of type `monitor` has two states, and that the transition from the first to the second state is always unconditionally executable. This means that during verifications the addition of this specific form of the monitor process will double the size of the reachable state space. We can avoid this doubling by restricting the execution of the assertion to only those cases where it could actually lead to the detection of an assertion violation, for instance, as follows:

```
active proctype monitor()
{
        atomic { !invariant -> assert(false) }
}
```

This also solves another problem with the first version. Note that if our model contains a timeout condition, then the first monitor process would always be forced to execute the assertion before the system variable `timeout` variable could be set to true. This would mean that the assertion could never be checked beyond the first firing of a `timeout`. The second version of the monitor does not have this problem.

Notes

A simulation, instead of a verification, will not necessarily prove that a safety property expressed with an `assert` statement is valid, because it will check its validity on just a randomly chosen execution. Note that placing a system invariant assertion inside a loop, as in

```
active proctype wrong()
{
        do
        :: assert(invariant)
        od
}
```

still cannot guarantee that a simulation would check the assertion at every step. Recall that the fact that a statement can be executed at every step does not guarantee that it also will be executed in that way. One way to accomplish a tighter connection between program steps and assertion checks is to use a one−state `never` claim, for instance, as in:

```
never {
        do
        :: assert(invariant)
        od
}
```

This is an acceptable alternative in verifications, but since `never` claims are ignored in simulation runs, it would make it impossible to detect the assertion violation during simulations.

See Also

ltl, never, timeout, trace

assignment

Name

assignment – for assigning a new value to a variable.

Syntax

varref = any_expr

varref++ as shorthand for varref = varref +1

varref–– as shorthand for varref = varref –1

Executability

true

Effect

Replaces the value of varref with the value of any_expr, where necessary truncating the latter value to the range of the datatype of varref.

Description

The assignment statement has the standard semantics from most programming languages: replacing the value stored in a data object with the value returned by the evaluation of an expression. Other than in the C language, the assignment as a whole returns no value and can therefore itself not be part of an expression.

The variable reference that appears on the left–hand side of the assignment operator can be a scalar variable, an array element, or a structure element.

Examples

```
a = 12                  /* scalar */
r.b[a] = a * 4 + 7      /* array element in structure */
```

Note that it is not valid to write:

```
a = b++
```

because the right−hand side of this assignment is not a side effect free expression in PROMELA, but it is shorthand for another assignment statement. The effect of this statement can be obtained, though, by writing:

```
atomic { a = b; b++ }
```

or even more efficiently:

```
d_step { a = b; b++ }
```

Similarly, there are no shorthands for other C shorthands, such as `++b`, `--b`, `b *= 2`, `b += a`, etc. Where needed, their effect can be reproduced by using the non−shortened equivalents, or in some cases with `atomic` or `d_step` sequences.

Notes

There are no compound assignments in PROMELA, e.g., assignments of structures to structures or arrays to arrays in a single operation. If `x` and `y` are structures, though, the effect of a compound assignment could be approximated by passing the structure through a message channel, for instance as in:

```
typedef D {
        short f;
        byte g
};

chan m = [1] of { D };

init {
        D x, y;

        m!x;    /* send structure x to channel m     */
        m?y     /* receive and assign to structure y */
}
```

All variables must be declared before they can be referenced or assigned to. The default initial value of all variables is zero.

See Also

arrays, condition, datatypes, typedef

atomic

Name

`atomic` – for defining a fragment of code that is to be executed indivisibly.

Syntax

`atomic` { sequence }

Effect

Within the semantics model, as defined in Chapter 7, a side effect of the execution of any statement, except the last, from an atomic sequence is to set global system variable `exclusive` to the instantiation number of the executing process, thus preserving the exclusive privilige to execute.

Description

If a sequence of statements is enclosed in parentheses and prefixed with the keyword `atomic`, this indicates that the sequence is to be executed as one indivisible unit, non–interleaved with other processes. In the interleaving of process executions, no other process can execute statements from the moment that the first statement of an atomic sequence is executed until the last one has completed. The sequence can contain arbitrary PROMELA statements, and may be non–deterministic.

If any statement within the atomic sequence blocks, atomicity is lost, and other processes are then allowed to start executing statements. When the blocked statement becomes executable again, the execution of the atomic sequence can be resumed at any time, but not necessarily immediately. Before the process can resume the atomic execution of the remainder of the sequence, the process must first compete with all other active processes in the system to regain control, that is, it must first be scheduled for execution.

If an atomic sequence contains a rendezvous send statement, control passes from sender to receiver when the rendezvous handshake completes. Control can return to the sender at a later time, under the normal rules of non–deterministic process interleaving, to allow it to continue the atomic execution of the remainder of the sequence. In the special case where the recepient of the rendezvous handshake is also inside an atomic sequence, atomicity will be passed through the rendezvous handshake from sender to receiver and is not interrupted (except that another process now holds the exclusive privilige to execute).

An atomic sequence can be used wherever a PROMELA statement can be used. The first statement of the sequence is called its guard, because it determines when the sequence can be started. It is allowed, though not good style, to jump into the middle of an atomic sequence with a `goto` statement, or to jump out of it in the same way. After jumping into the sequence, atomic execution may begin when the process gains control, provided that the statement jumped to is executable. After jumping out of an atomic sequence, atomicity is lost, unless the target of the jump is also contained in an atomic sequence.

Examples

```
atomic {        /* swap the values of a and b */
        tmp = b;
        b = a;
        a = tmp
}
```

In the example, the values of two variables `a` and `b` are swapped in an uninterruptable sequence of statement executions. The execution of this sequence cannot be blocked, since all the statements it contains are always unconditionally executable.

An example of a non−deterministic `atomic` sequence is the following:

```
atomic {
        if
        :: a = 1
        :: a = 2
        fi;
        if
        :: b = 1
        :: b = 2
        fi
}
```

In this example, the variables `a` and `b` are assigned a single value, with no possible intervening statement from any other process. There are four possible ways to execute this `atomic` sequence.

It is possible to create a global atomic chain of executions, with two or more processes alternately executing, by passing control back and forth with rendezvous operations.

```
chan q = [0] of { bool };
active proctype X() { atomic { A; q!0; B  } }
active proctype Y() { atomic { q?0 -> C } }
```

In this example, for instance, execution could start in process `X` with the program block named `A`. When the rendezvous handshake is executed, atomicity would pass to process `Y`, which now starts executing the block named `C`. When it terminates, control can pass back to `X`, which can then atomically execute the block named `B`.

It is often useful to use atomic sequences to start a series of processes in such a way that none of them can start executing statements until all of them have been initialized:

```
atomic {
```

```
        run A(1,2);
        run B(2,3);
        run C(3,1)
}
```

Notes

Atomic sequences can be used to reduce the complexity of a verification.

If an infinite loop is accidently included in an atomic sequence, the verifier cannot always recognize the cycle. In the default depth−first search mode, the occurrence of such an infinite cycle will ultimately lead to the depth limit being exceeded, which will truncate the loop. In breadth−first search mode, though, this type of an infinite cycle will be detected. Note that it is an error if an infinite cycle appears inside an atomic sequence, since in that case the atomic sequence could not possibly be executed atomically in any real implementation.

PROMELA `d_step` sequences can be executed significantly more efficiently by the verifier than `atomic` sequences, but do not allow non−determinism.

See Also

d_step, goto, receive, send

break

Name

`break` – jump to the end of the innermostn `do` loop.

Syntax

`break`

Description

The keyword `break` does not indicate an executable statement, but it instead acts like a special type of semicolon: merely indicating the next statement to be executed. The search for the next statement to execute continues at the point that immediately follows the innermost `do` loop.

When the keyword `break` does not follow a statement, but appears as a guard in an option of a selection structure or `do` loop, then the execution of this statement takes one execution step to reach the target state, as if it were a `skip`. In all other cases, the execution of a `break` statement requires no separate step; the move to the target state then occurs after the execution of the preceding statement is completed.

If the repetition structure in which the `break` statement occurs is the last statement in a `proctype` body or `never` claim, then the target state for the `break` is the process's or claim's normal termination state, where the process or claim remains until it dies and is removed from the system.

Examples

```
L1: do
    :: t1 -> t2
    :: t3 -> break
    :: break
    od;
L2: ...
```

In this example, control reaches the label `L1` immediately after statement `t2` is executed. Control can also reach label `L2` immediately after statement `t3` is executed, and optionally, in one execution step, control can also move from label `L1` to label `L2`.

Notes

It is an error to place a `break` statement where there is no surrounding repetition structure. The effect of a `break` statement can always be replicated with the use of a `goto` statement and a label.

See Also

do, goto, if, labels, skip

chan

Name

chan – syntax for declaring and initializing message passing channels.

Syntax

chan name

chan name = '[' const ']' of { typename [, typename] * }

Description

Channels are used to transfer messages between active processes. Channels are declared using the keyword chan, either locally or globally, much like integer variables. Channels by default store messages in first–in first–out order (but see also the sorted send option in the manual page for send and the random receive option in the manual page for receive).

The keyword chan can be followed by one or more names, in a comma–separated list, each optionally followed by a channel initializer. The syntax

```
chan a, b, c[3]
```

declares the names a, b, and c as uninitialized channels, the last one as an array of three elements.

A channel variable must be initialized before it can be used to transfer messages. It is rare to declare just a channel name without initialization, but it occurs in, for instance, proctype parameter lists, where the initialized version of a channel is not passed to the process until a process is instantiated with a run operator.

The channel initializer specifies a channel capacity, as a constant, and the structure of the messages that can be stored in the channel, as a comma–separated list of type names. If the channel capacity is larger than zero, a buffered channel is initialized, with the given number of slots to store messages. If the capacity is specified to be zero, a rendezvous port, also called a synchronous channel, is created. Rendezvous ports can pass messages only through synchronous handshakes between sender and receiver, but they cannot store messages.

All data types can be used inside a channel initializer, including typedef structure names, but not including the typename unsigned.

Examples

The following channel declaration contains an initializer:

```
chan a = [16] of { short }
```

The initializer says that channel `a` can store up to 16 messages. Each message is defined to have only one single field, which must be of type `short`. Similarly,

```
chan c[3] = [0] of { mtype }
```

initializes an array of three rendezvous channels for messages that contain just one message field, of type `mtype`.

The following is an example of the declaration of a channel that can pass messages with multiple field:

```
chan qname = [8] of { mtype, int, chan, byte }
```

This time the channel can store up to eight messages, each consisting of four fields of the types listed. The `chan` field can be used to pass a channel identifier from one process to another. In this way, a channel that is declared locally within one process, can be made accessible to other processes. A locally declared and instantiated channel disappears, though, when the process that contain the declaration dies.

Notes

The first field in a channel type declaration is conventionally of type `mtype`, and is used to store a message type indicator in symbolic form.

In verification, buffered channels contribute significantly to verification complexity. For an initial verification run, choose a small channel capacity, of, say, two or three slots. If the verification completes swiftly, consider increasing the capacity to a larger size.

See Also

arrays, datatypes, empty, full, len, mtype, nempty, nfull, poll, receive, send

comments

Name

comments – default preprocessing rules for comments.

Syntax

/'*' [any_ascii_char] * '*'/

Description

A comment starts with the two character sequence `/*` and ends at the first occurrence of the two character sequence `*/`. In between these two delimiters, any text, including newlines and control characters, is allowed. None of the text has semantic meaning in PROMELA.

A comment can be placed at any point in a verification model where white space (spaces, tabs, newlines) can appear.

Examples

```
/* comment */ init /* comment */ {
        int /* an integer */ v /* variable */;

        v /* this / * is * / okay */ ++;
}
```

This PROMELA fragment is indistinguishable to the parser to the following PROMELA text, written without comments:

```
init {
        int v;
        v++;
}
```

Notes

Comments are removed from the PROMELA source before any other operation is performed. The comments are removed by invoking the standard C preprocessor `cpp` (or any equivalent program, such as `gcc -E`), which then runs as an external program in the background. This means that the precise rules for comments are determined by the specific C preprocessor that is used. Some preprocessors, for instance, accept the C++ commenting style, where comments can start with two forward slashes and end at the first newline. The specific preprocessor that is used can be set by the user. For more details on this, see the manual page for `macros`.

With the default preprocessor, conform ANSI−C conventions, comments do not nest. Be careful, therefore, that if a closing comment delimiter is accidentily deleted, all text up to and including the end of the next comment may be stripped.

On a PC, SPIN first tries to use a small, built−in macro preprocessor. When this fails, for instance, when macros with multiple parameters are used or when additional preprocessor directives are provided on the command line, the standard external C preprocessor is called. The use of the built−in preprocessor can, with older PC operating systems, avoid the the awkward brief appearance of an external shell window in the parsing phase.

See Also

macros

cond_expr

Name

conditional expression – shorthand for a conditional evaluation.

Syntax

(any_expr –> any_expr :any_expr)

Description

The conditional expression in PROMELA is based on the version from the C programming language. To avoid parsing conflicts, though, the syntax differs slightly from C. Where in C one would write

```
p?q:r
```

the corresponding expression in PROMELA is

```
(p -> q : r)
```

The question mark from the C version is replaced with an arrow symbol, to avoid confusion with the PROMELA receive operator. The round braces around the conditional expression are required, in this case to avoid the misinterpretation of the arrow symbol as a statement separator.

When the first expression (`p` in the example) evaluates to non–zero, the conditional expression as a whole obtains the value of the second expression (`q`), and else it obtains the value of the last expression (`r`).

Examples

The following example shows a simple way to implement conditional rendezvous operations.

```
chan q[3] = [0] of { mtype };

sender:    q[(P -> 1 : 2)]!msg -> ...

receiver:  q[(Q -> 1 : 0)]?msg -> ...
```

Two dummy rendezvous channels (`q[0]` and `q[2]`) are used here to deflect handshake attempts that should fail. The handshake can only successfully complete (on channel `q[1]`) if both the boolean expression `P` at the receiver side and the boolean expression `Q` at the sender side evaluate to true simultaneously. The dummy rendezvous channels `q[0]` and `q[2]` that are used here do not contribute any measurable overhead in a verification, since rendezvous channels take up no memory in the state vector.

An alternative way of specifying a conditional rendezvous operation is to add an extra message field to the channel and to use the predefined `eval` function in the receive statement, as follows.

```
global:         chan port = [0] of { mtype, byte, byte };

sender:         port!mesg(12, (P -> 1 : 0))
receiver:       port?mesg(data, eval(Q -> 1 : 2))
```

The handshake can again only happen if both `P` and `Q` evaluate to true. Unfortunately, the message field cannot be declared as a boolean, since we need a third value to make sure no match occurs when both `P` and `Q` evaluate to false.

See Also

condition, do, eval, if, unless

condition

Name

condition statement – for conditional execution and synchronization.

Syntax

expr

Executability

(expr != 0)

Effect

`none`

Description

In PROMELA, a standalone expression is a valid statement. A condition statement is often used as a guard at the start of an option sequence in a selection or repetition structure. Execution of a condition statement is blocked until the expression evaluates to a non–zero value (or, equivalently, to the boolean value true). All PROMELA expressions are required to be side effect free.

Examples

```
(1)                   /* always executable               */
(0)                   /* never executable                */
skip                  /* always executable, same as (1)  */
true                  /* always executable, same as skip */
false                 /* always blocks, same as (0)      */
a == b                /* executable only when a equals b */
```

A condition statement can only be executed (passed) if it holds. This means that the statement from the first example can always be passed, the second can never be passed, and the last cannot be passed as long as the values of variables `a` and `b` differ. If the variables `a` and `b` are local, the result of the evaluation cannot be influenced by other processes, and this statement will work as either `true` or `false`, depending on the values of the variables. If at least one of the variables is global, the statement can act as a synchronizer between processes.

See Also

do, else, false, if, skip, true, timeout, unless

d_step

Name

d_step – introduces a deterministic code fragment that is executed indivisibly.

Syntax

d_step { sequence }

Description

A d_step sequence is executed as if it were one single indivisible statement. It is comparable to an atomic sequence, but it differs from such sequences on the following three points:

- No goto jumps into or out of a d_step sequence are allowed.
- The sequence is executed deterministically. If non–determinism is present, it is resolved in a fixed and deterministic way, for instance, by always selecting the first true guard in every selection and repetition structure.
- It is an error if the execution of any statement inside the sequence can block. This means, for instance, that in most cases send and receive statements cannot be used inside d_step sequences.

Examples

The following example uses a d_step sequence to swap the value of all elements in two arrays:

```
#define N      16

byte a[N], B[N];

init {
        d_step {        /* swap elements */
                byte i, tmp;

                i = 0;
                do
                :: i < N ->
                        tmp = b[i];
                        b[i] = a[i];
                        a[i] = tmp; i++
                :: else ->
                        break
                od;
                skip    /* add target for break */
        }
        ...
```

```
}
```

A number of points should be noted in this example. First, the scope of variables `i` and `tmp` is independent of the precise point of declaration within the `init` body. In particular, by placing the declaration inside the `d_step` sequence we do not limit the scope of these variables to the `d_step` sequence: they remain visible also after the sequence.

Second, we have to be careful that the loop that is contained within this `d_step` sequence terminates. No system states are saved, restored, or checked during the execution of a `d_step` sequence. If an infinite loop is accidentily included in such a sequence, it can cause the verifier to hang.

Third and last, because one cannot jump into or out of a `d_step` sequence, a `break` from a `do` loop which appears as the last construct in a `d_step` sequence will trigger a parse error from SPIN. Note that this type of `break` statement creates an hidden jump out of the `d_step`, to the statement that immediately follows the `do` loop, which is outside the `d_step` itself in this case. The problem can be avoided by inserting a dummy `skip` after the loop, as shown in the example. There is no run−time penalty for this `skip` statement.

Notes

A `d_step` sequence can be executed much more efficiently during verifications than an `atomic` sequence. The difference in performance can be significant, especially in large−scale verifications.

The `d_step` sequence also provides a mechanism in PROMELA to add new types of statements to the language, translating into new types of transitions in the underlying automata. A `c_code` statement has similar properties.

See Also

atomic, c_code, goto, sequence

datatypes

Name

bit, bool, byte, pid, short, int, unsigned – predefined data types.

Syntax

typename name [= anyexpr]

unsigned name : constant [= anyexpr]

Description

There are seven predefined integer data types: bit, bool, byte, pid, short, int, and unsigned. There are also constructors for user–defined data types (see the manual pages for mtype, and typedef), and there is a separate predefined data type for message passing channels (see the manual page for chan).

Variables of the predefined types can be declared in C–like style, with a declaration that consists of a typename followed by a comma–separated list of one or more identifiers. Each variable can optionally be followed by an initializer. Each variable can also optionally be declared as an array, rather than as a scalar (see the manual page for arrays).

The predefined data types differ only in the domain of integer values that they provide. The precise domain may be system dependent in the same way that the corresponding data types in the C language can be system dependent.

Variables of type bit and bool are stored in a single bit of memory, which means that they can hold only binary, or boolean values.

ISO compliant implementations of C define the domains of all integer data types in a system header file named limits.h, which is accessible by the C compiler. Table 16.2 summarizes these definitions for a typical system.

Variables of type unsigned are stored in the number of bits that is specified in the (required) constant field from the declaration. For instance,

```
unsigned x : 5 = 15;
```

declares a variable named x that is stored in five bits of memory. This declaration also states that the variable is to be initialized to the value 15. As with all variable declarations, an explicit initialization field is optional. The default initial value for all variables is zero. This applies both to scalar variables and to array variables, and it applies to both global and to local variables.

If an attempt is made to assign a value outside the domain of the variable type, the actual value assigned is obtained by a type cast operation that truncates the value to the domain. Information is lost if such a truncation is applied. SPIN will warn if this happens only during random or guided simulation runs.

Table 16.2. Typical Data Ranges

Type	C–Equivalent	limits.h	Typical Range
bit	bit–field	–	0..1
bool	bit–field	–	0..1
byte	unsigned char	CHAR_BIT	0..255
pid	unsigned char	CHAR_BIT	0..255
short	short int	SHRT_MIN..SHRT_MAX	-2^{15}.. $2^{15}-1$
int	int	INT_MIN..INT_MAX	-2^{31}.. $2^{31}-1$

Scope: The scope of a variable declaration is global if it appears outside all proctype or init declarations. The scope of a local variable includes the complete body of a proctype. The declaration itself can be placed anywhere within the proctype or init declaration, provided only that it appears before the first use of the variable. Each separate process has a private copy of all variables that are declared locally within the corresponding proctype or init declaration.

The formal parameters of a proctype are indistinguishable from local variables. These formal parameters are initialized to the values that are specified in a run statement, or they are initialized to zero when the process is instantiated through an active prefix on a proctype declaration.

Examples

The code fragment

```
byte a, b = 2; short c[3] = 3;
```

declares the names a and b as variables of type byte, and c as an array of three variables of type short. Variable a has the default initial value zero. Variable b is initialized to the value 2, and all three elements of array c are initialized to 3.

A variable may also be initialized with an expression, but this is generally not recommended. Note that if global variables are referenced in such initializations, the precise value of such globals may be uncertain. If local variables from the same proctype declaration are referenced in one of the variable declarations, there are some additional dangers that can be caused by the fact the variable declarations can physically appear anywhere in a proctype declaration, but functionally they always act as if they are all moved to the start of the proctype body.

In the following model fragment, for instance, the value that is assigned to variable b in the declaration is 2, and not 4, as might be expected.

```
init {
        byte a = 2;

        a = 4;

        byte b = a;

        printf("b: %d\n", b)
}
```

When a process is instantiated, SPIN first collects all variable declarations from the corresponding proctype declaration, and it then creates and initializes each of these variables, in order of declaration in the proctype, but otherwise before the process itself starts executing. The example code above, therefore, is evaluated as if the declaration of variable b was moved to the start of the proctype declaration, immediately following that of a. Use with caution.

Notes

Each process has a predefined local variable _pid of type pid that holds the process instantiation number. Each model also has a predefined, write−only, global variable _ (underscore) of type int that can be used as a scratch variable, and predefined, read−only, global variables _nr_pr (of type int) and _last (of type pid). See the corresponding manual pages for further details on these variables.

An array of bit, bool, or unsigned variables is stored internally as an array of byte variables. This may affect the behavior of the model if, for instance, the user relies on automatic truncation effects during a verification (an unwise strategy). When the verifier source is generated in verbose mode, SPIN will warn if it encounters such cases.

In the C language, the keywords short and unsigned can be used as a prefix of int. This is not valid in PROMELA.

See Also

_, _last, _pid, arrays, chan, mtype, run, typedef

do

Name

do – repetition construct.

Syntax

do :: sequence [:: sequence] * od

Description

The repetition construct, like all other control–flow constructs, is strictly seen not a statement, but a convenient method to define the structure of the underlying automaton.

A repetition construct has a single start and stop state. Each option sequence within the construct defines outgoing transitions for the start state. The end of each option sequence transfers control back to the start state of the construct, allowing for repeated execution. The stop state of the construct is only reachable via a break statement from within one of its option sequences.

There must be at least one option sequence in each repetition construct. Each option sequence starts with a double–colon. The first statement in each sequence is called its guard. An option can be selected for execution only when its guard statement is executable. If more than one guard statement is executable, one of them will be selected non–deterministically. If none of the guards are executable, the repetition construct as a whole blocks.

A repetition construct as a whole is executable if and only if at least one of its guards is executable.

Examples

The following example defines a cyclic process that non–deterministically increments or decrements a variable named count:

```
byte count;

active proctype counter()
{
        do
        :: count++
        :: count-
        :: (count == 0) ->
                break
        od
}
```

In this example the loop can be broken only when `count` reaches zero. It need not terminate, though, because the other two options always remain unconditionally executable. To force termination, we can modify the program as follows:

```
active proctype counter()
{
	do
	:: count != 0 ->
		if
		:: count++
		:: count--
		fi
	:: else ->
		break
	od
}
```

Notes

The semantics of a PROMELA repetition construct differ from a similar control flow construct tha was included in Dijkstra's seminal proposal for a non−deterministic guarded command language. In Dijkstra's language, the repetition construct is aborted when none of the guards are executable; in PROMELA, execution is merely blocked in this case. In PROMELA, executability is used as the basic mechanism for enforcing process synchronization, and it is not considered to be an error if statements occasionally block. The PROMELA repetition construct also differs from a similar control flow construct in Hoare's classic language CSP. In CSP, send and receive statements cannot appear as guards of an option sequence. In PROMELA, there is no such restriction.

The guard statements in option sequences cannot individually be prefixed by a label, since all option sequences start from the same state (the start state of the construct). If a label is required, it should be placed before the keyword `do`.

See Also

break, else, goto, if, timeout, unless

else

Name

`else` – a system defined condition statement.

Syntax

`else`

Description

The predefined condition statement `else` is intended to be used as a guard (i.e., the first statement) of an option sequence inside selection or repetition constructs.

An `else` condition statement is executable if and only if no other statement within the same process is executable at the same local control state (i.e., process state).

It is an error to define control flow constructs in which more than one `else` may need to be evaluated in a single process state.

Examples

In the first example, the condition statement `else` is equivalent to the regular expression statement `(a < b)`.

```
if
:: a > b -> ...
:: a == b -> ...
:: else -> ...    /* evaluates to: a < b */
fi
```

Note also that round braces are optional around expression statements.

In this example:

```
A: do
   :: if
      :: x > 0 -> x-
      :: else -> break
      fi
   :: else -> x = 10
```

```
od
```

both `else` statements apply to the same control state, which is marked with the label `A` here. To show the ambiguity more clearly, we can rewrite this example also as:

```
A: do
   :: x > 0 -> x--
   :: else -> break
   :: else -> x = 10
   od
```

It is unclear what should happen when `(x < 0)`, and therefore the SPIN parser will reject constructions such as these.

Another construction that the parser will reject is the use of an `else` in combination with an operation on a channel, for instance, as follows:

```
A: if
   :: q?a -> ...
   :: else -> ...
   fi
```

Note that a race condition is built–in to this type of code. How long should the process wait, for instance, before deciding that the message receive operation will not be executable? The problem can be avoided by using message poll operations, for instance, as follows:

```
A: if
   :: atomic { q?[a] -> q?a }
   :: else -> ...
   fi
```

Now the meaning is clear, if the message `a` is present in channel `q` when control reaches the statement that was marked with the label `A`, then that message will be retrieved, otherwise the `else` clause will be selected.

Notes

The semantics as given would in principle also allow for an `else` to be used outside selection or repetition constructs, in a non–branching sequence of statements. The `else` would then be equivalent to a `skip` statement, since it would have no alternatives within the local context. The PROMELA parser, however, will flag such use as an error.

The executability of the `else` statement depends only on local context within a process. The PROMELA semantics for `timeout` can be seen as a global version of `else`. A `timeout` is executable only when no alternative statement within the global context of the system is executable. A `timeout` may not be combined with an `else` in the same selection construct.

See Also

condition, do, false, if, skip, true, timeout, unless

empty

Name

`empty` – predefined, boolean function to test emptiness of a buffered channel.

Syntax

`empty` (name)

Description

`Empty` is a predefined function that takes the name of a channel as an argument and returns true if the number of messages that it currently holds is zero; otherwise it returns false. The expression

```
empty(q)
```

where `q` is a channel name, is equivalent to the expression

```
(len(q) == 0)
```

Examples

```
chan q = [8] of { mtype };

d_step {
        do
        :: q?_
        :: empty(q) -> break
        od;
        skip
}
```

This example shows how the contents of a message channel can be flushed in one indivisible step, without knowing, or storing, the detailed contents of the channel. Note that execution of this code is deterministic. The reason for the `skip` statement at the end is explained in the manual page for `d_step`.

Notes

A call on `empty` can be used as a guard, or it can be used in combination with other conditionals in a boolean expression. The expression in which it appears, though, may not be negated. (The SPIN parser will intercept this.) Another predefined function, `nempty`, can be used when the negated version is needed. The reason for the use of `empty` and `nempty` is to assist SPIN's partial order reduction strategy during verification.

If predefined functions such as `empty` and `nempty` are used in the symbol definitions of an LTL formula, they may unintentionally appear under a negation sign in the generated automaton, which can then trigger a surprising syntax error from SPIN. The easiest way to remedy such a problem, if it occurs, is to revise the generated `never` claim automaton directly, and replace every occurrence of `!empty()` with `nempty()` and every occurrence of `!nempty()` with `empty()`.

See Also

_, condition, full, ltl len, nempty, nfull

enabled

Name

`enabled` – predefined boolean function for testing the enabledness of a process from within a `never` claim.

Syntax

`enabled` (any_expr)

Description

This predefined function can only be used inside a `never` claim, or equivalently in the symbol definition for an LTL formula.

Given the instantiation number of an active process, the function returns true if the process has at least one executable statement in its current control state, and false otherwise. When given the instantiation number of a non–existing process, the function always returns false.

In every global state where `enabled(p)` returns true, the process with instantiation number `p` has at least one executable statement. Of course, the executability status of that process can change after the next execution step is taken in the system, which may or may not be from process `p`.

Examples

The following `never` claim attempts to match executions in which the process with instantiation number one remains enabled infinitely long without ever executing.

```
never {
accept:
        do
        :: _last != 1 && enabled(1)
        od
}
```

Notes

The use of this function is incompatible with SPIN's partial order reduction strategy, and can therefore increase the computational requirements of a verification.

See Also

_last, _pid, ltl, never, pc_value, run

end

Name

`end` – label–name prefix for marking valid termination states.

Syntax

`end [a-zA-Z0-9_] *`: stmnt

Description

An end–state label is any label name that starts with the three–character sequence `end`. End–state labels can be used in `proctype`, `trace`, and `notrace` declarations.

When used in a `proctype` declaration, the end–state label marks a local control state that is acceptable as a valid termination point for all instantiations of that `proctype`.

If used in an event `trace` definition, the end–state label marks a global control state that corresponds to a valid termination point for the system as a whole.

If used in an event `notrace` definition, though, the normal meaning reverses: the event trace is now considered to have been completely matched when the `end` state is reached, thus signifying an error condition, rather than normal system termination.

End–state labels have no special meaning when used in `never` claims.

Examples

In the following example the end–state label defines that the expected termination point of the process is at the start of the loop.

```
active proctype dijkstra()
{
end:    do
        :: sema!p -> sema?v
        od
}
```

It will now be flagged as an invalid end–state error if the system that contains this `proctype` declaration can terminate in a state where the process of type `dijkstra` remains at the control state that exists just after the arrow symbol.

Notes

It is considered an invalid end–state error if a system can terminate in a state where not all active processes are either at the end of their code (i.e., at the closing curly brace of their `proctype` declarations) or at a local state that is marked with and end–state label.

If the run–time option `-q` is used with the compiled verifier, an additional constraint is applied for a state to be considered a valid `end` state: all message channels must then also be empty.

See Also

accept, labels, notrace, progress, trace

eval

Name

`eval` – predefined unary function to turn an expression into a constant.

Syntax

`eval` (any_expr)

Description

The intended use of `eval` is in `receive` statements to force a match of a message field with the current value of a local or global variable. Normally, such a match can only be forced by specifying a constant. If a variable name is used directly, without the `eval` function, the variable would be assigned the value from the corresponding message field, instead of serving as a match of values.

Examples

In the following example the two receive operations are only executable if the precise values specified were sent to channel `q`: first an `ack` and then a `msg`.

```
mtype = { msg, ack, other };
chan q = [4] of { mtype };

mtype x;

x = ack; q?eval(x)      /* same as: q?ack */
x = msg; q?eval(x)      /* same as: q?msg */
```

Without the `eval` function, writing simply

```
q?x
```

would mean that whatever value was sent to the channel (e.g., the value `other`) would be assigned to `x` when the receive operation is executed.

Notes

Any expression can be used as an argument to the `eval` function. The result of the evaluation of the expression is then used as if it were a constant value.

This mechanism can also be used to specify a conditional rendezvous operation, for instance by using the value true in the sender and using a conditional expression with an `eval` function at the receiver; see also the manual page for conditional expressions.

See Also

cond_expr, condition, poll, receive

false

Name

`false` – predefined boolean constant.

Syntax

`false`

Description

The keyword `false` is a synonym of the constant value zero (0), and can be used in any context. If it is used as a stand−alone condition statement, it will block system execution as if it were a halt instruction.

Notes

Because they are intercepted in the lexical analyzer as meta terms, `false`, `true`, and `skip` do not show up as such in error traces. They will appear as their numeric equivalents `(0)` or `(1)`.

See

condition, skip, true

float

Name

`float` – floating point numbers.

Description

There are no floating point numbers in basic PROMELA because the purpose the language is to encourage abstraction from the computational aspects of a distributed application while focusing on the verification of process interaction, synchronization, and coordination.

Consider, for instance, the verification of a sequential C procedure that computes square roots. Exhaustive state–based verification would not be the best approach to verify this procedure. In a verification model, it often suffices to abstract this type of procedure into a simple two–state demon that non–deterministically decides to give either a correct or incorrect answer. The following example illustrates this approach.

```
mtype = { number, correct, incorrect };
chan sqrt = [0] of { mtype, chan };

active proctype sqrt_server()
{
        do
        :: sqrt?number(answer) ->
                /* abstract from local computations */
                if
                :: answer!correct
                :: answer!incorrect
                fi
        od
}

active proctype user()
{       chan me = [0] of { mtype };

        do
        :: sqrt!number(me);
                if
                :: me?correct -> break
                :: me?incorrect ->
                        ...
                fi;
        od;
        ...
}
```

The predefined data types from PROMELA are a compromise between notational convenience and modest constraints that can facilitate the construction of tractable verification models. The largest numeric quantity that can be manipulated is, for instance, a 32−bit integer number. The number of different values that even one single integer variable can record, for instance, when used as a simple counter, is already well beyond the scope of a state−based model checker. Even integer quantities, therefore, are to be treated with some suspicion in verification models, and can very often be replaced advantageously with `byte` or `bit` variables.

Notes

In the newer versions of SPIN, there is an indirect way to use external data types, such as `float`, via embedded code and embedded declarations. The burden on the user to find abstractions can thus be lightened, in return for a potential increase in verification complexity. When using embedded C code, the user can decide separately if some or all of the embedded data objects should be treated as part of the state descriptor in the verification model, with the use of `c_state` or `c_track` declarators. See Chapter 17 for a detailed description.

See Also

c_code, c_decl, c_expr, datatypes

full

Name

`full` – predefined, boolean function to test fullness of a channel.

Syntax

`full` (varref)

Description

`Full` is a predefined function that takes the name of a channel as an argument and returns true if that channel currently contains its maximum number of messages, and otherwise it returns false. It is equivalent to the expression

```
(len(q) == QSZ)
```

where `q` is the channel name, and `QSZ` is the message capacity of the channel.

This function can only be applied to buffered channels. The value returned for rendezvous channels would always be false, since a rendezvous channel cannot store messages.

Examples

```
chan q = [8] of { byte };
byte one_more = 0;

do
:: q!one_more; one_more++         /* send messages */
:: full(q) -> break            /* until full    */
od;
assert(len(q) == 8)
```

Notes

`Full` can be used as a guard, by itself, or it can be used as a general boolean function in expressions. It can, however, not be negated (for an explanation see also the manual page for `empty`).

If predefined functions such as `full`, or `nfull` are used in the symbol definitions of an LTL formula, they may unintentionally appear under a negation sign in the generated automaton, which can then trigger a surprising syntax error from SPIN.

See Also

condition, empty, len, ltl, nempty, nfull

goto

Name

`goto` – unconditional jump to a labeled statement.

Syntax

`goto` name

Description

The `goto` is normally not executed, but is used by the parser to determine the target control state for the immediately preceding statement; see also the manual page for `break`. The target state is identified by the label name and must be unique within the surrounding `proctype` declaration or `never` claim.

In cases where there is no immediately preceding statement, for instance, when the `goto` appears as a guard in an option of a selection or repetition structure, the `goto` is executed as if it were a `skip`, taking one execution step to reach the labeled state.

Examples

The following program fragment defines two control states, labeled by `L1` and `L2`:

```
L1:     if
        :: a != b -> goto L1
        :: a == b -> goto L2
        fi;
L2:     ...
```

If the values of variables `a` and `b` are equal, control moves from `L1` to `L2` immediately following the execution of condition statement `a == b`. If the values are unequal, control returns to `L1` immediately following the execution (evaluation) of `a != b`. The statement is therefore equivalent to

```
L1:     do
        :: a != b
        :: a == b -> break
        od;
L2:
```

and could also be written more efficiently in PROMELA as simply:

```
L1:     a == b;
L2:
```

Note that the last version makes use of the capability of PROMELA to synchronize on a standalone condition statement.

Notes

It is an error if no target for the `goto` is defined within the surrounding `proctype` or `never` claim declaration.

See Also

break, condition, labels

hidden

Name

`hidden` – for excluding data from the state descriptor during verification.

Syntax

`hidden` typename ivar

Description

The keyword `hidden` can be used to prefix the declaration of any variable to exclude the value of that variable from the definition of the global system state. The addition of this prefix can affect only the verification process, by potentially changing the outcome of state matching operations.

Examples

```
hidden byte a;
hidden short p[3];
```

Notes

The prefix should only be used for write−only scratch variables. Alternatively, the predefined write−only scratch variable _ (underscore) can always be used instead of a `hidden` integer variable.

It is safe to use `hidden` variables as pseudo−local variables inside `d_step` sequences, provided that they are not referenced anywhere outside that sequence.

See Also

_, datatypes, local, show

hierarchy

Name

`hierarchy` – for defining layered systems.

Description

There is no mechanism for defining a hierarchically layered system in PROMELA, nor is there a good excuse to justify this omission. At present, the only structuring principles supported in PROMELA are `proctype`s, `inline`s, and `macros`.

See Also

inline, macros, proctype, procedures

if

Name

`if` – selection construct.

Syntax

`if` :: sequence [:: sequence] * `fi`

Description

The selection construct, like all other control–flow constructs, is strictly seen not a statement, but a convenient method to define the structure of the underlying automaton. Each selection construct has a unique start and stop state. Each option sequence within the construct defines outgoing transitions for the start state, leading to the stop state. There can be one or more option sequences. By default, the end of each option sequence leads to the control state that follows the construct.

There must be at least one option sequence in each selection construct. Each option sequence starts with a double–colon. The first statement in each sequence is called its guard. An option can be selected for execution only when its guard statement is executable. If more than one guard statement is executable, one of them will be selected non–deterministically. If none of the guards are executable, the selection construct as a whole blocks.

The selection construct as a whole is executable if and only if at least one of its guards is executable.

Examples

Using the relative values of two variables `a` and `b` to choose between two options, we can write

```
if
:: (a != b) -> ...
:: (a == b) -> ...
fi
```

This selection structure contains two option sequences, each preceded by a double colon. Only one sequence from the list will be executed. A sequence can be selected only if its guard statement is executable (the first statement). In the example the two guards are mutually exclusive, but this is not required.

The guards from a selection structure cannot be prefixed by labels individually. These guards really define the outgoing transitions of a single control state, and therefore any label on one guard is really a label on the source state for all guards belonging on the selection construct itself (cf.label `L0` in the next example). It is tempting to circumvent this rule and try to label a guard by inserting a `skip` in front of it, for instance, as follows:

```
L0:     if
        :: skip;
L1:             (a != b) -> ...
        :: (a == b) -> ...
        fi;
```

But note that this modification alters the meaning of the selection from a choice between `(a != b)` and `(a == b)`, to a choice between `skip` (which is the same as `(1)` or `true`) and `(a == b)`. The addition of the `skip` statement also adds an extra intermediate state, immediately followin the `skip` statement itself.

Notes

The semantics of a PROMELA selection construct differ from similar control flow constructs in Hoare's language CSP, and in Dijkstra's earlier definition of a non−deterministic guarded command language. In Dijkstra's definition, the selection construct is aborted when none of the guards is executable. In PROMELA, execution blocks in this case. In PROMELA, executability is used as the basic means to enforce process synchronization, and it is not considered to be an error if statements block temporarily. Another difference with CSP is that in PROMELA there is no restriction on the type of statement that can be used as a guard of an option sequence. Any type of statement can be used as a guard, including assignments, and send or receive operations.

See Also

do, else, goto, timeout

init

Name

init – for declaring an initial process.

Syntax

init { sequence }

Description

The init keyword is used to declare the behavior of a process that is active in the initial system state.

An init process has no parameters, and no additional copies of the process can be created (that is, the keyword cannot be used as an argument to the run operator).

Active processes can be differentiated from each other by the value of their process instantiation number, which is available in the predefined local variable _pid. Active processes are always instantiated in the order in which they appear in the model, so that the first such process (whether it is declared as an active process or as an init process) will receive the lowest instantiation number, which is zero.

Examples

The smallest possible PROMELA model is:

```
init { skip }
```

where skip is PROMELA's null statement, or perhaps more usefully

```
init { printf("hello world\n") }
```

The init process is most commonly used to initialize global variables, and to instantiate other processes, through the use of the run operator, before system execution starts. Any process, not just the init process, can do so, though.

It is convention to instantiate groups of processes within atomic sequences, to make sure that their execution begins at the same instant. For instance, in the leader election example, included as a test case in the SPIN distribution, the initial process is used to start up N copies of the proctype node. Each new instance of the `proctype` is given different parameters, which in this case consist of two channel names and an indentifying number. The node `proctype` is then of the form:

```
proctype node(chan in, chan out, byte mynumber)
{       ...
}
```

and the `init` process is structured as follows.

```
init {
   byte proc;
   atomic {
     proc = 1;
     do
     :: proc <= N ->
        run node (q[proc-1],q[proc%N],(N+I-proc)%N+1);
        proc++
     :: proc > N ->
        break
     od
   }
}
```

After the instantiation, the initial process terminates.

A process in PROMELA, however, cannot die and be removed from the system until all its children have died first. That is, PROMELA processes can only die in reverse order of creation (in stack order). This means that if an `init` process is used to create all other processes in the system, the `init` process itself will continue to exist, and take up memory, as long as the system exists. Systems in which all processes can be instantiated with `active` prefixes, instead of through the intermediacy of an `init` process, can therefore often be verified more efficiently. The following code fragment illustrates an alternative initialization for the `leader` election protocol, avoiding the use of an `init` process:

```
active [N] proctype node ()
{       chan in  = q[_pid];
        chan out = q[(_pid+1)%N];
        byte mynumber = (N+I-(_pid+1))%N+1;
        ...
}
```

Because no parameter values can be passed to an `active` process declaration, the parameters are now replaced with local variables.

Notes

The `init` keyword has become largely redundant with the addition of the `active` prefix for `proctype` declarations.

See Also

_pid, active, proctype, run, skip

inline

Name

`inline` – a stylized version of a macro.

Syntax

`inline` name ([arg_lst]) { sequence }

Description

An `inline` definition must appear before its first use, and must always be defined globally, that is, at the same level as a `proctype` declaration. An `inline` definition works much like a preprocessor macro, in the sense that it just defines a replacement text for a symbolic name, possibly with parameters. It does not define a new variable scope. The body of an `inline` is directly pasted into the body of a `proctype` at each point of invocation. An invocation (an `inline` call) is performed with a syntax that is similar to a procedure call in C, but, like a macro, a PROMELA `inline` cannot return a value to the caller.

An `inline` call may appear anywhere a stand–alone PROMELA statement can appear. This means that, unlike a macro call, an `inline` call cannot appear in a parameter list of the `run` operator, and it cannot be used as an operand in an expression. It also cannot be used on the left– or right–hand side of an assignment statement.

The parameters to an `inline` definition are typically names of variables.

An `inline` definition may itself contain other `inline` calls, but it may not call itself recursively.

Examples

The following example illustrates the use of `inline` definitions in a version of the alternating bit protocol.

```
mtype = { msg0, msg1, ack0, ack1 };

chan sender = [1] of { mtype };
chan receiver = [1] of { mtype };

inline recv(cur_msg, cur_ack, lst_msg, lst_ack)
{
        do
        :: receiver?cur_msg ->
                sender!cur_ack; break /* accept */
        :: receiver?lst_msg ->
                sender!lst_ack
        od;
```

```
}

inline phase(msg, good_ack, bad_ack)
{
        do
        :: sender?good_ack -> break
        :: sender?bad_ack
        :: timeout ->
                if
                :: receiver!msg;
                :: skip /* lose message */
                fi;
        od
}

active proctype Sender()
{
        do
        :: phase(msg1, ack1, ack0);
           phase(msg0, ack0, ack1)
        od
}

active proctype Receiver()
{
        do
        :: recv(msg1, ack1, msg0, ack0);
           recv(msg0, ack0, msg1, ack1)
        od
}
```

In simulations, line number references are preserved and will point to the source line inside the `inline` definition where possible. In some cases, in the example for instance at the start of the `Sender` and the `Receiver` process, the control point is inside the `proctype` body and not yet inside the `inline`.

Notes

The PROMELA scope rules for variables are not affected by `inline` definitions. If, for instance, the body of an `inline` contains variable declarations, their scope would be the same as if they were declared outside the `inline`, at the point of invocation. The scope of such variables is the entire body of the `proctype` in which the invocation appears. If such an `inline` would be invoked in two different places within the same `proctype`, the declaration would also appear twice, and a syntax error would result.

See Also

comments, macros

labels

Name

`label` – to identify a unique control state in a `proctype` declaration.

Syntax

name : stmnt

Description

Any statement or control–flow construct can be preceded by a label. The label can, but need not, be used as a destination of a `goto` or can be used in a remote reference inside a `never` claim. Label names must be unique within the surrounding `proctype`, `trace`, `notrace`, or `never` claim declaration.

A label always prefixes a statement, and thereby uniquely identifies a control state in a transition system, that is, the source state of the transition that corresponds to the labeled statement.

Any number of labels can be attached to a single statement.

Examples

The following `proctype` declaration translates into a transition system with precisely three local process states: initial state `S1`, state `S2` in between the send and the receive, and the (unreachable) final state `S3`, immediately following the repetition construct.

```
active proctype dijkstra()
{
S0:
S1:     do
        :: q!p ->
S2:             q?v
        :: true
        od
/* S3 */
}
```

The first state has two labels: `S0` and `S1`. This state has two outgoing transitions: one corresponding to the send statement `q!p`, and one corresponding to the condition statement `true`. Observe carefully that there is no separate control state at the start of each guard in a selection or repetition construct. Both guards share the same start state `S1`.

Notes

A label name can be any alphanumeric character string, with the exception that the first character in the label name may not be a digit or an underscore symbol.

The guard statement in a selection or repetition construct cannot be prefixed by a label individually; see the manual page for `if` and `do` for details.

There are three types of labels with special meaning, see the manual pages named `accept`, `end`, and `progress`.

See Also

accept, do, end, if, goto, progress, remoterefs

len

Name

`len` – predefined, integer function to determine the number of messages that is stored in a buffered channel.

Syntax

`len` (varref)

Description

A predefined function that takes the name of a channel as an argument and returns the number of messages that it currently holds.

Examples

```
#define QSZ     4

chan q = [QSZ] of { mtype, short };

len(q) > 0    /* same as nempty(q) */
len(q) ==   0    /* same as empty(q) */
len(q) ==   QSZ  /* same as full(q) */
len(q) < QSZ  /* same as nfull(q) */
```

Notes

When possible, it is always better to use the predefined, boolean functions `empty`, `nempty`, `full`, and `nfull`, since these define special cases that can be exploited in SPIN's partial order reduction algorithm during verification.

If `len` is used stand–alone as a condition statement, it will block execution until the channel is non–empty.

See Also

chan, condition, empty, full, nempty, nfull, xr, xs

local

Name

`local` – prefix on global variable declarations to assert exclusive use by a single process.

Syntax

`local` typename ivar

Description

The keyword `local` can be used to prefix the declaration of any global variable. It persuades the partial order reduction algorithm in the model checker to treat the variable as if it were declared local to a single process, yet by being declared global it can freely be used in LTL formulae and in `never` claims.

The addition of this prefix can increase the effect of partial order reduction during verification, and lower verification complexity.

Examples

```
local byte a;
local short p[3];
```

Notes

If a variable marked as `local` is in fact accessed by more than one process, the partial order reduction may become invalid and the result of a verification incomplete. Such violations are not detected by the verifier.

See Also

_, datatypes, hidden, ltl, never, show

ltl

Name

ltl – linear time temporal logic formulae for specifying correctness requirements.

Syntax

Grammar:

ltl ::= opd | (ltl) | ltl binop ltl | unop ltl

Operands (opd):

true, false, and user–defined names starting with a lower–case letter

Unary Operators (unop):
Binary Operators (binop):

Description

SPIN can translate LTL formulae into PROMELA never claims with command line option -f. The never claim that is generated encodes the Büuchi acceptance conditions from the LTL formula. Formally, any ω–run that satisfies the LTL formula is guaranteed to correspond to an accepting run of the never claim.

The operands of an LTL formula are often one–character symbols, such as p, q, r, but they can also be symbolic names, provided that they start with a lowercase character, to avoid confusion with some of the temporal operators which are in uppercase. The names or symbols must be defined to represent boolean expressions on global variables from the model. The names or symbols are normally defined with macro definitions.

All binary operators are left–associative. Parentheses can be used to override this default. Note that implication and equivalence are not temporal but logical operators (see Chapter 6).

Examples

Some examples of valid LTL formulae follow, as they would be passed in command–line arguments to SPIN for translation into never claims. Each formula passed to SPIN has to be quoted. We use single quotes in all examples in this book, which will work correctly on most systems (including UNIX systems and Windows systems with the cygwin toolset). On some systems double quotes can also be used.

```
spin -f '[] p'
spin -f '!( <> !q )'
spin -f 'p U q'
```

```
spin -f 'p U ([] (q U r))'
```

The conditions `p`, `q`, and `r` can be defined with macros, for instance as:

```
#define p      (a > b)
#define q      (len(q) < 5)
#define r      (root@Label)
```

elsewhere in the PROMELA model. It is prudent to always enclose these macro definitions in round braces to avoid misinterpretation of the precedence rules on any operators in the context where the names end up being used in the final `never` claim. The variables `a` and `b`, the channel name `q`, and the `proctype` name `root` from the preceding example, must be globally declared.

Notes

If the SPIN sources are compiled with the preprocessor directive `-DNXT`, the set of temporal operators is extended with one additional unary operator: `X` (next). The `X` operator asserts the truth of the subformula that follows it for the next system state that is reached. The use of this operator can void the validity of the partial order reduction algorithm that is used in SPIN, if it changes the stutter invariance of an LTL formula. For the partial order reduction strategy to be valid, only LTL properties that are stutter invariant can be used. Every LTL property that does not contain the `X` operator is guaranteed to satisfy the required property. A property that is not stutter invariant can still be checked, but only without the application of partial order reduction.

An alternative converter for LTL formulae, that can often produce smaller automata, is the tool `ltl2ba`, see p. 145.

See Also

condition, macros, never, notrace, remoterefs, trace

macros

Name

macros and include files – preprocessing support.

Syntax

```
#define name token-string
#define name (arg, ..., arg) token-string
#ifdef name
#ifndef name
#if constant-expression
#else
#endif
#undef name
#include "filename"
```

Description

PROMELA source text is always processed by the C preprocessor, conventionally named `cpp`, before being parsed by SPIN. When properly compiled, SPIN has a link to the C preprocessor built–in, so that this first processing step becomes invisible to the user. If a problem arises, though, or if a different preprocessor should be used, SPIN recognizes an option `-Pxxx` that allows one to define a full pathname for an alternative preprocessor. The only requirement is that this preprocessor should read standard input and write its result on standard output.

Examples

```
#include "promela_model"

#define p       (a>b)

never { /* <>!p */
        do
        :: !p -> assert(false)
        :: else /* else ignore */
        od
}
```

It is always wise to put braces around the replacement text in the macro–definitions to make sure the precedence of operator evaluation is preserved when a macro name is used in a different context, for example, within a composite boolean expression.

Notes

The details of the working of the preprocessor can be system dependent. For the specifics, consult the manual pages for `cpp` that came with the C compiler that is installed on your system.

On PCs, if no macros with more than one parameter appear in the model, and no extra compiler directives are defined on the command line, SPIN will use a simple built–in version of the C preprocessor to bypass the call on the external program. When needed, this call can be suppressed by adding a dummy compiler directive to the command line, as in:

```
$ spin -DDUMMY -a model
```

The call could also be suppressed by adding a dummy macro definition with more than one parameter to the model itself, as in:

```
#define dummy(a,b)     (a+b)
```

The preprocessor that is used can be modified in several ways. The default preprocessor, for instance, can be set to `m4` by recompiling SPIN itself with the compiler directive `-DCPP=/bin/m4`. The choice of preprocessor can also be changed on the command line, for instance, by invoking SPIN as:

```
$ spin -P/bin/m4    model
```

Extra definitions can be passed to the preprocessor from the command line, as in:

```
$ spin -E-I/usr/greg -DMAX=5 -UXAM model
```

which has the same effect as adding the following two definitions at the start of the model:

```
#define MAX     5
#undef  XAM
```

as well as passing the additional directive `-I/usr/greg` to the preprocessor, which results in the addition of directory `/usr/greg` to the list of directories that the preprocessor will search for include files.

See Also

comments, never

mtype

Name

mtype – for defining symbolic names of numeric constants.

Syntax

mtype [=] { name [, name]* }

mtype name [= mtype_name]

mtype name '[' const ']' [= mtype_name]

Description

An mtype declaration allows for the introduction of symbolic names for constant values. There can be multiple mtype declarations in a verification model. If multiple declarations are given, they are equivalent to a single mtype declaration that contains the concatenation of all separate lists of symbolic names.

If one or more mtype declarations are present, the keyword mtype can be used as a data type, to introduce variables that obtain their values from the range of symbolic names that was declared. This data type can also be used inside chan declarations, for specifying the type of message fields.

Examples

The declaration

```
mtype = { ack, nak, err, next, accept }
```

is functionally equivalent to the sequence of macro definitions:

```
#define ack      5
#define nak      4
#define err      3
#define next     2
#define accept   1
```

Note that the symbols are numbered in the reverse order of their definition in the mtype declarations, and that the lowest number assigned is one, not zero.

If multiple `mtype` declarations appear in the model, each new set of symbols is prepended to the previously defined set, which can make the final internal numbering of the symbols somewhat less predictable.

The convention is to place an assignment operator in between the keyword `mtype` and the list of symbolic names that follows, but this is not required.

The symbolic names are preserved in tracebacks and error reports for all data that is explicitly declared with data type `mtype`.

In this example:

```
mtype a; mtype p[4] = nak;
chan q = [4] of { mtype, byte, short, mtype };
```

the `mtype` variable `a` is not initialized. It will by default be initialized to zero, which is outside the range of possible `mtype` values (identifying the variable as uninitialized). All four elements of array `p` are initialized to the symbolic name `nak`. Channel `q`, finally, has a channel initializer that declares the type of the first and last field in each message to be of type `mtype`.

Notes

Variables of type `mtype` are stored in a variable of type `unsigned char` in their C equivalent. Therefore, there can be at most 255 distinct symbolic names in an `mtype` declaration.

The utility function `printm` can be used to print the symbolic name of a single `mtype` variable. Alternatively, in random or guided simulations with SPIN, the name can be printed with the special `printf` conversion character sequence `%e`. The following two lines, for instance, both print the name `nak` (without spaces, linefeeds, or any other decoration):

```
mtype = { ack, nak, err, next, accept }

init {
        mtype x = nak;

        printm(x);
        printf("%e", x)
}
```

The `printm` form is prefered, since it will also work when error traces are reproduced with the verifier, for models with embedded C code.

See Also

datatypes, printf, printm

nempty

Name

`nempty` – predefined, boolean function to test emptiness of a channel.

Syntax

`nempty` (varref)

Description

The expression `nempty(q)`, with `q` a channel name, is equivalent to the expression

```
(len(q) != 0)
```

where `q` is a channel name. The PROMELA grammar prohibits this from being written as `!empty(q)`.

Using `nempty` instead of its equivalents can preserve the validity of reductions that are applied during verifications, especially in combination with the use of `xr` and `xs` channel assertions.

Notes

Note that if predefined functions such as `empty`, `nempty`, `full`, and `nfull` are used in macro definitions used for propositional symbols in LTL formulae, they may well unintentionally appear under a negation sign, which will trigger syntax errors from SPIN.

See Also

condition, empty, full, len, ltl, nfull, xr, xs

never

Name

`never` – declaration of a temporal claim.

Syntax

`never {` sequence `}`

Description

A `never` claim can be used to define system behavior that, for whatever reason, is of special interest. It is most commonly used to specify behavior that should never happen. The claim is defined as a series of propositions, or boolean expressions, on the system state that must become true in the sequence specified for the behavior of interest to be matched.

A `never` claim can be used to match either finite or infinite behaviors. Finite behavior is matched if the claim can reach its final state (that is, its closing curly brace). Infinite behavior is matched if the claim permits an ω–acceptance cycle. `Never` claims, therefore, can be used to verify both safety and liveness properties of a system.

Almost all PROMELA language constructs can be used inside a claim declaration. The only exceptions are those statements that can have a side effect on the system state. This means that a `never` claim may not contain assignment or message passing statements. Side effect free channel poll operations, and arbitrary condition statements are allowed.

`Never` claims can either be written by hand or they can be generated mechanically from LTL formula, see the manual page for `ltl`.

There is a small number of predefined variables and functions that may only be used inside `never` claims. They are defined in separate manual pages, named `_last, enabled`, `np_`, `pc_value`, and `remoterefs`.

Examples

In effect, when a `never` claim is present, the system and the claim execute in lockstep. That is, we can think of system execution as always consisting of a pair of transitions: one in the claim and one in the system, with the second transition coming from any one of the active processes. The claim automaton always executes first. If the claim automaton does not have any executable transitions, no further move is possible, and the search along this path stops. The search will then backtrack so that other executions can be explored.

This means that we can easily use a `never` claim to define a search restriction; we do not necessarily have to use the claim only for the specification of correctness properties. For example, the claim

```
never /* [] p */
{
	do
	:: p
	od
}
```

would restrict system behavior to those states where property `p` holds.

We can also use a search restriction in combination with an LTL property. To prove, for instance, that the model satisfies LTL property `<>q`, we can use the `never` claim that is generated with the SPIN command (using the negation of the property):

```
$ spin -f '!<> q'
```

Using the generated claim in a verification run can help us find counterexamples to the property. If we want to exclude non−progress behaviors from the search for errors, we can extend the LTL formula with the corresponding restriction, as follows:

```
$ spin -f '([]<> !np_) -> (!<> q)'
```

Alternatively, if we wanted to restrict the search to only non −progress behaviors, we can negate the precondition and write:

```
$ spin -f '(<>[] np_) -> (!<> q)'
```

The claim automaton must be able to make its first transition, starting in its initial claim state, from the global initial system state of the model. This rule can sometimes have unexpected consequences, especially when remote referencing operations are used. Consider, for instance, the following model:[1]

[1] The example is from Rob Gerth.

```
byte aap;

proctype noot()
{
mies:	skip
}

init {
	aap = run noot()
```

```
}
```

with the `never` claim defined as follows:

```
never {
        do
        :: noot[aap]@mies -> break
        :: else
        od
}
```

The intent of this claim is to say that the process of type `noot`, with pid `aap`, cannot ever reach its state labeled `mies`. If this happened, the claim would reach its final state, and a violation would be flagged by the verifier. We can predict that this property is not satisfied, and when we run the verifier it will indeed report a counterexample, but the counterexample is created for a different reason.

In the initial system state the `never` claim is evaluated for the first time. In that state only the `init` process exists. To evaluate expression `noot[aap]@mies` the value of variable `aap` is determined, and it is found to be zero (since the variable was not assigned to yet, and still has its default initial value). The process with pid zero is the `init` process, which happens to be in its first state. The label `mies` also points to the first state, but of a process that has not been created yet. Accidentally, therefore, the evaluation of the remote reference expression yields true, and the claim terminates, triggering an error report. The simulator, finally, on replaying the error trail, will reveal the true nature of this error in the evaluation of the remote reference.

A correct version of the claim can be written as follows:

```
never {
        true;
        do
        :: noot[aap]@mies -> break
        :: else
        od
}
```

In this version we made sure that the remote reference expression is not evaluated until the process that is referred to exists (that is, after the first execution step in the `init` process is completed).

Note that it is not possible to shortcut this method by attempting the global declaration:

```
byte aap = run noot(); /* an invalid initializer */
```

In this case, with only one process of type `noot`, we can also avoid using variable `aap` by using the shorter remote reference:

```
noot@mies
```

To translate an LTL formula into a `never` claim, we have to consider first whether the formula expresses a positive or a negative property. A positive property expresses a good behavior that we would like our system to have. A negative property expresses a bad behavior that we claim the system does not have. A `never` claim is normally only used to formalize negative properties (behaviors that should never happen), which means that positive properties must be negated before they are translated into a claim.

Suppose that the LTL formula `<>[]p`, with `p` a boolean expression, expresses a negative claim (that is, it is considered a correctness violation if there exists any execution sequence in which eventually `p` can remain true infinitely long). This can be written in a `never` claim as:

```
never { /* <>[]p */
        do
        :: true       /* after an arbitrarily long prefix */
        :: p -> break                /* p becomes true */
        od;
accept: do
        :: p            /* and remains true forever after */
        od
}
```

Note that in this case the claim does not terminate and also does not necessarily match all system behaviors. It is sufficient if it precisely captures all violations of our correctness requirement, and no more.

If the LTL formula expressed a positive property, we first have to invert it to the corresponding negative property. For instance, if we claim that immediately from the initial state forward the value of `p` remains true, the negation of that property is: `![]p` which can be translated into a `never` claim. The requirement says that it is a violation if `p` does not always remain true.

```
never { /* ![]p = <>!p*/
        do
        :: true
        :: !p -> break
        od
}
```

In this specification, we have used the implicit match of a claim upon reaching the final state of the automaton. Since the first violation of the property suffices to disprove it, we could also have written:

```
never {
        do
        :: !p -> break
        :: else
```

```
        od
}
```

or, if we abandon the correspondence with LTL and Büchi automata for a moment, even more tersely as:

```
never { do :: assert(p) od }
```

Notes

It is good practice to confine the use of `accept` labels to `never` claims. SPIN automatically generates the `accept` labels within the claim when it generates claims from LTL formulae on run−time option `-f`.

The behavior specified in a `never` claim is matched if the claim can terminate, that is, if execution can reach the closing curly brace of the claim body. In terms of Büchi acceptance, this means that in a search for liveness properties, the final state of the claim is interpreted as the implicit acceptance cycle:

```
accept_all: do :: true od
```

The dummy claim

```
never {
        true
}
```

therefore always matches, and reports a violation, after precisely one execution step of the system. If a `never` claim contains no `accept` labels, then a search for cycles with run−time option `-a` is unnecessary and the claim can be proven or disproven with a simple search for safety properties. When the verifier is used in breadth−first search mode, only safety properties can be proven, including those expressed by `never` claims.

See Also

last, accept, assert, enabled, ltl, notrace, np, pc_value, poll, progress, remoterefs, trace

nfull

Name

`nfull` – predefined, boolean function to test fullness of a channel.

Syntax

`nfull` (varref)

Description

The expression `nfull(q)` is equivalent to the expression

```
(len(q) < QSZ)
```

where `q` is a channel name, and `QSZ` the capacity of this channel. The PROMELA grammar prohibits the same from being written as `!full(q)`.

Using `nfull` instead of its equivalents can preserve the validity of reductions that are applied during verifications, especially in combination with the use of `xr` and `xs` channel assertions.

Notes

Note that if predefined functions such as `empty`, `nempty`, `full`, and `nfull` are used in macro definitions used for propositional symbols in LTL formulae, they may well unintentionally appear under a negation sign, which will trigger syntax errors from SPIN.

See Also

condition, empty, full, len, ltl, nempty, xr, xs

np_

Name

np_ – a global, predefined, read–only boolean variable.

Syntax

np_

Description

This global predefined, read–only variable is defined to be true in all global system states that are not explicitly marked as progress states, and is false in all other states. The system is in a progress state if at least one active process is at a local control state that was marked with a user–defined progress label, or if the current global system state is marked by a progress label in an event trace definition.

The np_ variable is meant to be used exclusively inside never claims, to define system properties.

Examples

The following non–deterministic never claim accepts all non–progress cycles:

```
never { /* <>[] np_ */
        do
        :: true
        :: np_ -> break
        od;
accept: do
        :: np_
        od
}
```

This claim is identical to the one that the verifier generates, and automatically adds to the model, when the verifier source is compiled with the directive -DNP, as in:

```
$ cc -DNP -o pan pan.c
```

Note that the claim automaton allows for an arbitrary, finite–length prefix of the computation where either progress or non–progress states can occur. The claim automaton can move to its accepting state only when the system is in a non–progress state, and it can only stay there infinitely long if the system can indefinitely remain in non−progress states only.

See Also

condition, ltl, never, progress

pc_value

Name

`pc_value` – a predefined, integer function for use in `never` claims.

Syntax

`pc_value` (any_expr)

Description

The call `pc_value(x)` returns the current control state (an integer) of the process with instantiation number `x`. The correspondence between the state numbers reported by `pc_value` and statements or line numbers in the PROMELA source can be checked with run–time option `-d` on the verifiers generated by SPIN, as in:

```
$ spin -a model.pml
$ cc -o pan pan.c
$ ./pan -d
...
```

The use of this function is restricted to `never` claims.

Examples

```
never {
        do
        :: pc_value(1) <= pc_value(2)
        && pc_value(2) <= pc_value(3)
        && pc_value(3) <= pc_value(4)
        && pc_value(4) <= pc_value(5)
        od
}
```

This claim is a flawed attempt to enforce a symmetry reduction among five processes. This particular attempt is flawed in that it does not necessarily preserve the correctness properties of the system being verified. See also the discussion in Chapter 4, p. 94.)

Notes

As the example indicates, this function is primarily supported for experimental use, and may not survive in future revisions of the language.

See Also

condition, never

pointers

Name

`pointers` – indirect memory addressing.

Description

There are no pointers in the basic PROMELA language, although there is a way to circumvent this restriction through the use of embedded C code.

The two main reasons for leaving pointers out of the basic language are efficiency and tractability. To make verification possible, the verifier needs to be able to track all data that are part of reachable system states. SPIN maintains all such data, that is, local process states, local and global variables, and channel contents, in a single data structure called the system "state vector." The efficiency of the SPIN verifiers is in large part due to the availability of all state data within the simple, flat state vector structure, which allows each state comparison and state copying operation to be performed with a single system call.

The performance of a SPIN verifier can be measured in the number of reachable system states per second that can be generated and analyzed. In the current system, this performance is determined exclusively by the length of the state vector: a vector twice as long requires twice as much time to verify per state, and vice versa; every reduction in the length of a state vector translates into an increase of the verifier's efficiency. The cost per state is in most cases a small constant factor times the time needed to copy the bits in the state vector from one place to another (that is, the cost of an invocation of the system routine `memcpy()`).

The use of data that are only accessible through pointers during verification runs requires the verifier to collect the relevant data from all memory locations that could be pointed to at any one time and copy such information into the state vector. The associated overhead immediately translates in reduced verification efficiency.

See Chapter 17 for a discussion of the indirect support for pointers through the use of embedded C code fragments.

See Also

c_code, c_decl, c_expr

poll

Name

poll – a side effect free test for the executability of a non–rendezvous receive statements.

Syntax

name ? '[' recv_args ']'

name ?? '[' recv_args ']'

Description

A channel poll operation looks just like a receive statement, but with the list of message fields enclosed in square brackets. It returns either true or false, depending on the executability of the corresponding receive (i.e., the same operation written without the square brackets). Because its evaluation is side effect free, this form can be used freely in expressions or even assignments where a standard receive operation cannot be used.

The state of the channel, and all variables used, is guaranteed not to change as a result of the evaluation of this condition statement.

Examples

In the following example we use a channel poll operation to place an additional constraint on a timeout condition:

```
qname?[ack, var] && timeout
```

Notes

Channel poll operations do not work on rendezvous channels because synchronous channels never store messages that a poll operation could refer to. Messages are always passed instantly from sender to receiver in a rendezvous handshake.

It is relatively simple to create a conditional receive operation, with the help of a channel poll operation. For instance, if we want to define an extra boolean condition P that must hold before a given receive operation may be executed, we can write simply:

```
atomic { P && qname?[ack, var] -> qname[ack,var] }
```

This is harder to do for rendezvous channels; see the manual page for `cond_expr` for some examples.

See Also

cond_expr, condition, eval, receive

printf

Name

`printf` – for printing text during random or guided simulation runs.

Syntax

`printf ( string [, arg_lst ] )`

`printm ( expression )`

Executability

true

Effect

`none`

Description

A `printf` statement is similar to a `skip` statement in the sense that it is always executable and has no other effect on the state of the system than to change the control state of the process that executes it. A useful side effect of the statement is that it can print a string on the standard output stream during simulation runs. The PROMELA `printf` statement supports a subset of the options from its namesake in the programming language C. The first argument is an arbitrary string, in double quotes.

Six conversion specifications are recognized within the string. Upon printing, each subsequent conversion specification is replaced with the value of the next argument from the list that follows the string.
In addition, the white−space escape sequences \t (for a tab character) and \n (for a newline) are also recognized. Unlike the C version, optional width and precision fields are not supported.

The alternative form `printm` can be used to print just the symbolic name of an `mtype` constant. The two print commands in the following sequence, for instance, would both print the string `pear`:

```
mtype = { apple, pear, orange };
mtype x = pear;
printf("%e", x);
printm(x);
```

The method using `printf` works only when SPIN runs in simulation mode though, it does not work when an error trail is reproduced with the verifier (e.g., when embedded C code fragments are used). The alternative, using `printm`, always works.

Examples

```
printf("numbers: %d\t%d\n", (-10)%(-9), (-10)<<(-2))
```

Notes

`Printf` statements are useful during simulation and debugging of a verification model. In verification, however, they are of no value, and therefore not normally enabled. The order in which `printf`s are executed during verification is determined by the depth−first search traversal of the reachability graph, which does not necessarily make sense if interpreted as part of a straight execution. When SPIN generates the verifier's source text, therefore, it replaces every call to `printf` with a special one that is called `Printf`. The latter function is only allowed to produce output during the replay of an error trace. This function can also be called from within embedded C code fragments, to suppress unwanted output during verification runs.

Special Notes on XSPIN: The text printed by a `printf` statement that begins with the five characters: "`MSC:` " (three letters followed by a colon and a space) is automatically included in message sequence charts. For instance, when the statement

```
printf("MSC: State Idle\n")
```

is used, the string `State Idle` will included in the message sequence chart when this statement is reached. A more detailed description of this feature can also be found in Chapter 12, p. 272.

It is also possible to set breakpoints for a random simulation run, when XSPIN is used. To do so, the text that follows the `MSC:` prefix must match the five characters: `BREAK`, as in:

```
printf("MSC: BREAK\n")
```

These simulation breakpoints can be made conditional by embedding them into selection constructs. For instance:

```
if
:: P -> printf("MSC: BREAK\n")
:: else /* no breakpoint */
fi
```

See Also

do, if, skip

priority

Name

`priority` – for setting a numeric simulation priority for a process.

Syntax

`active [ '[' `const` ']' ] proctype` name `(` [decl_lst] `) priority` const { sequence }

`run` name `(` [arg_lst] `) priority const`

Description

Process priorities can be used in random simulations to change the probability that specific processes are scheduled for execution.

An execution priority is specified either as an optional parameter to a `run` operator, or as a suffix to an `active proctype` declaration. The optional `priority` field follows the closing brace of the parameter list in a `proctype` declaration.

The default execution priority for a process is one. Higher numbers indicate higher priorities, in such a way that a priority ten process is ten times more likely to execute than a priority one process.

The priority specified in an `active proctype` declaration affects all processes that are initiated through the `active` prefix, but no others. A process instantiated with a `run` statement is always assigned the priority that is explicitly or implicitly specified there (overriding the priority that may be specified in the `proctype` declaration for that process).

Examples

```
run name(...) priority 3
active proctype name() priority 12 { sequence }
```

If both a `priority` clause and a `provided` clause are specified, the `priority` clause should appear first.

```
active proctype name() priority 5 provided (a<b) {...}
```

Notes

Priority annotations only affect random simulation runs. They have no effect during verification, or in guided and interactive simulation runs. A priority designation on a `proctype` declaration that contains no `active` prefix is ignored.

See Also

active, proctype, provided

probabilities

Name

`probabilities` – for distinguishing between high and low probability actions.

Description

There is no mechanism in PROMELA for indicating the probability of a statement execution, other than during random simulations with `priority` tags.

SPIN is designed to check the unconditional correctness of a system. High probability executions are easily intercepted with standard testing and debugging techniques, but only model checking techniques are able to reproducibly detect the remaining classes of errors.

Disastrous error scenarios often have a low probability of occurrence that only model checkers can catch reliably. The use of probability tags on statement executions would remove the independence of probability, which seems counter to the premise of logic model checking. Phrased differently, verification in SPIN is concerned with possible behavior, not with probable behavior. In a well–designed system, erroneous behavior should be impossible, not just improbable.

To exclude known low probability event scenarios from consideration during model checking, a variety of other techniques may be used, including the use of model restriction, LTL properties, and the use of progress–state, end–state, and accept–state labels.

See Also

if, do, priority, progress, unless

procedures

Name

`procedures` – for structuring a program text.

Description

There is no explicit support in the basic PROMELA language for defining procedures or functions. This restrction can be circumvented in some cases through the use of either `inline` primitives, or embedded C code fragments.

The reason for this restriction to the basic language is that SPIN targets the verification of process interaction and process coordination structures, and not internal process computations. Abstraction is then best done at the process and system level, not at a computational level. It is possible to approximate a procedure call mechanism with PROMELA process instantiations, but this is rarely a good idea. Consider, for instance, the following model:

```
#ifndef N
#define N       12
#endif

int f = 1;

proctype fact(int v)
{
        if
        :: v > 1 -> f = v*f; run fact(v-1)
        :: else
        fi
}

init {
        run fact(N);
        (_nr_pr == 1) ->
        printf("%d! = %d\n", N, f)
}
```

Initially, there is just one process in this system, the `init` process. It instantiates a process of type `fact` passing it the value of constant `N`, which is defined in a macro. If the parameter passed to the process of type `fact` is greater than one, the value of global integer `f` is multiplied by `v`, and another copy of `fact` is instantiated with a lower value of the parameter.

The procedure of course closely mimics a recursive procedure to compute the factorial of `N`. If we store the model in a file called `badidea` and execute the model, we get

```
$ spin badidea
12! = 479001600
13 processes created
```

which indeed is the correct value for the factorial. But, there are a few potential gotcha's here. First, the processes that are instantiated will execute asynchronously with the already running processes. Specifically, we cannot assume that the process that is instantiated in a `run` statement terminates its execution before the process that executed the `run` reaches its next statement. Generally, the newly created process will start executing concurrently with its creator. Nothing can be assumed about the speed of execution of a running process. If a particular order of execution is important, this must be enforced explicitly through process synchronization. In the initially running `init` process from the example, synchronization is achieved in one place with the expression

```
(_nr_pr == 1)
```

The variable `_nr_pr` is a predefined, global system variable that records the number of current executing processes, see the corresponding manual page. Because there is initially just one executing process (the process of type `main` itself), we know in this case that all newly instantiated processes must have terminated once the evaluation of this expression yields true. Recall that a condition statement can only be executed in PROMELA if it evaluates to true, which gives us the required synchronization, and guarantees that the final value of `f` is not printed before it is fully computed.

A more obvious gotcha is that the maximum useful value we can choose for the constant `N` is limited by the maximum number of processes that can simultaneously be instantiated. The maximum value that can be represented in a variable of type `int` is more restrictive in this case, though. The size of an `int` is the same in PROMELA as it is in the underlying programming language C, which at the time of writing means only 32 bits on most machines. The maximum signed value that can be represented in a 32 bit word is $2^{31} - 1 = 2,$ $147, 483, 648$, which means that the largest factorial we can compute with our model is an unimpressive 13! = 1,932,053,504. To do better, we would need a data type `double` or `float`, but PROMELA deliberately does not have them. The only way we could get these would be through the use of embedded C code fragments. The more fundamental reason why these data types are not part of native PROMELA is that any need to represent data quantities of this size almost certainly means that the user is trying to model a computational problem, and not a process synchronization problem. The omission of the larger data types from the language serves as a gentle warning to the user that the language is meant for design verification, and not for design implementation.

If a procedural mechanism is to be used, the most efficient method would be to use a `macro` or an `inline` definition. This amounts to an automatic inlining of the text of a procedure call into the body of each process that invokes it. A disadvantage of a `macro` is that line–number references will be restricted to the location of the macro call, not a line number within a macro definition itself. This problem does not exist with an `inline` definition.

If a separate process is used to model the procedure, the best way to do so is to declare it as a permanent server by declaring it as an `active proctype:` receiving requests from user processes via a special globally defined channel, and responding to these requests via a user–provided local channel.

The least attractive method is to instantiate a new copy of a process once for each procedure call and wait for that process to return a result (via a global variable or a message channel) and then disappear from the system before proceeding. This is less attractive because it produces the overhead of process creation and deletion, and can add the complication of determining reliably when precisely a process has disappeared from the system.

See Also

_nr_pr, active, c_code, c_expr, hierarchy, inline, macros, proctype

proctype

Name

proctype – for declaring new process behavior.

Syntax

`proctype` name ([decl_lst]) { sequence }

`D_proctype` name ([decl_lst]) { sequence }

Description

All process behavior must be declared before it can be instantiated. The `proctype` construct is used for the declaration. Instantiation can be done either with the `run` operator, or with the prefix `active` that can be used at the time of declaration.

Declarations for local variables and message channels may be placed anywhere inside the `proctype` body. In all cases, though, these declarations are treated as if they were all placed at the start of the `proctype` declaration. The scope of local variables cannot be restricted to only part of the `proctype` body.

The keyword `D_proctype` can be used to declare process behavior that is to be executed completely deterministically. If non–determinism is nonetheless present in this type of process definition, it is resolved in simulations in a deterministic, though otherwise undefined, manner. During verifications an error is reported if non–determinism is encountered in a `D_proctype` process.

Examples

The following program declares a `proctype` with one local variable named `state`:

```
proctype A(mtype x) { mtype state; state = x }
```

The process type is named `A`, and has one formal parameter named `x`.

Notes

Within a `proctype` body, formal parameters are indistinguishable from local variables. Their only distinguishing feature is that their initial values can be determined by an instantiating process, at the moment when a new copy of the process is created.

See Also

_pid, active, init, priority, provided, remoterefs, run

progress

Name

`progress` – label–name prefix for specifying liveness properties.

Syntax

`progress [a-zA-Z0-9_]*` : stmnt

Description

A progress label is any label name that starts with the eight–character sequence `progress`. It can appear anywhere a label can appear.

A label always prefixes a statement, and thereby uniquely identifies a local process state (i.e., the source state of the transition that corresponds to the labeled statement). A progress label marks a state that is required to be traversed in any infinite execution sequence.

A progress label can appear in a `proctype`, or `trace` declaration, but has no special semantics when used in a `never` claim or in `notrace` declarations. Because a global system state is a composite of local component states (e.g., `proctype` instantiations, and an optional `trace` component), a progress label indirectly also marks global system states where one or more of the component systems is labeled with a `progress` label.

Progress labels are used to define correctness claims. A progress label states the requirement that the labeled global state must be visited infinitely often in any infinite system execution. Any violation of this requirement can be reported by the verifier as a non–progress cycle.

Examples

```
active proctype dijkstra()
{
        do
        :: sema!p ->
progress:       sema?v
        od
}
```

The requirement expressed here is that any infinite system execution contains infinitely many executions of the statement `sema?v`.

Notes

Progress labels are typically used to mark a state where effective progress is being made in an execution, such as a sequence number being incremented or valid data being accepted by a receiver in a data transfer protocol. They can, however, also be used during verifications to eliminate harmless variations of liveness violations. One such application, for instance, can be to mark message loss events with a pseudo `progress` label, to indicate that sequences that contain infinitely many message loss events are of secondary interest. If we now search for non−progress executions, we will no longer see any executions that involve infinitely many message loss events.

SPIN has a special mode to prove absence of non−progress cycles. It does so with the predefined LTL formula:

```
(<>[] np_)
```

which formalizes non−progress as a standard Büchi acceptance property.

The standard stutter−extension, to extend finite execution sequences into infinite ones by stuttering (repeating) the final state of the sequence, is applied in the detection of all acceptance properties, including non−progress cycles.

The manual page for `never` claims describes how the predefined variable `np_` can also be used to restrict a verification to precisely the set of either progress or non−progress cycles.

See Also

accept, end, labels, ltl, never, np_, trace

provided

Name

provided – for setting a global constraint on process execution.

Syntax

proctype name ([decl_lst]) provided (expr) { sequence }

Description

Any proctype declaration can be suffixed by an optional provided clause to constrain its execution to those global system states for which the corresponding expression evaluates to true. The provided clause has the effect of labeling all statements in the proctype declaration with an additional, user–defined executability constraint.

Examples

The declaration:

```
byte a, b;
active proctype A() provided (a > b)
{
	...
}
```

makes the execution of all instances of proctype A conditional on the truth of the expression (a>b), which is, for instance, not true in the initial system state. The expression can contain global references, or references to the process's _pid, but no references to any local variables or parameters.

If both a priority clause and a provided clause are specified, the priority clause should come first.

```
active proctype name() priority 2 provided (a > b )
{
	...
}
```

Notes

`Provided` clauses are incompatible with partial order reduction. They can be useful during random simulations, or in rare cases to control and reduce the complexity of verifications.

See Also

_pid, active, hidden, priority, proctype

rand

Name

`rand` – for random number generation.

Description

There is no predefined random number generation function in PROMELA. The reason is that during a verification we effectively check for all possible executions of a system. Having even a single occurrence of a call on the random number generator would increase the number of cases to inspect by the full range of the random numbers that could be generated: usually a huge number. Random number generators can be useful on a simulation, but they can be disastrous when allowed in verification.

In almost all cases, PROMELA's notion of non−determinism can replace the need for a random number generator. Note that to make a random choice between `N` alternatives, it suffices to place these `N` alternatives in a selection structure with `N` options. The verifier will interpret the non−determinism accurately, and is not bound to the restrictions of a pseudo−random number generation algorithm.

During random simulations, SPIN will internally make calls on a (pseudo) random number generator to resolve all cases of non−determinism. During verifications no such calls are made, because effectively all options for behavior will be explored in this mode, one at a time.

PROMELA's equivalent of a "random number generator" is the following program:

```
active proctype randnr()
{       /*
         * don't call this rand()...
         * to avoid a clash with the C library routine
         */
        byte nr;        /* pick random value  */
        do
        :: nr++         /* randomly increment */
        :: nr--         /* or decrement       */
        :: break        /* or stop            */
        do;
        printf("nr: %d\n")      /* nr: 0..255 */
}
```

Note that the verifier would generate at least 256 distinct reachable states for this model. The simulator, on the other hand, would traverse the model only once, but it could execute a sequence of any length (from one to infinitely many execution steps). A simulation run will only terminate if the simulator eventually selects the `break` option (which is guaranteed only in a statistical sense).

Notes

Through the use of embedded C code, a user can surreptitiously include calls on an external C library `rand()` function into a model. To avoid problems with irreproducible behavior, the SPIN−generated verifiers intercept such calls and redefine them in such a way that the depth−first search process at the very least remains deterministic. SPIN accomplishes this by pre−allocating an integer array of the maximum search depth `maxdepth`, and filling that array with the first `maxdepth` random numbers that are generated. Those numbers are then reused each time the search returns to a previously visited point in the search, to secure the sanity of the search process.

The seed for this pre−computation of random numbers is fixed, so that subsequent runs will always give the same result, and to allow for the faithful replay of error scenarios. Even though this provides some safeguards, the use of random number generation is best avoided, also in embedded C code.

See Also

c_code, c_expr, if, do

real−time

Name

`real time` – for relating properties to real−time bounds.

Description

In the basic PROMELA language there is no mechanism for expressing properties of clocks or of time related properties or events. There are good algorithms for integrating real−time constraints into the model checking process, but most attention has so far been given to real−time verification problems in hardware circuit design, rather than the real−time verification of asynchronous software, which is the domain of the SPIN model checker.

The best known of these algorithms incur significant performance penalties compared with untimed verification. Each clock variable added to a model can increase the time and memory requirements of verification by an order of magnitude. Considering that one needs at least two or three such clock variables to define meaningful constraints, this seems to imply, for the time being, that a real−time capability requires at least three to four orders of magnitude more time and memory than the verification of the same system without time constraints.

The good news is that if a correctness property can be proven for an untimed PROMELA model, it is guaranteed to preserve its correctness under all possible real−time constraints. The result is therefore robust, it can be obtained efficiently, and it encourages good design practice. In concurrent software design it is usually unwise to link logical correctness with real−time performance.

PROMELA is a language for specifying systems of asynchronous processes. For the definition of such a system we abstract from the behavior of the process scheduler and from any assumption about the relative speed of execution of the various processes. These assumptions are safe, and the minimal assumptions required to allow us to construct proofs of correctness. The assumptions differ fundamentally from those that can be made for hardware systems, which are often driven by a single known clock, with relative speeds of execution precisely known. What often is just and safe in hardware verification is, therefore, not necessarily just and safe in software verification.

SPIN guarantees that all verification results remain valid independent of where and how processes are executed, timeshared on a single CPU, in true concurrency on a multiprocessor, or with different processes running on CPUs of different makes and varying speeds. Two points are worth considering in this context: first, such a guarantee can no longer be given if real−time constraints are introduced, and secondly, most of the existing real−time verification methods assume a true concurrency model, which inadvertently excludes the more common method of concurrent process execution by timesharing.

It can be hard to define realistic time bounds for an abstract software system. Typically, little can be firmly known about the real−time performance of an implementation. It is generally unwise to rely on speculative information, when attempting to establish a system's critical correctness properties.

See Also

priorities, probabilities

receive

Name

receive statement – for receiving messages from channels.

Syntax

name `?` recv_args

name `??` recv_args

name `?`< recv_args >

name `??`< recv_args >

Executability

The first and the third form of the statement, written with a single question mark, are executable if the first message in the channel matches the pattern from the receive statement.

The second and fourth form of the statement, written with a double question mark, are executable if there exists at least one message anywhere in the channel that matches the pattern from the receive statement. The first such message is then used.

A match of a message is obtained if all message fields that contain constant values in the receive statement equal the values of the corresponding message fields in the message.

Effect

If a variable appears in the list of arguments to the receive statement, the value from the corresponding field in the message that is matched is copied into that variable upon reception of the message. If no angle brackets are used, the message is removed from the channel buffer after the values are copied. If angle brackets are used, the message is not removed and remains in the channel.

Description

The number of message fields that is specified in the receive statement must always match the number of fields that is declared in the channel declaration for the channel addressed. The types of the variables used in the message fields must be compatible with the corresponding fields from the channel declaration. For integer data types, an equal or larger value range for the variable is considered to be compatible (e.g., a `byte` field may be received in a `short` variable, etc.). Message fields that were declared to contain a user–defined data type or a `chan` must always match precisely.

The first form of the receive statement is most commonly used. The remaining forms serve only special purposes, and can only be used on buffered message channels.

The second form of the receive statement, written with two question marks, is called a random receive statement. The variants with angle brackets have no special name.

Because all four types of receive statements discussed here can have side effects, they cannot be used inside expressions (see the manual page `poll` for some alternatives).

Examples

```
chan set = [8] of { byte };
byte x;

set!!3; set!!5; set!!2; /* sorted send operations */

set?x;                  /* get first element */
if
:: set?<x>        /* copy first element */
:: set??5               /* is there a 5 in the set? */
:: empty(set)
fi
```

In this example we first send three values into a channel that can contain up to eight messages with one single field of type `byte`. The values are within the range that is expected, so no value truncations will occur. The use of the sorted send operator (the double exclamation) causes the three values to be stored in numerical order. A regular receive operation is now used to retrieve the first element from the channel, which should be the value two.

The selection statement that follows has three options for execution. If the channel is empty at this point, only the third statement will be executable. If the channel is non−empty, and contains at least one message with the value five, the second option will be executable. Because of the use of the random receive operator (the double question mark), the target message may appear anywhere in the channel buffer and need not be the first message. It is removed from the channel when matched. The first option in the selection structure is executable if the channel contains any message at all. Its effect when executed will be to copy the value of the first message that is in the channel at this point into variable `x`. If all is well, this should be the value three. If this option is executed, the message will remain in the channel buffer, due to the use of the angle brackets.

See Also

chan, empty, eval, full, len, nempty, nfull, poll, send

remoterefs

Name

remote references – a mechanism for testing the local control state of an active process, or the value of a local variable in an active process from within a `never` claim.

Syntax

name ['[' any_expr ']'] @labelname

name ['[' any_expr '] '] : varname

Description

The remote reference operators take either two or three arguments: the first, required, argument is the name of a previously declared `proctype`, a second, optional, argument is an expression enclosed in square brackets, which provides the process instantiation number of an active process. With the first form of remote reference, the third argument is the name of a control–flow label that must exist within the named `proctype`. With the second form, the third argument is the name of a local variable from the named `proctype`.

The second argument can be omitted, together with the square brackets, if there is known to be only one instantiated process of the type that is named.

A remote reference expression returns a non–zero value if and only if the process referred to is currently in the local control state that was marked by the label name given.

Examples

```
active proctype main () {
        byte x;
L:      (x < 3) ->
        x++
}
never { /* process main cannot remain at L forever */
accept: do
        :: main@L
        od
}
```

Notes

Because `init`, `never`, `trace`, and `notrace` are not valid `proctype` names but keywords, it is not possible to refer to the state of these special processes with a remote reference:

```
init@label     /* invalid */
never[0]@label /* invalid */
```

Note that the use of `init`, can always be avoided, by replacing it with an `active proctype`.

A remote variable reference, the second form of a remote reference, bypasses the standard scope rules of PROMELA by making it possible for the `never` claim to refer to the current value of local variables inside a running process.

For instance, if we wanted to refer to the variable `count` in the process of type `Dijkstra` in the example on page 77, we could do so with the syntax `Dijkstra[0] : count`, or if there is only one such process, we can use the shorter form `Dijkstra : count`.

The use of remote variable references is not compatible with SPIN's partial order reduction strategy. A wiser strategy is therefore usually to turn local variables whose values are relevant to a global correctness property into global variables, so that they can be referenced as such. See especially the manual page for `hidden` for an efficient way of doing this that preserves the benefits of partial order reduction.

See Also

_pid, active, condition, hidden, proctype, run

run

Name

`run` – predefined, unary operator for creating new processes.

Syntax

`run` name `(` `[` arg_lst `]` `)`

Description

The `run` operator takes as arguments the name of a previously declared `proctype`, and a possibly empty list of actual parameters that must match the number and types of the formal parameters of that `proctype`. The operator returns zero if the maximum number of processes is already running, otherwise it returns the process instantiation number of a new process that is created. The new process executes asynchronously with the existing active processes from this point on. When the `run` operator completes, the new process need not have executed any statements.

The `run` operator must pass actual parameter values to the new process, if the `proctype` declaration specified a non–empty formal parameter list. Only message channels and instances of the basic data types can be passed as parameters. Arrays of variables cannot be passed.

`Run` can be used in any process to spawn new processes, not just in the initial process. An active process need not disappear from the system immediately when it terminates (i.e., reaches the end of the body of its process type declaration). It can only truly disappear if all younger processes have terminated first. That is, processes can only disappear from the system in reverse order of their creation.

Examples

```
proctype A(byte state; short set)
{
        (state == 1) -> state = set
}

init {
        run A(1, 3)
}
```

Notes

Because PROMELA defines finite state systems, the number of processes and message channels is required to be bounded. SPIN limits the number of active processes to 255.

Because `run` is an operator, `run A()` is an expression that can be embedded in other expressions. It is the only operator allowed inside expressions that can have a side effect, and therefore there are some special restrictions that are imposed on expressions that contain `run` operators.

Note, for instance, that if the condition statement

```
(run A() && run B())
```

were allowed, in the evaluation of this expression it would be possible that the first application of the `run` operator succeeds, and the second fails when the maximum number of runnable processes is reached. This would produce the value false for the expression, and the condition statement would then block, yet a side effect of the evaluation has occurred. Each time the evaluation of the expression is repeated, one more process could then be created.

Therefore, the SPIN parser imposes the restriction that an expression cannot contain more than one `run` operator, and this operator cannot be combined in a compound expression with other conditionals. Also, as a further precaution, an attempt to create a 256th process is always flagged as an error by the verifier, although technically it would suffice to allow the `run` operator to return a zero value.

See Also

_pid, active, priority, proctype, provided, remoterefs

scanf

Name

`scanf` – to read input from the standard input stream.

Description

There is no routine in PROMELA comparable to the C library function `scanf` to read input from the standard input stream or from a file or device. The reason is that PROMELA models must be `closed` to be verifiable. That is, all input sources must be part of the model. It is relatively easy to build a little process that acts as if it were the `scanf` routine, and that sends to user processes that request its services a non–deterministically chosen response from the set of anticipated responses.

As a small compromise, PROMELA does include a special predefined channel named `STDIN` that can be used to read characters from the standard input during simulation experiments. The use of `STDIN` is not supported in verification runs.

See Also

c_code, printf, STDIN

send

Name

send statement – for sending messages to channels.

Syntax

name ! send_args

name ! ! send_args

Executability

A send statement on a buffered channel is executable in every global system state where the target channel is non−full. SPIN supports a mechanism to override this default with option `-m`. When this option is used, a send statement on a buffered channel is always executable, and the message is lost if the target channel is full.

The execution of a send statement on a rendezvous channel consists, conceptually, of two steps: a rendezvous offer and a rendezvous accept. The rendezvous offer can be made at any time (see Chapter 7). The offer can be accepted only if another active process can perform the matching receive operation immediately (i.e., with no intervening steps by any process). The rendezvous send operation can only take place if the offer made is accepted by a matching receive operation in another process.

Effect

For buffered channels, assuming no message loss occurs (see above), the message is added to the channel. In the first form of the send statement, with a single exclamation mark, the message is appended to the tail of the channel, maintaining fifo (first in, first out) order. In the second form, with a double exclamation mark, the message is inserted into the channel immediately ahead of the first message in the channel that succeeds it in numerical order. To determine the numerical order, all message fields are significant.

Within the semantics model, the effect of issuing the rendezvous offer is to set global system variable `handshake` to the channel identity of the target channel (see Chapter 7).

Description

The number of message fields that is specified in the send statement must always match the number of fields that is declared in the channel declaration for the target channel, and the values of the expressions specified in the message fields must be compatible with the datatype that was declared for the corresponding field. If the type of a message field is either a user−defined type or `chan`, then the types must match precisely.

The first form of the send statement is the standard fifo send. The second form, with the double exclamation mark, is called a sorted send operation. The sorted send operation can be exploited by, for instance, listing an appropriate message field (e.g., a sequence number) as the first field of each message, thus forcing a message ordering in the target channel.

Examples

In the following example our test process uses sorted send operations to send three messages into a buffered channel named x. Then it adds one more message with the value four.

```
chan x = [4] of { short };

active proctype tester()
{
        x!!3; x!!2; x!!1; x!4;
        x?1; x?2; x?3; x?4
}
```

All four values are now receivable in numerical order; the last message only coincidentally, but the first three due to the ordering discipline that is enforced by the sorted send operators. A simulation confirms this:

```
$ spin -c tester.pml
proc 0 = tester
q   0
  1   x!3
  1   x!2
  1   x!1
  1   x!4
  1   x?1
  1   x?2
  1   x?3
  1   x?4
_____________
final state:
_____________
1 process created
```

Notes

By convention, the first field in a message is used to specify the message type, and is defined as an `mtype`.

Sorted send operations and fifo send operations can safely be mixed.

See Also

chan, empty, full, len, nempty, nfull, poll, receive

separators

Name

separators – for sequential composition of statements and declarations.

Syntax

step ; step

step -> step

Description

The semicolon and the arrow are equivalent statement separators in PROMELA; they are not statement terminators, although the parser has been taught to be forgiving for occasional lapses. The last statement in a sequence need not be followed by a statement separator, unlike, for instance, in the C programming language.

Examples

```
x = 3;
atomic {
        x = y;
        y = x  /* no separator is required here */
};             /* but it is required here... */
y = 3
```

Notes

The convention is to reserve the use of the arrow separator to follow condition statements, such as guards in selection or repetition structures. The arrow symbol can thus be used to visually identify those points in the code where execution could block.

See Also

break, labels, goto

sequence

Name

`sequence` – curly braces, used to enclose a block of code.

Syntax

{ sequence }

Description

Any sequence of PROMELA statements may be enclosed in curly braces and treated syntactically as if it were a statement. This facility is most useful for defining `unless` constructs, but can also generally be used to structure a model.

Examples

```
if
:: a < b -> { tmp = a; a = b; b = a }
:: else  ->
        { printf("unexpected case\n");
          assert(false)
        }
fi
```

The more common use is for structuring `unless` statements, as in:

```
{ tmp = a; a = b; b = a; }
unless
{ a >= b }
```

Note the differences between these two examples. In the first example, the value of the expression `a < b` is checked once, just before the bracketed sequence is executed. In the second example, the value of the negated expression is checked before each statement execution in the main sequence, and execution is interrupted when that expression becomes true.

Notes

The last statement in a sequence need not be followed by a statement separator, but if the sequence is followed by another statement, the sequence as a whole should be separated from that next statement with a statement separator.

See Also

atomic, d_step, unless

show

Name

`show` – to allow for tracking of the access to specific variables in message sequence charts.

Syntax

`show` typename name

Description

This keyword has no semantic content. It only serves to determine which variables should be tracked and included in message sequence chart displays in the XSPIN tool. Updates of the value of all variables that are declared with this prefix are maintained visually, in a separate process line, in these message sequence charts.

Notes

The use of this prefix only affects the information that XSPIN includes in message sequence charts, and the information that SPIN includes in Postscript versions of message sequence charts under SPIN option `-M`.

See Also

datatypes, hidden, local, show

skip

Name

`skip` – shorthand for a dummy, nil statement.

Syntax

`skip`

Description

The keyword `skip` is a meta term that is translated by the SPIN lexical analyzer into the constant value one (1), just like the predefined boolean constant `true`. The intended use of the shorthand is stand–alone, as a dummy statement. When used as a statement, the `skip` is interpreted as a special case of a condition statement. This condition statement is always executable, and has no effect when executed, other than a possible change of the control–state of the executing process.

There are few cases where a `skip` statement is needed to satisfy syntax requirements. A common use is to make it possible to place a `label` at the end of a statement sequence, to allow for a `goto` jump to that point. Because only statements can be prefixed by a label, we must use a dummy `skip` statement as a placeholder in those cases.

Examples

```
proctype A()
{
L0:     if
        :: cond1 -> goto L1 /* jump to end    */
        :: else -> skip     /* skip redundant */
        fi;

        ...

L1:     skip
}
```

The `skip` statement that follows label `L1` is required in this example. The use of the `skip` statement following the `else` guard in the selection structure above is redundant. The above selection can be written more tersely as:

```
L0:    if
```

```
    :: cond1 -> goto L1
    :: else
    fi;
```

Because PROMELA is an asynchronous language, the `skip` is never needed, nor effective, to introduce delay in process executions. In PROMELA, by definition, there can always be an arbitrary, and unknowable, delay between any two subsequent statement executions in a `proctype` body. This semantics correspond to the golden rule of concurrent systems design that forbids assumptions about the relative execution speed of asynchronous processes in a concurrent system. When SPIN's weak fairness constraint is enforced we can tighten this semantics a little, to conform to, what is known as, Dijkstra's finite progress assumption. In this case, when control reaches a statement, and that statement is and remains executable, we can are allowed to assume that the statement will be executed within a finite period of time (i.e., we can exclude the case where the delay would be infinite).

Notes

The opposite of `skip` is the zero condition statement `(0)`, which is never executable. In cases where such a blocking statement might be needed, often an assertion statement is more effective. Note that `assert(false)` and `assert(0)` are equivalent. Similarly, `assert(true)` and `assert(1)` are equivalent and indistinguishable from both `assert(skip)` and `skip`.

Because `skip` is intercepted in the lexical analyzer as a meta term, it does not appear literally in error traces. It will only show up as its numeric equivalent (1).

See Also

assert, condition, else, false, true

STDIN

Name

STDIN – predefined read–only channel, for use in simulation.

Syntax

```
chan STDIN; STDIN?var
```

Description

During simulation runs, it is sometimes useful to be able to connect SPIN to other programs that can produce useful input, or directly to the standard input stream to read input from the terminal or from a file.

Examples

A sample use of this feature is this model of a word count program:

```
chan STDIN; /* no channel initialization */

init {
	int c, nl, nw, nc;
	bool inword = false;
	do
	:: STDIN?c ->
		if
		:: c == -1 -> break /* EOF */
		:: c == '\n' -> nc++; nl++
		:: else -> nc++
		fi;
		if
		:: c == ' ' || c == '\t' || c == '\n' ->
			inword = false
		:: else ->
			if
			:: !inword ->
				nw++; inword = true
			:: else /* do nothing */
		fi	fi
	od;
	printf("%d\t%d\t%d\n", nl, nw, nc)
}
```

Notes

The STDIN channel can be used only in simulations. The name has no special meaning in verification. A verification for the example model would report an attempt to receive data from an unitialized channel.

See Also

chan, poll, printf, receive

timeout

Name

`timeout` – a predefined, global, read–only, boolean variable.

Syntax

`timeout`

Description

`Timeout` is a predefined, global, read–only, boolean variable that is true in all global system states where no statement is executable in any active process, and otherwise is false (see also Chapter 7).

A `timeout` used as a guard in a selection or repetition construct provides an escape from a system hang state. It allows a process to abort waiting for a condition that can no longer become true.

Examples

The first example shows how `timeout` can be used to implement a watchdog process that sends a reset message to a channel named `guard` each time the system enters a hang state.

```
active proctype watchdog()
{       do
        :: timeout -> guard!reset
        od
}
```

A more traditional use is to place a timeout as an alternative to a potentially blocking statement, to guard against a system deadlock if the statement becomes permanently blocked.

```
do
:: q?message -> ...
:: timeout -> break
od
```

Notes

The `timeout` statement can not specify a timeout interval. `Timeouts` are used to model only possible system behaviors, not detailed real−time behavior. To model premature expiration of timers, consider replacing the `timeout` variable with the constant value `true`, for instance, as in:

```
#define timeout true
```

A `timeout` can be combined with other expressions to form more complex wait conditions, but can not be combined with `else`. Note that `timeout`, if used as a condition statement, can be considered to be a system level `else` statement. Where the `else` statement becomes executable only when no other statements within the executing process can be executed, a `timeout` statement becomes executable only when no other statements anywhere in the system can be executed.

See Also

condition, do, else, if, unless

trace

Name

`trace`, `notrace` – for defining event sequences as properties.

Syntax

`trace` { sequence }

`notrace` { sequence }

Description

Much like a `never` claim declaration, a `trace` or `notrace` declaration does not specify new behavior, but instead states a correctness requirement on existing behavior in the remainder of the system. All channel names referenced in a `trace` or `notrace` declaration must be globally declared message channels, and all message fields must either be globally known (possibly symbolic) constants, or the predefined global variable _, which can be used in this context to specify don't care conditions. The structure and place of a `trace` event declaration within a PROMELA model is similar to that of a `never` claim: it must be declared globally.

An event `trace` declaration defines a correctness claim with the following properties:

- Each channel name that is used in an event `trace` declaration is monitored for compliance with the structure and context of the `trace` declaration.
- If only send operations on a channel appear in the `trace`, then only send operations on that channel are subject to the check. The same is true for receive operations. If both types appear, both are subject to the check, and they must occur in the relative order that the `trace` declaration gives.
- An event `trace` declaration may contain only send and receive operations (that is, events), but it can contain any control flow construct. This means that no global or local variables can be declared or referred to. This excludes the use of assignments and conditions. Send and receive operations are restricted to simple sends or receives; they cannot be variations such as random receive, sorted send, receive test, etc.
- Message fields that must be matched in sends or receives must be specified either with the help of symbolic `mtype` names, or with constants. Message fields that have don't care values can be matched with the predefined write–only variable _ (underscore).
- Sends and receives that appear in an event `trace` are called monitored events. These events do not generate new behavior, but they are required to match send or receive events on the same channels in the model with matching message parameters. A send or receive event occurs whenever a send or a receive statement is executed, that is, an event occurs during a state transition.
- An event `trace` can capture the occurrence of receive events on rendezvous channels.
- An event `trace` causes a correctness violation if a send or receive action is executed in the system that is within the scope of the event `trace`, but that cannot be matched by a monitored event within that declaration.
- One can use `accept`, `progress`, and `end`–state labels in event `trace` declarations, with the usual interpretation.

An event trace declaration must always be deterministic.

A trace declaration specifies behavior that must be matched by the remainder of the specification, and a notrace declares behavior that may not be matched.

A notrace definition is subject to the same requirements as a trace definition, but acts as its logical negation. A notrace definition is violated if the event sequence that is specified can be matched completely, that is, if either a user–defined end state in the trace definition is reached, or the closing curly brace of the declaration.

Examples

An event trace declaration that specifies the correctness requirement that send operations on channel q1 alternate with receive operations on channel q2, and furthermore that all send operations on q1 are (claimed to be) exclusively messages of type a, and all receive operations on channel q2 are exclusively messages of type b, is written as follows:

```
mtype = { a, b };
trace {
        do
        :: q1!a; q2?b
        od
}
```

Notes

There are two significant differences between an event trace and a never claim declaration: First, an event trace matches event occurrences that can occur in the transitions between system states, whereas a never claim matches boolean propositions on system states.

A system state, for the purposes of verification, is a stable value assignment to all variables, process states, and message channels. The transitions of a never claim are labeled with boolean propositions (expressions) that must evaluate to true in system states. The transitions of an event trace are labeled directly with monitored events that must occur in system transitions in the order that is given in the trace declaration.

The second difference is that an event trace monitors only a subset of the events in a system: only those of the types that are mentioned in the trace (i.e., the monitored events). A never claim, on the other hand, looks at all global systems states that are reached, and must be able to match the state assignments in the system at every state.

An event trace automaton, just like a never claim automaton, has a current state, but it only executes transitions if one of the monitored events occurs. That is, unlike a never claim, it does not execute synchronously with the system.

It is relatively easy to monitor receive events on rendezvous channels with an event trace assertion, but very hard to do so with a never claim. Monitoring the send event on a rendezvous channel is also possible, but it would also have to match all rendezvous send offers that are made, including those that do not lead to an accepting receive event.

See Also

_, accept, assert, end, ltl, never, progress

true

Name

`true` – predefined boolean constant.

Syntax

`true`

Description

The keyword `true` is a synonym of the constant value one `(1)`, and can be used in any context. Because of the mapping to `(1), true` is also a synonym of `skip`. It supports a more natural syntax for manipulating boolean values.

Notes

Because it is intercepted in the lexical analyzer as a meta term, `true` is always replaced by its numeric equivalent in error traces.

Semantically, `true`, `skip`, and `(1)` are indistinguishable. Which term is best used depends on context and convention.

See

condition, false, skip

typedef

Name

typedef – to declare a user–defined structured data type.

Syntax

typedef name { decl_lst }

Description

Typedef declarations can be used to introduce user–defined data types. User–defined types can be used anywhere predefined integer data types can be used. Specifically, they can be used as formal and actual parameters for proctype declarations and instantiations, as fields in message channels, and as arguments in message send and receive statements.

A typedef declaration can refer to other, previously declared typedef structures, but it may not be self–referential. A typedef definition must always be global, but it can be used to declare both local and global data objects of the new type.

Examples

The first example shows how to declare a two–dimensional array of elements of type byte with a typedef.

```
typedef array { /* typedefs must be global */
        byte aa[4]
};

init {
        array a[8];     /* 8x4 = 32 bytes total */
        a[3] .aa[1] = 5
}
```

The following example introduces two user–defined types named D and Msg, and declares an array of two objects of type Msg, named top:

```
typedef D {
        short f;
        byte g
};
```

```
typedef Msg {
        byte a[3];
        int fld1;
        D   fld2;
        chan p[3];
        bit b
};
Msg top[2];
```

The elements of `top` can be referenced as, for instance:

```
top[1].fld2.g = top[0].a[2]
```

Objects of type `Msg` can be passed through a channel, provided that they do not contain any field of type `unsigned`.

```
chan q = [2] of { Msg };
q!top[0]; q?top[1]
```

If we delete the arrays from the declaration of type `Msg` we can also use objects of this type in a `run` parameter, for instance, as follows:

```
typedef D {
        short f;
        byte g
};

typedef Msg {
        int fld1;
        D   fld2;
        bit b
};

Msg top[2];

proctype boo(Msg m)
{
        printf("fld1=%d\n", m.fld1);
}

init {
        chan q = [2] of { Msg };

        top[0].fld1 = 12;
        q!top[0]; q?top[1];
        run boo(top[1])
}
```

Notes

The current SPIN implementation imposes the following restrictions on the use of `typedef` objects. It is not possible to assign the value of a complete `typedef` object directly to another such object of the same type in a single assignment. A `typedef` object may be sent through a message channel as a unit provided that it contains no fields of type `unsigned`. A `typedef` object can also be used as a parameter in a `run` statement, but in this case it may not contain arrays.

Beware that the use of this keyword differs from its namesake in the C programming language. The working of the C version of a `typedef` statement is best approximated with a macro definition.

See Also

arrays, datatypes, macros, mtype

unless

Name

unless – to define exception handling routines.

Syntax

stmnt unless stmnt

Description

Similar to the repetition and selection constructs, the unless construct is not really a statement, but a method to define the structure of the underlying automaton and to distinguish between higher and lower priority of transitions within a single process. The construct can appear anywhere a basic PROMELA statement can appear.

The first statement, generally defined as a block or sequence of basic statements, is called the main sequence. The second statement is called the escape sequence. The guard of either sequence can be either a single statement, or it can be an if, do, or lower level unless construct with multiple guards and options for execution.

The executability of all basic statements in the main sequence is constrained to the non−executability of all guard statements of the escape sequence. If and when one of the guard statements of the escape sequence becomes executable, execution proceeds with the remainder of the escape sequence and does not return to the main sequence. If all guards of the escape sequence remain unexecutable throughout the execution of the main sequence, the escape sequence as a whole is skipped.

The effect of the escape sequence is distributed to all the basic statements inside the main sequence, including those that are contained inside atomic sequences. If a d_step sequence is included, though, the escape affects only its guard statement (that is, the first statement) of the sequence, and not the remaining statements inside the d_step. A d_step is always equivalent to a single statement that can only be executed in its entirety from start to finish.

As noted, the guard statement of an unless construct can itself be a selection or a repetition construct, allowing for a non−deterministic selection of a specific executable escape. Following the semantics model from Chapter 7, the guard statements of an escape sequence are assigned a higher priority than the basic statements from the main sequence.

Unless constructs may be nested. In that case, the guard statements from each unless statement take higher priority than those from the statements that are enclosed. This priority rule can be reversed, giving the highest priority to the most deeply nested unless escapes, by using SPIN run−time option −J. This option is called −J because it enforces a priority rule that matches the evaluation order of nested catch statements in Java programs.

PROMELA `unless` statements are meant to facilitate the modeling of error handling methods in implementation level languages.

Examples

Consider the following `unless` statement:

```
{ B1; B2; B3 } unless { C1; C2 }
```

where the parts inside the curly braces are arbitrary PROMELA fragments. Execution of this `unless` statement begins with the execution of `B1`. Before each statement execution in the sequence `B1;B2;B3`, the executability of the first statement, or guard, of fragment `C1` is checked using the normal PROMELA semantics of executability. Execution of statements from `B1;B2;B3` proceeds only while the guard statement of `C1` remains unexecutable. The first instant that this guard of the escape sequence is found to be executable, control changes to it, and execution continues as defined for `C1;C2`. Individual statement executions remain indivisible, so control can only change from inside `B1;B2;B3` to the start of `C1` in between individual statement executions. If the guard of the escape sequence does not become executable during the execution of `B1;B2;B3`, it is skipped when `B3` terminates.

Another example of the use of `unless` is:

```
A;
do
:: b1 -> B1
:: b2 -> B2
...
od unless { c -> C };
D
```

The curly braces around the main or the escape sequence may be deleted if there can be no confusion about which statements belong to those sequences. In the example, condition `c` acts as a watchdog on the repetition construct from the main sequence. Note that this is not necessarily equivalent to the construct:

```
A;
do
:: b1 -> B1
:: b2 -> B2
...
:: c -> break
od;
C; D
```

if `B1` or `B2` are non−empty. In the first version of the example, execution of the iteration can be interrupted at any point inside each option sequence. In the second version, execution can only be interrupted at the start of the option sequences.

Notes

In the presence of rendezvous operations, the precise effect of an `unless` construct can be hard to assess. See the discussion in Chapter 7 for details on resolving apparent semantic conflicts.

See Also

atomic, do, if, sequence

xr

Name

`xr`, `xs` – for defining channel assertions.

Syntax

`xr` name [, name] *

`xs` name [, name] *

Description

Channel assertions such as

```
xr q1;
xs q2;
```

can only appear within a `proctype` declaration. The channel assertions are only valid if there can be at most one single instantiation of the `proctype` in which they appear.

The first type of assertion, `xr`, states that the executing process has exclusive read–access to the channel that is specified. That is, it is asserted to be the only process in the system (determined by its process instantiation number) that can receive messages from the channel.

The second type of assertion, `xs`, states that the process has exclusive write–access to the channel that is specified. That is, it is asserted to be the only process that can send messages to the channel.

Channel assertions have no effect in simulation runs. With the information that is provided in channel assertions, the partial order reduction algorithm that is normally used during verification, though, can optimize the search and achieve significantly greater reductions.

Any test on the contents or length of a channel referenced in a channel assertion, including receive poll operations, counts as both a read and a write access of that channel. If such access conflicts with a channel assertion, it is flagged as an error during the verification. If the error is reported, this means that the additional reductions that were applied may be invalid.

The only channel poll operations that are consistent with the use of channel assertions are `nempty` and `nfull`. Their predefined negations `empty` and `full` have no similar benefit, but are included for symmetry. The grammar prevents circumvention of the type rules by attempting constructions such as `!nempty(q)`, or `!full(q)`.

Summarizing: If a channel−name appears in an `xs(xr)` channel assertion, messages may be sent to (received from) the corresponding channel by only the process that contains the assertion, and that process can only use send (receive) operations, or one of the predefined operators `nempty` or `nfull`. All other types of access will generate run−time errors from the verifier.

Examples

```
chan q = [2] of { byte };
chan r = [2] of { byte };

active proctype S()
{       xs q;
        xr r;

        do
        :: q!12
        :: r?0 -> break
        od
}
active proctype R()
{
        xr q;
        xs r;

        do
        :: q?12
        :: r!0 -> break
        od
}
```

Notes

Channel assertions do not work for rendezvous channels.

For channel arrays, a channel assertion on any element of the array is applied to all elements.

In some cases, the check for compliance with the declared access patterns is too strict. This can happen, for instance, when a channel name is used as a parameter in a `run` statement, which is counted as both a read and a write access.

Another example of an unintended violation of a channel assertion can occur when a single process can be instantiated with different process instantiation numbers, depending on the precise moment that the process is instantiated in a run. In cases such as these, the checks on the validity of the channel assertions can be suppressed, while maintaining the reductions they allow. To do so, the verifier `pan.c` can be compiled with directive `−DXUSAFE`. Use with caution.

See Also

chan, len, nempty, nfull, send, receive

Name

A one sentence synopsis of the language construct and its main purpose.

Syntax

The syntax rules for the language construct. Optional terms are enclosed in (non−quoted) square brackets. The Kleene star `*` is used to indicate zero or more repetitions of an optional term. When the special symbols `'['`, `']'`, or `'*'`, appear as literals, they are quoted. For instance, in

```
chan name = '['const']' of { typename [, typename ] * }
```

the first two square brackets are literals, and the last two enclose an optional part of the definition that can be repeated zero or more times. The terms set in italic, such as name, const, and typename, refer to the grammar rules that follow.

EXECUTABILITY

Defines all conditions that must be satisfied for a basic statement from the fourth section to be eligible for execution. Some standard parts of these conditions are assumed and not repeated throughout. One such implied condition is, for instance, that the executing process has reached the point in its code where the basic statement is defined. Implied conditions of this type are defined in the description of PROMELA semantics in Chapter 7. If the executability clause is described as true, no conditions other than the implied conditions apply.

EFFECT

Defines the effect that the execution of a basic statement from the fourth section will cause on the system state. One standard part of the effect is again always implied and not repeated everywhere: the execution of the statement may change the local state of the executing process. If the effect clause is described as `none`, no effect other than the implicit change in local state is defined. See also the PROMELA semantics description in Chapter 7.

DESCRIPTION

Describes in informal terms the purpose and use of the language construct that is defined.

Examples

Gives some typical applications of the construct.

Notes

Adds some additional notes about special circumstances or cautions.

See Also

Gives references to other manual pages that may provide additional explanations.