EMBEDDED C CODE 17

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

SPIN Version 4 supports 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 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 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.

The verifiers that are generated with SPIN version 4 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, also defined in an external language. Since this ''external'' language cannot be interpreted by SPIN itself, simulations have to be done in a special way, but all verifications can be performed largely as before: the C compiler provides the necessary interpretation.

The next three primitives deal with various ways of declaring data and datatypes in C that either become part of the statevector, that are deliberately hidden from the statevector, or that refer to data objects declared in separately compiled code that is linked with the SPIN generated verifier.

If embedded C code is present in a model, the playback of counter-examples can no longer be done through SPIN's builtin guided simulation options. The SPIN generated verifiers, therefore, now have their own builtin error-trail playback capability. We will illustrate its use with an example.

AN EXAMPLE

We will discuss a short example of a model that makes use of some of the new features.

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

c_state "Coord pt" "Global"

int z = 3;      /* a 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 */
}
```

The `c_decl` primitive introduces a new data type named `Coord.` The new data type name may not match any of the existing type names that are already used inside the SPIN generated verifiers. The compiler will complain if this accidentally happens.

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

There is only one active process in this model. It re-initializes 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 of course will 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 will always be 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. The evaluation of a c_expr is again indivisible (i.e., atomic). Because SPIN may have to evaluate c_expr statements repeatedly, until it is found to be executable, a c_expr must always be side-effect free: it may only evaluate data, and not modify it.

DATA REFERENCES

A *global* data object that is declared with the standard declaration syntax in the PROMELA model (i.e., not with the help of c_decl, c_state, or c_track) 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 above, for instance, the global z can be referenced within a c_code or c_expr statement as now.z. Outside embedded C code fragments, the same variable can be referenced simply as z.

Any process *local* data object can also be referenced from within c_code and c_expr statements, but the syntax is different. The extended syntax again adds a special prefix that locates each data object in the model checker's state vector. The prefix starts with an upper-case 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 above, 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.

EXECUTION

When a PROMELA model contains embedded C code, SPIN will not be able to simulate its execution in the normal way, because it cannot directly parse or execute the portions of code that are enclosed in c_code or c_expr statements. If we try to run a simulation anyway, SPIN will make a best effort to comply, but it will only print the c_expr and c_code fragments as uninterpreted text and it will not try to execute. For the SPIN simulator these primitives are treated as if they were PROMELA skip statement.

To get a real execution that includes a full execution of all c_code and c_expr statements, we need must first need generate the pan.[chmbt] files, and compile it. We are now relying on the standard C compiler to accurately interpret the contents of all `c_code` and `c_expr` statements as part of the normal compilation of all generated code. We proceed as follows:

```
$ spin -a example       # SPIN Version 4
$ cc -o pan pan.c       # compile as usual
$ ./pan                 # run
values 0: 3, 0,1
pan: error: assertion violated 0 (at depth 5)
pan: wrote coord.trail
(Spin Version 4.0.0 -- 3 May 2002)
Warning: Search not completed
        + Partial Order Reduction

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

State-vector 20 byte, depth reached 5, errors: 1
       6 states, stored
       0 states, matched
       6 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)

1.573   memory usage (Mbyte)
```

The assertion violation was reported as expected, but note that the embedded `printf` statement was also executed, which shows that it works differently from a PROMELA print statement. We can get around this by using `Printf` instead of `printf` inside 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.

Now we have a counter-example, stored in a `pan.trail` file as usual, but again SPIN cannot interpret the trail file properly, because it has embedded C code in it. If we try anyway, SPIN produces something like this, printing out the embedded fragments of code without actually also executing their contents:

```
$ spin -t -p ex0
c_code2: {  now.pt.x = now.pt.y = 0;  }
  1: proc  0 (ex0) line  11 "coord" (state 1) [{c_code2}]
c_code3: now.pt.x == now.pt.y
  2: proc  0 (ex0) line  14 "coord" (state 2) [({c_code3})]
c_code4: {  now.pt.y++;  }
  3: proc  0 (ex0) line  15 "coord" (state 3)  [{c_code4}]
  4: proc  0 (ex0) line  16 "coord" (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 (ex0) line  19 "coord" (state 9) [{c_code5}]
spin: line  20 "coord", Error: assertion violated
spin: text of failed assertion: assert(0)
  6: proc  0 (ex0) line  20 "coord" (state 10) [assert(0)]

spin: trail ends after 6 steps
#processes: 1
  6: proc  0 (ex0) line  21 "coord" (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.

Better is 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: ex0(0):[ now.pt.x = now.pt.y = 0; ]
2: ex0(0):[( now.pt.x == now.pt.y )]
3: ex0(0):[ now.pt.y++; ]
4: ex0(0):[else]
values 0: 3, 0,1
5: ex0(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 here.

More information can be added to the output by adding option `-v`, or all output except that generated by print statements 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.

For an explanation of the special declarators `c_decl` and `c_track` we point to the manual pages that follow.

ISSUES TO CONSIDER

The capability to embed arbitrary fragments of C code into a PROMELA model is very powerful and therefore easily misused. The intent of these features is to support mechanized preprocessors that can automatically extract verification models from applications that are written in C. The preprocessor can include all the right safeguards that cannot easily be included in SPIN, without extending it with a full ANSI-C parser and analyzer that is. Most of the errors that can be made with the new primitives will be caught, but not necessarily by SPIN itself. The C compiler, when attempting to compile a model that contains embedded fragments of code, may object to ill-defined structures, or during execution the verifier may now get stuck hopelessly on faults that can be traced back to semantics 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 can get confused, and error trails produced by the verifier may not correspond to feasible executions of the modeled system. With some experience, these types of errors are relatively easy to diagnose though. Formally, they correspond to invalid ‘‘abstractions’’ of the model. The unintended ‘‘abstractions’’ then are caused by the missing `c_state` or `c_track` primitives.

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

NAME

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 semi-colon (a required statement terminator in C).

There are two forms of the `c_code` primitive: with our 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 (see **assert(4)**). The typical use of the expression clause is to add checks for nil-pointers or for bounds in array indices. For example, as in:

```
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 they must be meaningful within their context, as determined by the C compiler that is used to compile the complete `pan.[chtmb]` program that is generated by SPIN from the model.

Function and data declarations, for instance, can be placed in global `c_code` fragments, that appear in the model before the `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 to the generated programs uninterpreted, and will therefore not be able to 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 statevector (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 will appear outside the statevector. It can be manipulated as shown as a regular global pointer variable, but the values assigned to this variable will *not* be considered to be part of the global system state that the verifier will track.

To arrange for data objects to appear inside the statevector, and to be treated as system state variables, one or more of the primitives **c_decl(4)**, **c_state(4)**, and **c_track(4)** 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 statevector, 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 statevector. For a global variable, the required prefix is now.. The ampersand in &(now.q) takes the address of the global variable within the statevector.

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->. A capital P, followed by the name of the process type, which in turn is followed by an arrow ->.

See also the description on data access in **c_expr(4)**.

NOTES

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

SEE ALSO

c_expr(4), **c_decl(4)**, **c_state(4)**, **c_track(4)**.

NAME

Embedded C Data Declarations

SYNTAX

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

EXECUTABILITY

`True`

EFFECT

None. These primitives are used to add embedded C data type and data object declarations into a PROMELA model.

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 statevector 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 statevector 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 statevector structure. Therefore these declarations cannot be refered to from 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 statevector.

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 as 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 statevector 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 statevector 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 statevector
 * define H1 and up0 as global objects, 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_track "&RR" "sizeof(Rendez)"
/* RR must be declared elsewhere */

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

        printf("hello 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 executing code, since SPIN can instrument the generated verifier to directly embed data object 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 example above, and look in the generated files `pan.h` and `pan.c` for all appearances of variables `R1`, `R2`, `P1` and `up0`.

Avoid using the typenames `State`, `P0`, `P1`, `...`, and `Q0`, `Q1`, `...`, since these names are also used internally by SPIN. If the same names appear in user code, a name clash will result, which is caught by the C compiler.

SEE ALSO

c_expr(4), **c_code(4)**.

NAME

Conditional Expressions as Embedded C Code

SYNTAX

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

EXECUTABILITY

`True` if the return value of the arbitrary C code fragment that appears between the curly braces is non-zero, and otherwise `False`.

EFFECT

As defined by the semantics of the C code fragment 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 the more restrictive data types and language constructs.

There are two forms of the `c_expr` primitive: with our without an embedded expression in square brackets. A missing expression clause is equivalent to `[ 1 ]`. If an expression is specified between square brackets, its value will be evaluated as a general C expression *before* the expression inside the curly braces is evaluated. If the resulting value is non-zero, the C expression between the curly braces is evaluated next, to determine the executability of the `c_expr` statement as a whole. If the evaluation value of the expression between square brackets is zero, the code between the curly braces is ignored and the statement is treated as an assertion violation (see **assert(4)**). The typical use of the expression clause is to add checks for nil-pointers or for bounds in array indices. For example, as in:

```
c_expr [Pex->ptr != 0] { Pex->ptr->y }
```

Note that there is no semi-colon at the end of either C expression. If the expression between square braces yields *false* (zero), then an assertion violation is reported. Only if this expression yields *true* (non-zero), the C expression between curly braces is 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 it will block execution.

EXAMPLES

The following example contains a single do-loop with four options. The first two option sequences are equivalent, the only difference being in the way that the local variable `x` is accessed: either via embedded C code fragments 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(4)**, **c_state(4)**, and **c_track(4)**, 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 statevector 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. That 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 to create a `pan` executable.

NOTES

Note that there is no semi-colon before the closing curly brace of a `c_expr` construct. It would cause a syntax error if such a semi-colon appears. 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.

SEE ALSO

c_code(4), **c_decl(4)**, **c_state(4)**, **c_track(4)**.