EMBEDDED C CODE 17

"The purpose of analysis is not to compel belief but rather to suggest doubt."
(Imre Lakatos, Proofs and Refutations)

SPIN, versions 4.0 and later, support the inclusion of embedded C code into PROMELA models through the following five new primitives:

```
c_expr, c_code, c_decl, c_state, c_track
```

The purpose of these new primitives is primarily to provide support for automatic model extraction from C code. This means that it is not the intent of these extensions to be used in manually constructed models. The primitives provide a powerful extension, opening SPIN models to the full power of, and all the dangers of, arbitrary C code. The contents of the embedded code fragments cannot be checked by SPIN, neither in the parsing phase nor in the verification phase. They are trusted blindly and copied through from the text of the model into the code of the verifier that SPIN generates. In particular, if a piece of embedded C code contains an illegal operation, like a divide by zero operation or a nil-pointer dereference, the result can be a crash of the verifier while it performs the model checking. Later in this chapter we will provide some guidance on locating the precise cause of such errors if you accidentally run into them.

The verifiers that are generated by SPIN version 4.0 and higher use the embedded code fragments to define state transitions as part of a PROMELA model. As far as SPIN is concerned, a `c_code` statement is an uninterpreted state transformer, defined in an external language, and a `c_expr` statement is a user-defined boolean guard, similarly defined in an external language. Since this ''external'' language (C) cannot be interpreted by SPIN itself, simulation runs now have to be performed in a different way, as we will discuss. All verifications can be performed as before, though, with the standard C compiler

providing the required interpretation of all embedded code.

The primitives `c_decl` and `c_state` deal with various ways of declaring data types and data objects in C that either become part of the state vector, or that are deliberately hidden from it. The `c_track` primitive is used to instrument the code of the verifier to track the value of data objects holding state information that are declared elsewhere, perhaps even in in separately compiled code that is linked with the SPIN-generated verifier.

Because the SPIN parser does not attempt to interpret embedded C code fragments, random and guided simulation can no longer be done directly by SPIN itself. To account for this, the SPIN-generated verifiers are now provided with their own built-in error trail playback capability if the presence of embedded C code is detected.

AN EXAMPLE

We will illustrate the use of these features with the example shown in Figure 17.1. The `c_decl` primitive introduces a new data type named `Coord`. To avoid name clashes, the new data type name should not match any of the existing type names that are already used inside the SPIN-generated verifiers. The C compiler will complain if this accidentally happens; SPIN itself cannot detect these conflicts.

Because the new data type name may need to be referenced in other statements, we must secure that its definition is placed high up in the generated code for the verifiers. The `c_decl` statement accomplishes precisely that. The `c_decl` statement, then, is *only* meant to be used for the definition of new C data types, that may be referred to elsewhere in the model.

The `c_state` primitive introduces a new global data object `pt` of type `Coord` into the state vector. The object is initialized to zero.

There is only one active process in this model. It reinitializes the global variable `pt` to zero (in this case this is redundant), and then executes a loop. The loop continues until the elements of structure `pt` differ, which will, of course, happen after a single iteration. When the loop terminates, the elements of the C data object `pt` are printed. To make sure an error trail is generated, the next statement is a *false* assertion.

Arbitrary C syntax can be used in any `c_code` and `c_expr` statement. The difference between these two types of statements is that a `c_code` statement is always executed unconditionally and atomically, while a `c_expr` statement can only be executed (passed) if it returns non-zero when its body is evaluated as a C expression. If the evaluation returns zero, execution is blocked. The evaluation of a `c_expr` is again indivisible (i.e., atomic). Because SPIN may have to evaluate `c_expr` statements repeatedly until one of them becomes executable, a `c_expr` is required to be free from side effects: it may only evaluate data, not modify it.

```
c_decl {
   typedef struct Coord {
        int x, y;
   } Coord;
}

c_state "Coord pt" "Global" /* goes inside state vector */

int z = 3;               /* standard global declaration */

active proctype example()
{
   c_code { now.pt.x = now.pt.y = 0; };

   do
   :: c_expr { now.pt.x == now.pt.y } ->
              c_code { now.pt.y++; }
   :: else ->
              break
   od;
   c_code {
        printf("values %d: %d, %d,%d\n",
          Pexample->_pid, now.z, now.pt.x, now.pt.y);
   };
   assert(false)        /* trigger an error trail */
}
```

Figure 17.1 Example of Embedded C Code

DATA REFERENCES

A *global* data object that is declared with the normal PROMELA declaration syntax in the model (i.e., not with the help of `c_code` or `c_state`) can be referenced from within `c_code` and `c_expr` statements, but the reference has to be prefixed in this case with the string `now` followed by a period. In the example, for instance, the global `z` can be referenced within a `c_code` or `c_expr` statement as `now.z`. (The name `now` refers to the internal state vector, where all global data is stored during verification.) Outside embedded C code fragments, the same variable can be referenced simply as `z`.

A process *local* data object can also be referenced from within `c_code` and `c_expr` statements within the same process (i.e., if the object is declared within the current scope), but the syntax is different. The extended syntax again adds a special prefix that locates the data object in the state vector. The prefix starts with an uppercase letter `P` which is followed by the name of the `proctype` in which the reference occurs, followed by the pointer arrow. For the data objects declared locally in proctype `example`, for instance, the prefix

to be used is `Pexample->`.

In the example, this is illustrated by the reference to the predefined local variable `_pid` from within the `c_code` statement as `Pexample->_pid`.

The `_pid` variable of the process can be referenced, within the `init` process itself, as `Pinit->_pid`.

Another way to write this particular model is shown in Figure 17.2. In this version we have avoided the need for the prefixes on the variable names, by making use of the `c_track` primitive. The differences with the version in Figure 17.1 are small, but important.

We have declared the variable `pt` in a global `c_code` statement, which means that it gets included this time as a regular global variable that remains outside the state vector. Since this object holds state information, we add a `c_track` statement, specifying a pointer to the object and its size. SPIN will now arrange for the value of the object to be copied into (or out of) a specially reserved part of the state vector on each step. This is obviously less efficient than the method using `c_state`, but it avoids the need for the sometimes clumsy `now.` prefixes that are required for references to objects that are placed directly into the state vector. Note that the reference to variable `z` still requires this prefix, since it was declared as a normal global PROMELA variable, and similarly for the predefined local variable `_pid`. If we lifted the `printf` statement outside the enclosure of the `c_code` primitive, we could refer to variables `z` and `_pid` without a prefix, as regular PROMELA variables, but we could not refer to the C variable `pt` at all; these external objects are only visible inside `c_code`, `c_expr`, and `c_track` statements.

EXECUTION

When a PROMELA model contains embedded C code, SPIN cannot simulate its execution in the normal way because it cannot directly interpret the embedded code fragments. If we try to run a simulation anyway, SPIN will make a best effort to comply, but it will only print the *text* of the `c_expr` and `c_code` fragments that it encounters, without actually executing them.

To faithfully execute all embedded C code fragments, we must first generate the `pan.[chmbt]` files and compile them. We now rely on the standard C compiler to interpret the contents of all embedded code as part of the normal compilation process. For the first example, we proceed as follows:

```
$ spin -a example
$ cc -o pan pan.c       # compile
$ ./pan                 # and run
values 0: 3, 0,1
pan: error: assertion violated 0 (at depth 5)
pan: wrote coord.trail
```

The assertion violation was reported, as expected, but note that the embedded

```
c_decl {
   typedef struct Coord {
        int x, y;
   } Coord;
}

c_code { Coord pt; }          /* embedded declaration */
c_track "&pt" "sizeof(Coord)" /* track value of pt    */

int z = 3;             /* standard global declaration */

active proctype example()
{
   c_code { pt.x = pt.y = 0; }; /* no 'now.' prefixes */

   do
   :: c_expr { pt.x == pt.y } ->
                c_code { pt.y++; }
   :: else ->
                break
   od;
   c_code {
        printf("values %d: %d, %d,%d\n",
          Pexample->_pid, now.z, pt.x, pt.y);
   };
   assert(false)        /* trigger an error trail */
}
```

Figure 17.2 Replacing c_state with c_track Primitives

printf statement was also executed, which shows that it works differently from a PROMELA print statement. We can get around this by calling an internal SPIN routine named Printf instead of the standard library routine printf within embedded c_code fragments. This causes the verifier to enable the execution of the print statement only when reproducing an error trail, but not during the verification process itself.

The counterexample is stored in a trail file as usual, but SPIN itself cannot interpret the trail file completely because of the embedded C code statements that it contains. If we try anyway, SPIN produces something like this, printing out the embedded fragments of code without actually executing them:

```
$ spin -t -p example
c_code2: {  now.pt.x = now.pt.y = 0;  }
  1: proc 0 (example) line  11 ... (state 1) [{c_code2}]
```

```
c_code3: now.pt.x == now.pt.y
  2: proc 0 (example) line  14 ... (state 2) [({c_code3})]
c_code4: {  now.pt.y++;  }
  3: proc 0 (example) line  15 ... (state 3)  [{c_code4}]
  4: proc 0 (example) line  16 ... (state 4) [else]
c_code5: {  printf("values %d: %d %d,%d\n", \
        Pexample->_pid, now.z now.pt.x, now.pt.y);  }
  5: proc 0 (example) line  19 ... (state 9) [{c_code5}]
spin: line 20 ..., Error: assertion violated
spin: text of failed assertion: assert(0)
  6: proc 0 (example) line  20 ... (state 10) [assert(0)]

spin: trail ends after 6 steps
#processes: 1
  6: proc  0 (example) line  21 ... (state 11)
1 process created
```

The assertion is violated at the end, but this is merely because it was hard-wired to fail. None of the C data objects referenced were ever created during this run, and thus none of them had any values that were effectively assigned to them at the end. Note also that the text of the `c_code` fragment that is numbered `c_code5` here is printed out, but that the print statement that it contains is not itself executed, or else the values printed would have shown up in the output near this line.

It is better to use the trail replay option that is now available inside the generated `pan` verifier. The additional options are:

```
$ ./pan --
 ...
  -C  read and execute trail - columnated output
  -PN read and execute trail - restrict output to proc N
  -r  read and execute trail - default output
 ...
```

With the first of these options, the verifier produces the following information on the execution of the trail:

```
$ ./pan -C
1: example(0):[ now.pt.x = now.pt.y = 0; ]
2: example(0):[( now.pt.x == now.pt.y )]
3: example(0):[ now.pt.y++; ]
4: example(0):[else]
values 0: 3, 0,1
5: example(0):[ printf("values: %d,%d\n", \
        now.pt.x, now.pt.y); ]
pan: error: assertion violated 0 (at depth 6)
spin: trail ends after 6 steps
#processes 1:
```

```
  6: proc 0 (example)  line  20 (state 10)
                assert(0)
global vars:
        int    z:       3
local vars proc 0 (example):
        (none)
```

Note that in this run, the print statement was not just reproduced but also executed. Similarly, the data object `pt` was created, and its value is updated in the `c_code` statements so that the final values of its elements `pt` accurately reflect the execution. There is only one process here, with `_pid` value zero, so the columnation feature of this format is not evident.

More information can be added to the output by adding option `-v`. Alternatively, all output except the ones that are generated by explicit print statements in the model can be suppressed by adding option `-n`.

In long and complex error trails with multiple process executions, it can be helpful to restrict the trail output to just one of the executing processes. This can be done with the help of option `-P`, which should be followed by the pid number of the process of interest.

For a more detailed explanation of the special declarators `c_decl` and `c_track`, we point to the manual pages that follow at the end of this chapter.

ISSUES TO CONSIDER

The capability to embed arbitrary fragments of C code into a PROMELA model is powerful and therefore easily misused. The intent of these features is to support mechanized model extractors that can automatically extract an accurate, possibly abstract, representation of application level C code into a SPIN verification model. The model extractor (see Appendix D) can include all the right safeguards that cannot easily be included in SPIN without extending it into a full ANSI-C compiler and analyzer. Most of the errors that can be made with the new primitives will be caught, but not necessarily directly by SPIN. The C compiler, when attempting to compile a model that contains embedded fragments of code, may object to ill-defined structures, or the verifier may crash on faults that can be traced back to coding errors in the embedded code fragments.

If data that is manipulated inside the embedded C code fragments contains relevant state information, but is not declared as such with `c_state` or `c_track` primitives, then the search process itself can get confused, and error trails may be produced by the verifier that do not correspond to feasible executions of the modeled system. With some experience, these types of errors are relatively easy to diagnose. Formally, they correspond to invalid ‘‘abstractions’’ of the model. The unintended ‘‘abstractions’’ are caused by missing `c_state` or `c_track` primitives.

To see what happens when we forget to treat externally declared data objects

as carrying state information, consider the following simple model:

```
c_code { int x; }

active proctype simple()
{
   c_code { x = 2; };
   if
   :: c_code { x = x+2; }; assert(c_expr { x==4 })
   :: c_code { x = x*3; }; assert(c_expr { x==6 })
   fi
}
```

We have declared the variable `x` in a `c_code` statement, but omitted to track its value. The verifier will therefore ignore value changes in this variable when it stores and compares states, although it will faithfully perform every assignment or test of this variable in the execution of the model.

At first sight, it would seem obvious that neither one of the two could possibly fail, but when we perform the verification we see:

```
$ spin -a simple1.pr
$ cc -o pan pan.c
$ ./pan
pan: assertion violated (x == 6)
pan: wrote simple.pr.trail
...
```

To understand the reason for this error, consider for a moment how the depth-first search process proceeds in this case. The verifier starts by executing the assignment

```
c_code { x = 2; };
```

Next, it has the choice between two executable statements. It can either increment the value of `x` by two, or it can multiply it by three. As it happens, it will choose to try the first alternative first. It executes

```
c_code { x = x+2; }; assert(c_expr { x==4 })
```

Not surprisingly, the assertion holds. The search now reaches the end of the execution: there are no further statements to execute in this model. So, the depth-first search reverses and backs up to the point where it had to make a choice between two possible ways to proceed: at the start of the `if` statement. The verifier restores the state of the system to the control flow point at the start of the `if` statement, but since the variable `x` is not treated as a state variable, its value remains unchanged at this point. The search now proceeds, with `x` having the value four. The multiplication that is now executed to explore the second option sequence

```
c_code { x = x*3; }; assert(c_expr { x==6 })
```

which results in the unexpected value of twelve for `x`. As a result, the second

assertion fails. The counterexample that is generated will clearly show that there is confusion about the true value of x, which is the hint we can use to correct the model by supplying the missing c_track statement.

```
c_code { int x; }
c_track "&x" "sizeof(int)"

active proctype simple()
{
   c_code { x = 2; };
   if
   :: c_code { x = x+2; }; assert(c_expr { x==4 })
   :: c_code { x = x*3; }; assert(c_expr { x==6 })
   fi
}
```

Verification now produces the expected result:

```
$ spin -a simple2.pr
$ cc -o pan pan.c
$ ./pan
(Spin Version 4.0.7 -- 1 August 2003)
        + Partial Order Reduction

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        acceptance   cycles     - (not selected)
        invalid end states      +

State-vector 16 byte, depth reached 4, errors: 0
       8 states, stored
       0 states, matched
       8 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)

1.573   memory usage (Mbyte)

unreached in proctype simple
        (0 of 8 states)
```

How does one determine which external data objects from an application contain state information and which do not? This is ultimately a matter of judgement, and lacking proper judgement, a process of discovery. The determination can be automated, to some extent, for a given set of logic properties. A data dependency analysis may be used to determine what is relevant and what is not. A more detailed discussion of this issue, though important, is beyond the scope of this book.

DEFERRING FILE INCLUSION

It is often convenient to include a collection of C code into a model with a preprocessor `include` directive, for instance, as follows:

```
#include "promela.h"    /* Promela data definitions */

c_decl {
#include "c_types.h"    /* C data type definitions */
}

c_code {
#include "functions.c"  /* C function definitions */
}

#include "model.pr"     /* the Promela model itself */
```

When SPIN invokes the C preprocessor on this model, the contents of the included files are inserted into the text before the model text is parsed by the SPIN parser. This works well if the files that are included are relatively small, but since there is a limit on the maximum size of a `c_code` or `c_decl` statement, this can fail if the files exceed that limit. (At the time of writing, this limit is set at 64Kbytes.)

There is an easy way to avoid hitting this limit. Because C code fragments are not interpreted until the verifier code is parsed by the C compiler, there is no need to actually have the body of a `c_code` or `c_decl` statement inserted into the text of the model before it is passed to the SPIN parser. We can achieve this by prefixing the pound sign of the corresponding `include` directives with a backslash, as follows:

```
c_decl {
\#include "c_types.h"   /* C data type definitions */
}

c_code {
\#include "functions.c" /* C function definitions */
}
```

The SPIN parser will now simply copy the `include` directive itself into the generated C code, without expanding it first. The backslash can only be used in this way inside `c_decl` and `c_code` statements, and it is the recommended way to handle included files in these cases.

NAME

`c_code` – embedded C code fragments.

SYNTAX

c_code { `/* c code */` }
c_code `'['` `/* c expr */` `']'` { `/* c code */ ;` }

EXECUTABILITY

true

EFFECT

As defined by the semantics of the C code fragment placed between the curly braces.

DESCRIPTION

The `c_code` primitive supports the use of embedded C code fragments inside PROMELA models. The code must be syntactically valid C, and must be terminated by a semicolon (a required statement terminator in C).

There are two forms of the `c_code` primitive: with or without an embedded expression in square brackets. A missing expression clause is equivalent to `[ 1 ]`. If an expression is specified, its value will be evaluated as a general C expression *before* the C code fragment inside the curly braces is executed. If the result of the evaluation is non-zero, the `c_code` fragment is executed. If the result of the evaluation is zero, the code between the curly braces is ignored, and the statement is treated as an assertion violation. The typical use of the expression clause is to add checks for nil-pointers or for bounds in array indices. For example:

```
c_code [Pex->ptr != 0 && now.i < 10 && now.i >= 0] {
        Pex->ptr.x[now.i] = 12;
}
```

A `c_code` fragment can appear anywhere in a PROMELA model, but it must be meaningful within its context, as determined by the C compiler that is used to compile the complete model checking program that is generated by SPIN from the model.

Function and data declarations, for instance, can be placed in global `c_code` fragments preceding all `proctype` definitions. Code fragments that are placed inside a `proctype` definition cannot contain function or data declarations. Violations of such rules are caught by the C compiler. The SPIN parser merely passes all C code fragments through into the generated verifier uninterpreted, and therefore cannot detect such errors.

There can be any number of C statements inside a `c_code` fragment.

EXAMPLES

```
int q;
c_code { int *p; };
init {
        c_code { *p = 0; *p++; };
        c_code [p != 0] { *p = &(now.q); };
        c_code { Printf("%d\n", Pinit->_pid); }
}
```

In this example we first declare a normal PROMELA integer variable `q` that automatically becomes part of the verifier's internal state vector (called `now`) during verification. We also declare a global integer pointer `p` in a global `c_code` fragment. Since the contents of a C code fragment are not interpreted by SPIN when it generates the verifier, SPIN cannot know about the presence of the declaration for pointer variable `p`, and therefore this variable remains invisible to the verifier: its declaration appears outside the state vector. It can be manipulated as shown as a regular global pointer variable, but the values assigned to this variable are *not* considered to be part of the global system state that the verifier tracks.

To arrange for data objects to appear inside the state vector, and to be treated as system state variables, one or more of the primitives `c_decl`, `c_state`, and `c_track` should be used (for details, see the corresponding manual pages).

The local `c_code` fragment inside the `init` process manipulates the variable `p` in a direct way. Since the variable is not moved into the state vector, no prefix is needed to reference it.

In the second `c_code` fragment in the body of `init`, an expression clause is used that verifies that the pointer `p` has a non-zero value, which secures that the dereference operation that follows cannot result in a memory fault. (Of course, it would be wiser to add this expression clause also to the preceding `c_code` statement.) When the `c_code` statement is executed, the value of `p` is set to the address of the PROMELA integer variable `q`. Since the PROMELA variable is accessed inside a `c_code` fragment, we need a special prefix to identify it in the global state vector. For a global variable, the required prefix is the three-letter word `now` followed by a period. The ampersand in `&(now.q)` takes the address of the global variable within the state vector.

The last `c_code` statement in `init` prints the value of the process identifier for the running process. This is a predefined local variable.

To access the local variable in the `init` process, the required prefix is `Pinit->`. This format consists of the uppercase letter `P`, followed by the name of the process type, followed by an arrow `->`.

See also the description on data access in `c_expr`.

NOTES

The embedded C code fragments must be syntactically correct and complete. That is, they must contain proper punctuation with semicolons, using the standard semantics from C, not from PROMELA. Note, for instance, that semicolons are statement terminators in C, but statement separators in PROMELA.

Because embedded C code is not interpreted by the SPIN parser, `inline` parameter substitutions are not applied to those code fragments. In cases where this is needed, the `inline` definitions can be replaced with macro preprocessor definitions.

A common use of the `c_code` primitive is to include a larger piece of code into a model that is stored in a separate file, for instance, as follows:

```
c_code {
#include "someheaders.h"
#include "someCcode.c"
};
```

If the included code fragment is too large (in the current implementation of SPIN this means larger than about 64Kbyte of text), SPIN will complain about that and fail. A simple way to bypass this restriction, for instance, when generating the verification code with SPIN's `-a` option, is to defer the interpretation of the `include` directives by the SPIN preprocessor, and to copy them through into the generated code unseen. This can be accomplished as follows, by placing a backslash before the pound sign of any `include` directive that appears inside a `c_code` primitive.

```
c_code {
\#include "someheaders.h"
\#include "someCcode.c"
};
```

Functionally, this is identical to the previous version, but it makes sure that the SPIN preprocessor will not read in the text of the included files when the model is parsed.

SEE ALSO

c_expr, **c_decl**, **c_state**, **c_track**, **macros**

NAME

`c_decl`, `c_state`, `c_track` – embedded C data declarations.

SYNTAX

c_decl { `/* c declaration */` }
c_state string string [*string*]
c_track string string

EXECUTABILITY

true

DESCRIPTION

The primitives `c_decl`, `c_state`, and `c_track` are *global* primitives that can only appear in a model as global declarations outside all `proctype` declarations.

The `c_decl` primitive provides a capability to embed general C data type declarations into a model. These type declarations are placed in the generated `pan.h` file *before* the declaration of the state-vector structure, which is also included in that file. This means that the data types introduced in a `c_decl` primitive can be referenced anywhere in the generated code, including inside the state vector with the help of `c_state` primitives. Data type declarations can also be introduced in global `c_code` fragments, but in this case the generated code is placed in the `pan.c` file, and therefore appears necessarily *after* the declaration of the state-vector structure. Therefore, these declarations cannot be used inside the state vector.

The `c_state` keyword is followed by either two or three quoted strings. The first argument specifies the type and the name of a data object. The second argument the scope of that object. A third argument can optionally be used to specify an initial value for the data object. (It is best not to assume a known default initial value for objects that are declared in this way.)

There are three possible scopes: global, local, or hidden. A global scope is indicated by the use of the quoted string `"Global."` If local, the name `Local` must be followed by the name of the `proctype` in which the declaration is to appear, as in `"Local ex2."` If the quoted string `"Hidden"` is used for the second argument, the data object will be declared as a global object that remains *outside* the state vector.

The primitive `c_track` is a global primitive that can declare any state object, or more generally any piece of memory, as holding state information. This primitive takes two string arguments. The first argument specifies an address, typically a pointer to a data object declared elsewhere. The second argument gives the size in bytes of that object, or more

generally the number of bytes starting at the address that must be tracked as part of the system state.

EXAMPLES

The first example illustrates how `c_decl`, `c_code` and `c_state` declarations can be used to define either visible or hidden state variables, referring to type definitions that must precede the internal SPIN state-vector declaration. For an explanation of the rules for prefixing global and local variables inside `c_code` and `c_expr` statements, see the manual pages for these two statements.

```
c_decl {
        typedef struct Proc {
                int rlock;
                int state;
                struct Rendez *r;
        } Proc;

        typedef struct Rendez {
                int     lck;
                int     cond;
                Proc    *p;
        } Rendez;
}
c_code {
        Proc    H1;
        Proc    *up0 = &H1;
        Rendez  RR;
}

/*
 * The following two c_state declarations presume type
 * Rendez known the first enters R1 into state vector
 * as a global variable, and the second enters R2 into
 * proctype structure as local variable.
 */

c_state "Rendez R1" "Global"
c_state "Rendez R2" "Local ex2" "now.R1"

/*
 * The next two c_state declarations are kept outside
 * the state vector. They define H1 and up0 as global
 * objects, which are declared elsewhere.
 */

c_state "extern Proc H1" "Hidden"
c_state "extern Proc *up0" "Hidden"
```

```
/*
 * The following declaration defines that RR is to be
 * treated as a state variable -- no matter how it was
 * declared; it can be an arbitrary external variable.
 */

c_decl {
\#include "types.h"         /* declare type Rendez */
/* for the purpose of the backslash, see p. 504 */
}

c_track "&RR" "sizeof(Rendez)"

active proctype ex2()
{
        c_code { now.R1.cond = 1; };    /* global */
        c_code { Pex2->R2.lck = 0; };   /* local  */
        c_code { H1.rlock = up0->state; }; /* C */

        printf("This is Spin Version 4.0\n")
}
```

NOTES

SPIN instruments the code of the verifier to copy all data pointed to via `c_track` primitives into and out of the state vector on forward and backward moves during the depth-first search that it performs. Where there is a choice, the use of `c_state` primitives will always result in more efficiently executed code, since SPIN can instrument the generated verifier to directly embed data objects into the state vector itself, avoiding the copying process.

To get a better feeling for how precisely these primitives are interpreted by SPIN, consider generating code from the last example, and look in the generated files `pan.h` and `pan.c` for all appearances of variables `R1`, `R2`, `P1`, and `up0`.

Avoid using type-names that clash with internal types used within the SPIN-generated verifiers. This includes names such as `State`, `P0`, `P1`, etc., and `Q0`, `Q1`, etc. Name clashes caused by unfortunate choices of type names are reliably caught by the C compiler when the verification code is compiled.

SEE ALSO

c_expr, **c_code**

NAME

`c_expr` – conditional expressions as embedded C code.

SYNTAX

c_expr `{ /* c code */ }`
c_expr `'[' /* c expr */ ']' { /* c code */ }`

EXECUTABILITY

If the return value of the arbitrary C code fragment that appears between the curly braces is non-zero, then *true*; otherwise *false*.

EFFECT

As defined by the semantics of the C code fragment that is placed between the curly braces. The evaluation of the C code fragment should have no side effects.

DESCRIPTION

This primitive supports the use of embedded C code inside PROMELA models. A `c_expr` can be used to express guard conditions that are not necessarily expressible in PROMELA with its more restrictive data types and language constructs.

There are two forms of the `c_expr` primitive: with or without an additional assertion expression in square brackets. A missing assertion expression is equivalent to `[ 1 ]`. If an assertion expression is specified, its value is evaluated as a general C expression *before* the code inside the curly braces is evaluated. The normal (expected) case is that the assertion expression evaluates to a *non-zero* value (that is to an equivalent of the boolean value *true*). If so, the C code between the curly braces is evaluated next to determine the executability of the `c_expr` as a whole.

If the evaluation value of the assertion expression is zero (equivalent to *false*), the code between the curly braces is ignored and the statement is treated as an assertion violation.

The typical use of the assertion expression clause is to add checks for nil-pointers or for possible array bound violations in expressions. For example:

```
c_expr [Pex->ptr != NULL] { Pex->ptr->y }
```

Note that there is no semicolon at the end of either C expression. If the expression between square brackets yields *false* (zero), then an assertion violation is reported. Only if this expression yields *true* (non-zero), is the C expression between curly braces evaluated. If the value of this second expression yields *true*, the `c_expr` as a whole is deemed executable and can be passed; if *false*, the `c_expr` is unexecutable and blocks.

EXAMPLES

The following example contains a do-loop with four options. The first two options are equivalent, the only difference being in the way that local variable `x` is accessed: either via an embedded C code fragment or with the normal PROMELA constructs.

```
active proctype ex1()
{       int x;

        do
        :: c_expr { Pex1->x < 10 } ->
                c_code { Pex1->x++; }
        :: x < 10 -> x++
        :: c_expr { fct() } -> x--
        :: else -> break
        od
}
```

The local variable `x` is declared here as a PROMELA variable. Other primitives, such as `c_decl`, `c_state`, and `c_track` allow for the declaration of data types that are not directly supported in PROMELA.

The references to local variable `x` have a pointer prefix that always starts with a fixed capital letter `P` that is followed by the name of the `proctype` and an pointer arrow. This prefix locates the variable in the local state vector of the `proctype` instantiation.

The guard of the third option sequence invokes an externally defined C function named `fct()` that is presumed to return an integer value. This function can be declared in a global `c_code` fragment elsewhere in the model, or it can be declared externally in separately compiled code that is linked with the `pan.[chtmb]` verifier when it is compiled.

NOTES

Note that there is no semicolon before the closing curly brace of a `c_expr` construct. It causes a C syntax error if such a semicolon appears here. All syntax errors on embedded C code fragments are reported during the compilation of the generated `pan.[chtmb]` files. These errors are not detectable by the SPIN parser.

Because embedded C code is not processed by the SPIN parser, `inline` parameter substitutions are not applied to those code fragments. In cases where this is needed, the `inline` definitions can be replaced with macro preprocessor definitions.

SEE ALSO

c_code, **c_decl**, **c_state**, **c_track**, **macros**