Chapter 10. Notes on Model Extraction

"In all affairs it's a healthy thing now and then to hang a question mark on the things you have long taken for granted."

—(Bertrand Russell, 1872–1970)

Arguably, the most powerful tool we have in our arsenal for the verification of software applications is logical abstraction. By capturing the essence of a design in a mathematical model, we can often demonstrate conclusively that the design has certain inevitable properties. The purpose of a verification model, then, is to enable proof. If it fails to do so, within the resource limits that are available to the verification system, the model should be considered inadequate.

The Role of Abstraction

The type of abstraction that is appropriate for a given application depends both on the type of logical properties that we are interested in proving and on the resource limits of the verification system. This situation is quite familiar: it is no different from the one that applies in standard mathematics. When we reason about the correctness of a system without the benefit of a mechanized prover, using straight logic and pen and paper mathematics, we must also use our judgement in deciding which parts of a system are relevant, and which are not, with respect to the properties to be proven. Similarly, we must be aware of resource limitations in this situation as well. If only a very limited amount of time, or a very limited amount of mathematical talent, is available for rendering the proof, perhaps a coarser proof would have to be used. If unlimited time and talent is available, a more detailed proof may be possible. Whether we choose a mechanized process or a manual one, we have to recognize that some really difficult types of problems may remain beyond our reach. It is the skill of the verifier to solve as much of a problem as is feasible, within given resource limits.

For the best choice of an abstraction method in the construction of a verification model, we unavoidably have to rely on human judgement. Which parts of the system should we look at? What properties should apply? These types of decisions would be hard to automate. But even though there will unavoidably be a human element in the setup of a verification process, once the basic decisions about abstractions are made and recorded, it should in principle be possible to mechanize the remainder of the verification process.

Given the source text of an application and the properties of interest, we would like to generate a verification model automatically from the source text, where the model extraction process is guided by a user−defined abstraction function. In this chapter we discuss how we can do so.

From ANSI−C to PROMELA

To get an impression of what it takes to mechanically convert a C program into a PROMELA model, consider the following program. The program is one of the first examples used in Kernighan and Ritchie's introduction to the C programming language, with a slightly more interesting control structure than the infamous hello world example.

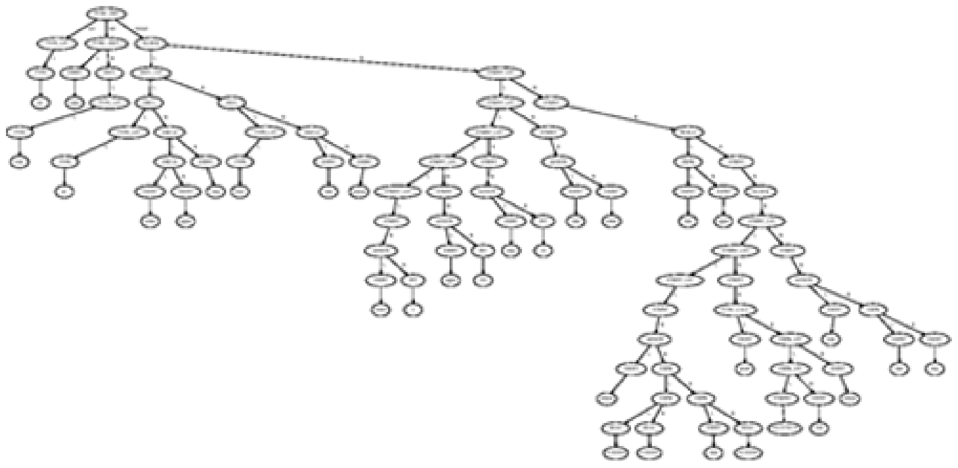
```
#include <stdio.h>

int
main(void)
{       int lower, upper, step;
        float fahr, celsius;

        lower = 0;
        upper = 300;
        step = 20;

        fahr = lower;
        while (fahr <= upper) {
                celsius = (5.0/9.0) * (fahr - 32.0);
                printf("%4.0f %6.1f\n", fahr, celsius);
                fahr = fahr + step;
        }
}
```

The program defines a function called `main`, declares five local variables, and does some standard manipulations to compute a conversion table from temperature measured in degrees Fahrenheit to the equivalent expressed in degrees Celsius. Suppose we wanted to convert this little program into a PROMELA model. The first problem we would run into is that PROMELA does not support the C data−type `float`, and has no keyword `while`. The control−structure that is used in the program, though, could easily be expressed in PROMELA. If we ignore the data−types for the moment, and pretend that they are integers, the `while` loop could be expressed like this:

```
fahr = lower;
do
:: (fahr <= upper) ->
        celsius = (5/9) * (fahr-32);
        printf("%d %d\n", fahr, celsius);
        fahr = fahr + step;
:: else -> break
od
```

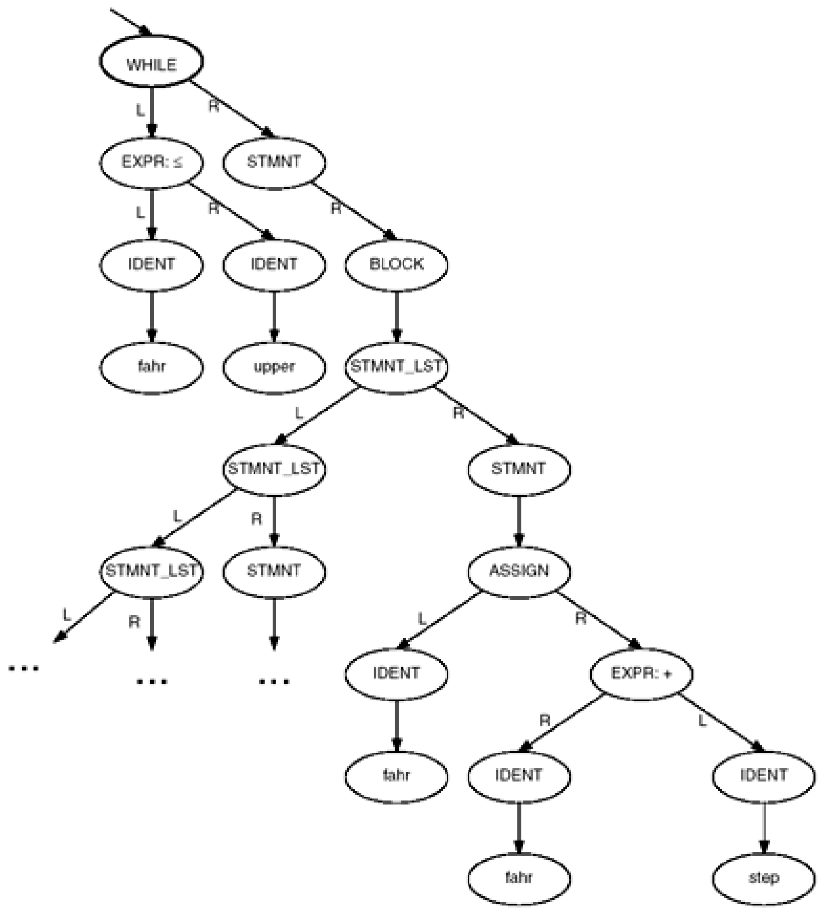
It is not hard to see almost all control−flow structures that can be defined in C can be replicated in PROMELA. (There are some exceptions, though, that we will consider shortly.) The control structure can be derived, for instance, from the standard parse−tree representation of a program that is constructed by a C compiler. Figure 10.1 shows a parse tree for the example temperature conversion program.

Figure 10.1. The Complete Parse Tree for fahr.c



The complete tree is large, even for this small program, including nodes for the main program, but also for all default global declarations that are retrieved from the included `stdio.h` file. The interesting part of the tree is on the lower right, which is reproduced in a little more detail in Figure 10.2. It contains the top of the parse–tree structure for the `while` loop.

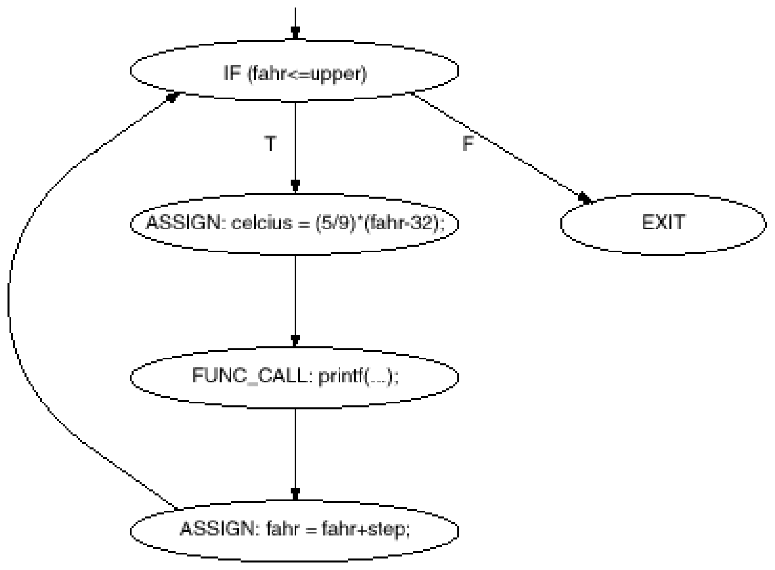
Figure 10.2. Part of Parse Tree Structure for the While Loop



The node at the top of Figure 10.2 has two successors. The left child (reached via the transition marked L defines the loop condition, and the right child defines the code fragment for the loop body as a sequence of statements that are further detailed in the subtree below it.

A parse tree is acyclic. To convert it into a control−flow graph we have to interpret the semantics of keywords such as `if`, `for`, `while`, and `goto`. The control flow graph representation shown in Figure 10.3 can be derived automatically from the parse tree structure in this way, and is already much closer to the final automaton representation that we would like to generate.

Figure 10.3. Control−Flow Graph for While Construct from Figure 10.2



So, although there can be some thorny issues that we will sidestep for now, it should be clear that most of the control flow from a C program can be reproduced in a PROMELA model without too many difficulties. But what about C declarations and basic C statements that manipulate the values of data objects that have no direct equivalent in PROMELA? To make a conversion work, we must also find a method to bridge this gap. One way to handle this would be to rely solely on abstraction methods, for instance, to map values of type `float`, which are not defined in PROMELA, to values of type `int`, or perhaps even `bool`. This may not always be possible, or convenient, though. It should, in principle, also be possible to use only minimal abstractions, and to build a verification model that behaves as closely as possible to the real application, including the use of variables of type `float`, for instance, if that is most convenient.

The key to the solution of this problem lies in the observation that SPIN already generates C code from PROMELA models when it generates the `pan.c` code for a verification run. To do this, SPIN must generate a fragment of C code for every basic PROMELA statement that can appear in the model. This mechanism gives us a good way to use C as a host language for other parts of a model specification. Rather than write the complete model in PROMELA, we can devise means to write parts of it directly in C code, in such a way that these C code fragments can be embedded directly into the verifiers that are generated by SPIN.

In SPIN version 4.0 and later, trusted fragments of C code can be embedded into a SPIN model as basic statements. For this purpose, PROMELA uses a new type of primitive statement, called `c_code`. An embedded C code statement for the manipulation of the two variables of type `float` in the example program, for instance, could be written in PROMELA as follows:

```
c_code { celsius = (5/9) * (fahr-32); };
```

Similarly, the model can contain trusted declarations for embedded data declarations of C data objects that are included into the model, and become an integral part of the state descriptor. The embedded statements and declarations are trusted, because they fall outside the PROMELA language definition and their syntax or semantics cannot be checked by the SPIN parser.

The extensions allow us to bypass the PROMELA parser to include native C data objects into the state vector of the model checker, and the matching state transformers that are written as embedded fragments of native C code. To declare the foreign data objects `fahr` and `celsius` as C data–types, we can use the declarator `c_state`, as follows:

```
c_state "float fahr" "Local main"
c_state "float celsius" "Local main"
```

The first of these two declarations states that the declaration `float fahr` is to be inserted into the `proctype` declaration called `main` as a local data object. Similarly, the second declaration introduces `celsius` as another data object of type `float`. The first argument to `c_state` remains uninterpreted. To SPIN it is merely a string that is inserted at the appropriate point into the source text of the model checker as a data declaration. If the declaration is in error, for instance, if it used the data type `floatt` instead of `float`, SPIN would not be able to detect it. The error would show up only when we compile the generated model checking code: the embedded fragments are trusted blindly by SPIN.

With these declarations, we have to modify our embedded C code fragment a little bit, to inform the C compiler that the two `float` variables are not regular C variables, but imported local variables that will appear in the state descriptor. This is done by prefixing the variable names, for instance, as follows:

```
c_code { Pmain->celsius = (5/9) * (Pmain->fahr-32); };
```

A detailed explanation of the rules for accessing local or global variables, and the extensions to PROMELA that support the use of embedded declarations and C code fragments, is given in Chapter 17. For the purpose of this chapter, it suffices to know the basic mechanism that can be exploited. Since the embedded code fragments bypass the checks that SPIN can normally make, the intent of the PROMELA extensions is primarily, perhaps exclusively, to support automated model extraction tools that can replicate trusted portions of application software in a PROMELA verification model.

The complete model for the Fahrenheit conversion program, generated automatically by the model extraction tool MODEX,[1] is shown in Figure 10.4.

[1] See Appendix D for downloading the MODEX software.

Figure 10.4 MODEX Generated SPIN model

```
$ spin -a fahr.pml
$ cc -o pan pan.c
c_state "float fahr" "Local main"
c_state "float celsius" "Local main"

active proctype main()
{   int lower;
    int upper;
    int step;

    c_code { Pmain->lower=0; };
    c_code { Pmain->upper=300; };
    c_code { Pmain->step=20; };
    c_code { Pmain->fahr=Pmain->lower; };

    do
    :: c_expr { (Pmain->fahr <= Pmain->upper) };
       c_code { Pmain->celsius =
                    ((5.0/9.0)*(Pmain->fahr-32.0)); };
       c_code { Printf("%4.0f %6.1f\n",
                    Pmain->fahr, Pmain->celsius); };
       c_code { Pmain->fahr = (Pmain->fahr+Pmain->step); };
    :: else -> break
    od
}
```

The model extractor makes sure that all declared state variables are accessed with the proper prefixes. The `Printf` function is a predefined function within SPIN that makes sure that calls to `printf` are suppressed during the depth−first search process, but enabled when an error trail is played back.

Although we have not specified any properties to be verified for this model, we can let the verifier check how many reachable system states there are.

```
$ ./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 32 byte, depth reached 70, errors: 0
      71 states, stored
       0 states, matched
      71 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)

1.573 memory usage (Mbyte)
unreached in proctype main
        (0 of 13 states)
```

The four statements in the loop are executed a total of sixteen times, once for each of the values of `fahr` between 0 and 300 inclusive (in increments of 20). That loop traversal should generate a total of 4x16 = 64 distinct system states. Then there are a total of four initialization statements for the various local variables, increasing the total to 68, plus the initial state for the system, and two terminal states: one where the process reaches the closing curly brace and terminates, and another state that is reached after the process has died and been removed from the state descriptor altogether. This gives us a sum total of 71 reachable states, which matches the number that is reported by the verifier.

There is nothing very remarkable about the run, other than the fact that the statements executed manipulate data objects and use operations that are not part of the PROMELA language. Note, for instance, that using the same principles we can analyze code that contains C pointers and arbitrary types of data structures, provided that we make sure that all data objects that contain state information are registered with the model checker so that it can be included in the state descriptors.

If we wanted to get a sample error trace from the model, it would be simple enough to add an `assert(false)` as the last statement in the model. If we repeat the verification, we now find the assertion violation after 70 steps.

Basically, what we have defined here is that the only possible execution of the model, which is the computation and printing of the conversion table, is erroneous. If we reproduce the error trail in non−verbose mode, using the executable `pan`, the computation and generation of the conversion table is reproduced precisely:

```
$ ./pan -r -n # replay error trail in non-verbose mode
   0  -17.8
  20   -6.7
  40    4.4
  60   15.6
  80   26.7
 100   37.8
 120   48.9
 140   60.0
 160   71.1
 180   82.2
 200   93.3
 220  104.4
 240  115.6
 260  126.7
 280  137.8
 300  148.9
pan: assertion violated 0 (at depth 70)
spin: trail ends after 70 steps
#processes 1:
 70:    proc 0 (main) line 28 (state 14)
                assert(0)
global vars:
local vars proc 0 (main):
        int    lower:   0
        int    upper:   300
        int    step:    20
```

If we were to use the standard guided simulation option of SPIN to reproduce the trail, the best that SPIN can do is to reproduce the text of every C code fragment, but it cannot execute it.

```
$ spin -t fahr.pml
c_code1:  { Pmain->lower=0; }
c_code2:  { Pmain->upper=300; }
c_code3:  { Pmain->step=20; }
c_code4:  { Pmain->fahr=Pmain->lower; }
c_code5:  (Pmain->fahr <= Pmain->upper)
c_code6:  {  Pmain->celsius =
                   ((5.0/9.0)*(Pmain->fahr-32.0)); }
c_code7:  {  Printf("%4.0f %6.1f\n",
                  Pmain->fahr, Pmain->celsius); }
c_code8:  {  Pmain->fahr = (Pmain->fahr+Pmain->step); }
...
spin: line  28 "fahr.pml", Error: assertion violated
spin: text of failed assertion: assert(0)
spin: trail ends after 70 steps
#processes: 1
 70:    proc 0 (main) line 30 "fahr.pml" (state 15)
1 process created
```

In order for SPIN to be able to also execute the embedded C code fragments in simulation mode, it would need to have access to a built–in interpreter for the complete C language. The alternative that has been adopted is, instead of using SPIN's existing simulation mode, to reproduce all error trails that contain embedded C code with the executable program `pan`, which of course already contains the compiled code for all embedded C code fragments.

Embedded Assertions

Because SPIN lacks a parser for the C language, it has to treat all embedded C code fragments as trusted code that is passed through to the model checker as user−defined text strings. The intent is that the C code fragments are generated by a model extraction program, but even then the possibility still exists that the code thus generated may contain subtle bugs that cannot be intercepted by the model checker either, and that could cause the program to crash without producing any useful results. A mild remedy is to allow the user, or model extractor, to annotate every `c_expr` and `c_code` statement with a precondition that, if it evaluates to true, can guarantee that the statement can be executed correctly.

The preconditions act as embedded assertions. We can write, for instance

```
c_state "int *ptr;" "Local main"
...
c_code [Pmain->ptr != NULL] { *(Pmain->ptr) = 5; };
```

to state that the integer pointer variable `ptr` must have a non−zero value for the pointer dereference operation that follows in the code fragment to be safely executable. If a case is found where `ptr` evaluates to `NULL` (i.e., the precondition evaluates to false), then an assertion violation is reported and an error trail can be generated. Without the optional precondition, the model checker would try to execute the dereference operation without checks, and an unhelpful crash of the program would result.

We can use this feature to intercept at least some very common causes of program failures: nil−pointer dereferencing, illegal memory access operations, and out of bound array indexing operations. Consider the following example. For simplicity, we will ignore standard prefixing on state variables for a moment, and illustrate the concept here with access to non−state variables only.

```
c_code {
        int *ptr;
        int x[256];
        int j;
};
...
c_code { ptr = x; };
if
:: c_expr [j >= 0 && j < 256] { x[j] != 25 } ->
        c_code [ptr >= x && ptr < &(x[256])] { *ptr = 25; }
:: else
fi
```

If the variable `j` is not initialized, more than likely it would cause an out of bound array index in the `c_expr` statement. The precondition checks for the bounds, so that such an occurrence can be intercepted as a failure to satisfy the precondition of the statement. Similarly, the correctness of indirect access to array locations via pointer `ptr` can be secured with the use of a precondition.

A Framework for Abstraction

Let us take another look at the temperature conversion example. We have noted that we can distinguish the problem of converting the control flow structure of the program cleanly from the problem of converting the actions that are performed: the basic statements. Converting the control flow structure is the easier of the two problems, although there can be some thorny issues there that we will consider more closely later.

We will now consider how we can apply user–defined abstractions systematically to the statements that appear in a program. It is important to note that the control flow aspect of a program is only of secondary importance in this regard. Once some of the basic statements in a program have been replaced with abstracted versions, it may well be that also the control flow structure of the program can be simplified. The latter is only done, though, if it does not change the meaning of the program. We will see some examples of this notion shortly. The abstractions we will consider here are applied exclusively to the basic statements that appear in a program, and to the data objects that they access.

Given an ANSI–C program, like the Fahrenheit to Celcius conversion example, the model extractor MODEX can generate a default translation of each procedure that appears in the program into a PROMELA `proctype`, using embedded C code fragments to reproduce those statements that have no equivalent in PROMELA itself. The translation can be specified as a MODEX lookup table, using a simple two column format with the source text on the left and the text for the corresponding abstraction to be used in the verification model on the right. For instance, a lookup table that describes the defaults used in MODEX (i.e., without user–defined abstraction) for the Fahrenheit program would map the nine entries:

```
(fahr<=upper)
!(fahr<=upper)
lower=0
upper=300
step=20
fahr=lower
fahr=(fahr+step)
celsius=((5/9)*(fahr-32))
printf("%4.0f %6.1f\n",fahr,celsius)
```

to the following nine results:

```
c_expr { (Pmain->fahr<=Pmain->upper) }
else
c_code { Pmain->lower=0; }
c_code { Pmain->upper=300; }
c_code { Pmain->step=20; }
c_code { Pmain->fahr=Pmain->lower; }
c_code { Pmain->fahr=(Pmain->fahr+Pmain->step); }
c_code { Pmain->celsius=((5.0/9.0)*(Pmain->fahr-32.0)); }
c_code { Printf("%4.0f %6.1f\n", \
                Pmain->fahr, Pmain->celsius); }
```

The first entry in the table is the conditional from the `while` statement in the C version of the code. The negation of that statement, corresponding to the exit condition from the loop, appears as a separate entry in the table, to make it possible to define a different translation for it. It would, for instance, be possible to replace the target code for both the loop condition and the loop exit condition with the boolean value true to create a non−deterministic control structure in the model. (In this case, of course, this would not be helpful.) The default conversion for the remaining statements simply embeds the original code within a PROMELA `c_code` statement, and prefixes every variable reference as required for local variables that appear in the model.

The default treatment in MODEX for variable declarations is to embed them into the model with the help of PROMELA `c_state` statements. All the defaults, though, can be overridden, for instance, to enforce abstraction.

As a small illustration of the mechanism that can be invoked here to enforce abstraction, let us assume we wanted to generate a pure PROMELA model from the C code, using integer variables and integer computations to approximate the results of the Fahrenheit to Celsius conversions. The mechanism for doing so that is supported in MODEX is the lookup table.

First, we replace the default embedded data declarations with an integer version. We do so by including the following five lines in the user−defined version of the lookup table that will be used to override MODEX's defaults.

```
Declare int fahr    main
Declare int celsius main
Declare int upper   main
Declare int lower   main
Declare int step    main
```

The keyword `Declare` is followed by three tab−separated fields. The first field specifies the name of a data type, the second the name of a variable, and the third specifies the scope of the variable. In this case, the scope is local and indicated by the name of the target `proctype`, "main." If global, the last field would contain the MODEX keyword `Global`.

Now that we have integer data, the C statements from the original program can be interpreted directly as PROMELA statements, rather than as embedded C code fragments. We can specify this by using the MODEX keyword `keep` as the target for the corresponding entries in the lookup table. This can be done for seven of the nine entries, as follows:

```
(fahr<=upper)       keep
!(fahr<=upper)      else
lower=0             keep
upper=300           keep
step=20             keep
fahr=lower          keep
fahr=(fahr+step)    keep
```

The computation of the conversion values for the variable `celcius` will have to be done a little more carefully. Note that the sub−expression `(5/9)` would evaluate to zero in integer arithmetic, resulting in all computed values being approximated as zero. With the following translations, we avoid this by making sure that the division by nine happens last, not first. Next, the print statement needs a slightly different format to print integer instead of floating point values. Because the integer data types are recognized by PROMELA directly, these statements need not be embedded in C code fragments, but can also be generated directly as PROMELA code, as follows:

```
celsius=((5/9)*(fahr-32)) celsius = ((fahr-32)*5)/9
printf(...                printf("%d %d\n",fahr,celsius)
```

The model extractor checks for an entry in user−supplied lookup tables, based on a textual match on entries in the left−hand side column, and uses the user−defined translation when given, or else the default translation. Simple abbreviations, such as the ellipses (the three dots at the end of the print statement) in the second entry above, are also supported. In this case every statement that starts with the character string "printf(" is matched and translated to the fixed target translation on right−hand side of this table entry.

From the new explicit lookup table, the following model can now be generated mechanically by the MODEX tool:

```
active proctype main()
{   int step;
    int lower;
    int upper;
    int celsius;
    int fahr;

    lower = 0;
    upper = 300;
    step = 20;
    fahr = lower;

    do
    :: (fahr<=upper);
        celsius = ((fahr-32)*5)/9;
        printf("%d %d\n", fahr, celsius);
        fahr=(fahr+step)
    :: else -> break
    od
}
```

The new model uses no embedded C code, so we can use it to generate the approximate integer values for the temperature conversions with a standard SPIN simulation run, as follows:

```
$ spin fahr2.pml
      0  -17
      20  -6
      40  4
      60  15
      80  26
      100  37
      120  48
      140  60
      160  71
      180  82
      200  93
      220  104
      240  115
      260  126
      280  137
      300  148
1 process created
```

A model extractor, used in combination with user−defined mapping tables, can generate almost any transformation of a given program, being restricted only to the fixed control flow structure that is specified in the original program. This can be considered to be both a strength and a weakness of the method. It clearly gives the model extractor all the power it needs to support arbitrary abstraction methods in applications of software verification. But the method can also easily be abused to generate meaningless or nonsensical abstractions. By itself, the model extractor is only a tool that can be used to define abstractions: it cannot determine what proper abstractions are or how they should be derived. For that we need different mechanisms that we discuss next.

Sound and Complete Abstraction

One critical issue that we have not yet discussed is how we can define abstractions and how we can make sure that they are meaningful. The best abstraction to be used in a given application will depend on the types of correctness properties that we are interested in proving. The properties alone determine which aspects of the application are relevant to the verification attempt, and which are not.

Let P be the original program, and let L be a logical property we want to prove about P. Let further α be an abstraction function.

Let us begin by considering the case where abstraction α is defined as a MODEX lookup table. We will denote by $\alpha(P)$ the abstract model that is derived by MODEX from program P for abstraction α. That is, $\alpha(P)$ is the model in which every basic statement in P is replaced with its target from the MODEX lookup table, but with the same control flow structure which is reproduced in the syntax of PROMELA.

Under a given abstraction α, the original property L will generally need to be modified to be usable as a property of the abstract model $\alpha(P)$. Property L may, for instance, refer to program locations in P or refer to data objects that were deleted or renamed by α. L may also refer to data objects for which the type was changed, for instance, from integer to boolean. We will denote this abstraction of L by $\alpha(L)$.

The inverse of abstraction α can be called a concretization, which we will denote by $\bar{\alpha}$. A concretization can be used to translate, or lift, abstract statements from the model back into the concrete domain of the original program. In general, because of the nature of abstraction, any given abstract statement can map to one or more possible concrete statements. This means that for a given statements s, $\alpha(s)$ defines a single abstract statement, but concretization $\bar{\alpha}(\alpha(s))$ defines a set of possible concrete statements, such that

$$\forall s,\ s \in \bar{\alpha}(\alpha(s)).$$

Similarly, for every abstract execution sequence ϕ in model α (P) we can derive a set of concrete execution sequences, denoted by $\bar{\alpha}(\phi)$, in the original program. Given that an abstraction will almost always remove some information from a program, it is not necessarily the case that for every feasible execution ϕ of the abstract program there also exists a corresponding feasible execution within $\bar{\alpha}(\phi)$ of the concrete program. This brings us to the definition of two useful types of requirements that we can impose on abstractions: logical soundness and completeness.

Definition 10.1 (Logical Soundness)

> Abstraction α is logically sound with respect to program P and property L if for any concrete execution ϕ of P that violates L there exists a corresponding abstract execution of α (P) in α (ϕ) that violates α (L).

Informally, this means that an abstraction is logically sound if it excludes the possibility of false positives. The correctness of the model always implies the correctness of the program.

Definition 10.2 (Logical Completeness)

> Abstraction α is logically complete with respect to program P and property L if, for any abstract execution ϕ of α(P) that violates α(L), there exists a corresponding concrete execution of P in $\bar{\alpha}(\phi)$ that violates L.

Informally, this means that an abstraction is logically complete if it excludes the possibility of false negatives. The incorrectness of the model always implies the incorrectness of the program.

Selective Data Hiding

An example of a fairly conservative and simple abstraction method that guarantees logical soundness and completeness with respect to any property that can be defined in LTL is selective data hiding. To use this method we must be able to identify a set of data objects that is provably irrelevant to the correctness properties that we are interested in proving about a model, and that can therefore be removed from the model, together with all associated operations.

This abstraction method can be automated by applying a fairly simple version of a program slicing algorithm. One such algorithm is built into SPIN. This algorithm computes, based on the given properties, which statements can be omitted from the model without affecting the soundness and completeness of the verification of those properties. The algorithm works as follows. First, a set of slice criteria is constructed, initially including only those data objects that are referred to explicitly in one or more correctness properties (e.g., in basic assertions or in an LTL formula). Through data and control dependency analysis, the algorithm then determines on which larger set of data objects the slice criteria depend for their values. All data objects that are independent of the slice criteria, and not contained in the set of slice criteria themselves, can then be considered irrelevant to the verification and can be removed from the model, together with all associated operations.

The data hiding operation can be implemented trivially with the help of a MODEX lookup table, by arranging for all irrelevant data manipulations to be mapped to either `true` (for condition statements) or to `skip` (for other statements). Note that under this transformation the basic control flow structure of the model is still retained, which means that no execution cycles can be added to or removed from the model, which is important to the preservation of liveness properties.

Although this method can be shown to preserve both logical soundness and logical completeness of the correctness properties that are used in deriving the abstraction, it does not necessarily have these desirable properties for some other types of correctness requirements that cannot be expressed in assertions or in LTL formulae. An example of such a property is absence of deadlock. Note that the introduction of extra behavior in a model can result in the disappearance of system deadlocks.

Example

To illustrate the use of selective data hiding, consider the model of a word count program shown in Figure 10.5. The program receives characters, encoded as integers, over the channel `stdin`, and counts the number of newlines, characters, and white–space separated words, up to an end–of–file marker which is encoded as the number `-1`.

Figure 10.5 Word Count Model

```
 1 chan STDIN;
 2 int c, nl, nw, nc;
 3
 4 init {
 5     bool inword = false;
 6
 7     do
 8     :: STDIN?c ->
 9         if
10        :: c == -1 ->   break      /* EOF */
11        :: c == '\n' -> nc++; nl++
12        :: else ->      nc++
13        fi;
14        if
15        :: c == ' ' || c == '\t' || c == '\n' ->
16            inword = false
17        :: else ->
18            if
19            :: !inword ->
20                nw++; inword = true
21            :: else /* do nothing */
22            fi
23        fi
24    od;
25    assert(nc >= nl);
26    printf("%d\t%d\t%d\n", nl, nw, nc)
27 }
```

The assertion checks that at the end of each execution the number of characters counted must always be larger than or equal to the number of newlines. We want to find a simpler version of the model that would allow us to check this specific property more efficiently. Variables `nc` and `nl` are clearly relevant to this verification, since they appear explicitly in the assertions. So clearly the statements in which these variables appear cannot be removed from the model. But which other statements can safely be removed?

When we invoke SPIN's built–in slicing algorithm it tells us:

```
$ spin -A wc.pml
spin: redundant in proctype :init: (for given property):
 line 19 ... [(!(inword))]
 line 20 ... [nw = (nw+1)]
 line 20 ... [inword = true]
 line 15 ... [((((c==' ')||(c=='\t'))||(c=='\n)))]
 line 16 ... [inword = false]
```

```
spin: redundant vars (for given property):
   int     nw      0       <:global:> <variable>
   bit     inword  0       <:init:>   <variable>
spin: consider using predicate abstraction to replace:
   int     c       0       <:global:> <variable>
```

From this output we can conclude that the program fragment between lines 14 to 23 is irrelevant, and similarly variables `nw` and `inword`. SPIN also suggests that the declaration of variable `c` could be improved: it is declared as an integer variable, but within the model only three or four value ranges of this variable are really relevant. We could do better by using four symbolic values for those ranges, and declaring `c` as an `mtype` variable. This suggestion, though, is independent of the data hiding abstraction that we can now apply. If we preserve the entire control−flow structure of the original, the abstraction based on data hiding could now be constructed (manually) as shown in Figure 10.6.

Figure 10.6 Abstracted Word Count Model

```
 1 chan STDIN;
 2 int c, nl, nc;
 3
 4 init {
 5
 6
 7     do
 8     :: STDIN?c ->
 9         if
10        :: c == -1 ->   break    /* EOF */
11        :: c == '\n' -> nc++; nl++
12        :: else ->      nc++
13        fi;
14        if
15        :: true ->
16            skip
17        :: true ->
18            if
19            :: true ->
20                skip; skip
21            :: true
22            fi
23        fi
24    od;
25    assert(nc >= nl);
26    printf("%d\t%d\n", nl, nc)
27 }
```

We can simplify this model without adding or deleting any control−flow cycles, by collapsing sequences of consecutive `true` and `skip` statements. It is important that we do not add or omit cycles from the model in simplifications of this type, because this can directly affect the proof of liveness properties. A cycle, for instance, could only safely be added to or omitted from the model if we separately prove that the cycle always terminates within a finite number of traversals. The simplified model looks as shown in Figure 10.7.

Figure 10.7 Simplified Model

```
 1 chan STDIN;
 2 int c, nl, nc;
 3
 4 init {

    do
    :: STDIN?c ->
        if
        :: c == -1 ->  break    /* EOF */
       :: c == '\n' -> nc++; nl++
       :: else ->      nc++
       fi
   od;
   assert(nc >= nl);
   printf("%d\t%d\n", nl, nc)
}
```

Because the abstraction we have applied is sound and complete, any possible execution that would lead to an assertion violation in this simplified model implies immediately that a similar violating execution must exist in the original concrete model. Perhaps surprisingly, there is indeed such an execution. An assertion violation can occur when the value of variable `nc` wraps around its maximal value of $2^{32} - 1$ before the value of variable `nl` does, which can happen for a sufficiently large number of input characters.

Bolder Abstractions

Logically sound and complete abstractions are not always sufficient to render large verification problems tractable. In those cases, one has to resort to abstraction strategies that lack either one or both of these qualities. These abstraction strategies are often based on human judgement of what the most interesting, or most suspect, system behaviors might be, and can therefore usually not be automated. Using these strategies also puts a greater burden on the user to rule out the possibility of false negatives or positives with additional, and often manual, analyses.

We will discuss two examples of abstraction methods in this class:

- Selective restriction
- Data type abstraction

The first method is neither sound nor complete. The second method is complete, but not necessarily sound.

Selective restriction is commonly used in applications of model checking tools to limit the scope of a verification to a subset of the original problem. We can do so, for instance, by limiting the maximum capacity of message buffers below what would be needed for a full verification, or by limiting the maximum number of active processes. This method is indubitably useful in an exploratory phase of a verification, to study problem variants with an often significantly lower computational complexity than the full problem that is to be solved. This type of abstraction, though, is to be used with care since it can introduce both false negatives and false positives into the verification process. An example of this type of selective restriction is, for instance, the verification model of a leader election algorithm that can be found in the standard SPIN distribution. To make finite state model checking possible, the number of processes that participate in the leader election procedure must be fixed, although clearly the full problem would require us to perform a verification for every conceivable number of processes. As another example from the same set of verification models in the SPIN distribution, consider the PROMELA model of a file transfer protocol named `pftp`. For exhaustive verification, each channel size should be set to a bound larger than the largest number of messages that can ever be stored in the channel. By lowering the bound, partial verifications can be done at a lower cost, though without any guarantee of soundness or completeness.

Data type abstraction aims to reduce the value range of selected data objects. An example of this type of abstraction could be to reduce a variable of type integer to an enumeration variable with just three values. The three values can then be used to represent three ranges of values in the integer domain (e.g., negative, zero, and positive). The change can be justified if the correctness properties of a model do not depend on detailed values, but only on the chosen value ranges.

A data type abstraction applied to one variable will generally also affect other variables within the same model. The type of, and operations on, all variables that depend on the modified variables, either directly or indirectly, may have to be adjusted. A data and control dependency analysis can again serve to identify the set of data objects that is affected in this operation, and can be used to deduce the required changes.

Denote by V the set of concrete values of an object and by A the associated set of abstract values under type abstraction α. To guarantee logical completeness, a data type abstraction must satisfy the following relation, known as the Galois connection:[2]

[2] Note that concretization function $\bar{\alpha}$ defines a set.

$$\forall v \in V,\ v \in \bar{\alpha}(\alpha(v)) \ \wedge\ \forall w \in A, \forall x \in \bar{\alpha}(w),\ w \equiv \alpha(x).$$

Consider, for instance, the property

$$\Box((x < 0) \rightarrow \Diamond(x \geq 0))$$

Table 10.1. Example of Type Abstraction

Statement	Abstraction
(x > 5)	(!neg_x)
x = 0	neg_x = false
x + +	if : :neg_x –> if /* non–deterministic choice */ : :neg_x = true : :neg_x = false fi : :else –> skip fi

The property depends only on the sign of variable x, but not on its absolute value. With a data type abstraction we can try to replace every occurrence of x in the model with a new variable that captures only its sign, and not its value. For example, if the model contains assignment and condition statements such as

```
(x > 5); x = 0; x++;
```

we can replace all occurrences of `x` in these statements with a new boolean variable `neg_x`. The property then becomes:

$$\Box((neg\_x) \rightarrow \Diamond(\neg neg\_x))$$

The assignments and conditions are now mapped as shown in Table 10.1. Under this abstraction, precise information about the value of the integer variable `x` can be replaced with non−deterministic guesses about the possible new values of the boolean variable `neg_x`. Note, for instance, that when `neg_x` is true, and the value of `x` is incremented, the new value of `x` could be either positive or remain negative. This is reflected in a non−deterministic choice in the assignment of either true or false to `neg_x`. If, however, `x` is known to be non−negative, it will remain so after the increment, and the value of `neg_x` remains false. The condition (x > 5) can clearly only be true when x is non−negative, but beyond that we cannot guess. The Galois connection holds for these abstractions.

Dealing With False Negatives

The occurrence of false negatives as a result of a logically unsound abstraction is not as harmful as it might at first sight seem to be. By analyzing concretizations of counterexample executions it can often quickly be determined what piece of information was lost in the abstraction that permitted the generation of the false negative. The counterexample in effect proves to the user that the information that was assumed irrelevant is in fact relevant. Guided by the counterexample, the abstraction can now be refined to eliminate the false negatives one by one, until either valid counterexamples are generated, or a proof of correctness is obtained. It is generally much harder to accurately analyze a false positive result of a model checker, for instance, if selective restrictions were applied: caveat emptor.

Thorny Issues With Embedded C Code

The support in SPIN for embedded C code significantly extends the range of applications that can be verified, but it is also fraught with danger. Like a good set of knifes, this extension can be powerful when used well, but also disastrous when used badly. As one simple example, it is readily possible to divide a floating pointer number by zero in an embedded C code fragment, or to dereference a nil−pointer. Since SPIN deliberately does not look inside embedded C code fragments, it cannot offer any help in diagnosing problems that are caused in this way. To SPIN, embedded C code fragments are trusted pieces of foreign code that define state transitions and become part of the model checker. In effect, the PROMELA extensions allow the user to redefine parts of the PROMELA language. It is ultimately the user's responsibility to make sure that these language extensions make sense.

We can place our trust in a model extraction tool such as MODEX to generate embedded C code that is (automatically) guarded with embedded assertions, but there are certainly still cases where also that protection can prove to be insufficient. Note, for instance, that it is readily possible within embedded C code fragments to unwittingly modify relevant state information that is beyond the purview of the model checker. External state information could, for instance, be read from external files, the contents of which can clearly not be tracked by the model checker. Hidden data could also be created and manipulated with calls to a memory allocator, or even by directly communicating with external processes through real network connections.

In cases where the model checker is set to work on a model with relevant state information that is not represented in the internal state vectors, false negatives and positives become possible. False negatives can again more easily be dealt with than false positives. Inspecting a counterexample can usually quickly reveal what state information was missed. False positives are also here the more dangerous flaw of an extended SPIN model. It will often be possible to incorporate hidden state information explicitly in a PROMELA model. Calls to a memory allocator such as `malloc`, for instance, can be replaced with calls to a specially constructed PROMELA model of the allocator, with all state information explicitly represented in the model. Similarly, information read from files or from network connections can be replaced with information retrieved from internal PROMELA processes, again making sure that all relevant state information becomes an explicit part of the verification model.

There can also be delicate issues in the framework we have sketched for model extraction from ANSI C source code. While it is true that most of the control−flow structures of a C program can be reproduced faithfully in a PROMELA verification model, there are some notable exceptions. The invocation of a function via a function pointer in C, for instance, could be preserved within an embedded C code fragment in PROMELA, but it would be very hard to intercept such calls and turn them into PROMELA process instantiations. In this case, too, we have to accept that there are limits to our verification technology. We can occasionally move the limits, but as these examples show, we cannot altogether remove them. Although much of the verification process can be automated, some barriers remain that can only be scaled with the help of human skill and judgement.

The Model Extraction Process

Using MODEX and SPIN, we can now approach software verification problems with the following general methodology.

- The first step is to decide what precisely the critical correctness properties of the application are. The properties of special interest are those that are non−computational in nature. Model checkers are especially strong in verifying concurrency−related problems.
- Next, we identify those parts of the system that contribute most critically to the behavior of primary interest. The effort here is again to find the smallest sufficient portion of the system to prove that the properties of interest are satisfied.
- The first two decisions can now guide us in the construction of an abstraction table that suppresses irrelevant detail and highlights the important aspects of the system. In many cases, no special user decisions will be needed and we can rely on the model extractor to use appropriate default translations. A model extractor like MODEX can also guide the user in the construction of the abstraction tables to make sure that no cases are missed.
- Verification can now begin. Inevitably, there will be things that need adjusting: mistranslations, or missed cases. The attempt itself to perform verification helps to identify these cases. If the model with its embedded C code is incomplete, for instance, either SPIN or the C compiler will reject it, in both cases with detailed explanations of the underlying reasons.
- Once we have a working verifier, we can start seeing counterexamples. Again, there will be a learning cycle, where false negatives will need to be weeded out and the abstraction tables adjusted based on everything that is learned in this phase.
- Eventually, either we find a valid counterexample, or a clean bill of health from the model checker. A valid counterexample is in this sense the desired outcome. A clean bill of health (i.e., the absence of counterexamples) should be treated with suspicion. Were the properties correctly formalized? Are they not vacuously true? Did the abstraction preserve enough information to prove or disprove them? A few experiments with properties that are known to be valid or invalid can be very illuminating in this phase, and can help build confidence in the accuracy of the results.

The process as it is sketched here still seems far removed from our ideal of developing a fully automated framework for thoroughly checking large and complex software packages. At the same time, though, it is encouraging that the approach we have described here offers an unprecedented level of thoroughness in software testing. As yet, there is no other method known that can verify distributed systems software at the level of accuracy that model extraction tools allow. The combination of model extraction and model checking enables us to eliminate the two main issues that hamper traditional testing by providing a reliable mechanism for both controllability and observability of all the essential elements of a distributed system.

To close the gap between the current user−driven methodology and a fully automated framework still requires some work to be done. Given the importance of this problem, and the number of people that are focusing on it today, we can be fairly confident that this gap can successfully be closed in the not too distant future.

The Halting Problem Revisited

In the days when C was still just a letter in the alphabet, and C++ a typo, it was already well established that it would be folly to search for a computer program that could decide mechanically if some arbitrary other computer program had an arbitrary given property. Turing's formal proof for the unsolvability of the halting problem was illustrated in 1965 by Strachey with the following basic argument. Suppose we had a program `mc(P)` that could decide in bounded time if some other program `P` would eventually terminate (i.e., halt); we could then write a new program `nmc(P)` that again inspects some other program `P` (e.g., after reading it), and uses `mc(P)` in the background. We now write `nmc(P)` in such a way that it terminates if `P` does not terminate, and vice–versa. All is well until we decide to run `nmc(nmc)` to see if `nmc` itself is guaranteed to terminate. After all, `nmc` is just another program, so this would be fair game.

Strachey's construction is similar to Betrand Russell's famous class paradox, which makes it somewhat dubious to use as a basis for a logical argument. In fairness, Strachey's construction does not really prove the unsolvability of the halting problem, but it does prove that it is in general impossible to write a program that can establish arbitrary properties of itself. This, of course, does not mean that no programs could be written that can establish specific properties of themselves. A word–count program, for instance, can trivially determine its own length, and a C–compiler can confirm its own syntactic correctness, and even recompile itself.

Clearly we cannot write a program that could ever prove its own logical correctness. Note, for instance, that such a program could easily report its own correctness erroneously.

So far, this debate about arbitrary programs and arbitrary properties is purely academic, as it has been for pretty much the last 100 years. Curiously, though, the extensions to SPIN with embedded C code have made it possible to actually write a very real version of Strachey's construction. First, we write a little script that returns true if SPIN determines that a given model has at least one invalid end state, and false if it does not.

```
#!/bin/sh
### filename: halts

echo -n "testing $1: "

spin -a $1                   # generate model
cc -DSAFETY -o pan pan.c     # compile it
./pan | grep "errors: 0"     # run it and grep stats
if $?                        # test exit status of grep
then
     echo "halts"
else
     echo "does not halt"
fi
```

We can try this on some little examples to make sure that it does the right thing. For instance, if we apply our script to the hello world example from Chapter 2, we get reassuringly:

```
$ ./halts hello.pml
halts
```

If we add a blocking statement to this model

```
active proctype main()
{
        printf("hello world0);
        false /* block the execution */
}
```

and run the script again, we get

```
$ ./halts hello_blocking.pml
does not halt
```

We can now try to invoke this program in a `c_expr` statement with SPIN version 4, in the devious manner that was envisioned by Strachey.

```
init { /* filename: strachey */
       do
       :: c_expr { system("halts strachey") } /* loop */
       :: else -> break
       od;
       false /* block the execution */
}
```

What would happen if we now execute

```
$ ./halts strachey
.....
```

The `halts` program ends up going into an infinite descent. Each time the verifier gets to the point where it needs to establish the executability of the `c_expr`, it needs to invoke the `halts` script once more and restart itself. The behavior of the model is defined in terms of itself, which puts it outside the scope of systems that can be verified with finitary methods. It would be hard to maintain that the infinite recursion is caused by the method that we used to implement the `halts` script. Note that if the `halts` script only needed to read the program before rendering its verdict, and did not need to execute it, the same infinite descent would still occur.

Curiously, the fact that the `halts` program loops on some inputs could in principle be detected by a higher−level program. But, as soon as we extend our framework again to give that new program the capability to reason about itself, we inevitably recreate the problem.

The example aptly illustrates that by allowing embedded C code inside SPIN models the modeling language becomes Turing complete, and we lose formal decidability. As yet another corollary of Kurt Gödel's famous incompleteness theorem: any system that is expressive enough to describe itself cannot be powerful enough to prove every true property within its domain.

Bibliographic Notes

Alan Turing's seminal paper on computable and uncomputable functions appeared in 1936, Turing [1936]. Christopher Strachey's elegant construction to illustrate the unsolvability of the halting problem first appeared in Strachey [1965]. In this very short note, Strachey refers to an informal conversation he had with Turing many years earlier about a different version of the proof.

The class paradox was first described in Russell [1903]. A fascinating glimpse of the correspondence between Russell and Frege about the discovery of this paradox in 1902 can be found in Heijenoort [2000]. Gödel's original paper is Gödel [1931].

General references to abstraction techniques can be found in the Bibliographic Notes to Chapter 5. Here we focus more on abstraction techniques that are used specifically for application in model extraction and model checking.

Data type abstractions can in some cases be computed mechanically for restricted types of statements and conditions, for instance, when operations are restricted to Pressburger arithmetic. In these cases, one can use a mechanized decision procedure for the necessary computations (see, for instance, Levitt [1998]). A description of an automated tool for computing program or model slices based on selective data hiding can be found in Dams, Hesse, and Holzmann [2002].

In the literature, logical completeness is often defined only for abstraction methods, not for abstract models as we have done in this chapter. For a more standard discussion of soundness and completeness, see, for instance, Kesten, Pnueli, and Vardi [2001].

An excellent overview of general program slicing techniques can be found in Tip [1995].

The first attempts to extract verification models mechanically from implementation level code targeted the conversion of Java source code into PROMELA models. Among the first to pursue this, starting in late 1997, were Klaus Havelund from the NASA Ames Research Center, and Matt Dwyer and John Hatcliff from Kansas State University, as described in Havelund and Pressburger [2000], Brat, Havelund, Park, and Visser [2000], and Corbett, Dwyer, Hatcliff, et al. [2000].

The work at Ames led to the Pathfinder verification tool. The first version of this tool, written by Havelund, converted Java to PROMELA, using a one−to−one mapping. The current version, called Java Pathfinder−2, was written at Ames by Willem Visser as a stand−alone model checker, that uses an instrumented virtual machine to perform the verification.

The work at Kansas State on the Bandera tool suite targets the conversion from Java to PROMELA, and it includes direct support for data type abstraction and program slicing. The Bandera tool supports a number of other model checking tools as alternative verifiers, in addition to the SPIN tool.

The work at Bell Labs, starting in 1998, was the first attempt to convert ANSI C source code into abstract PROMELA verification models. It is described in, for instance, Holzmann and Smith [1999,2000,2002], and Holzmann [2000]. The code for the model extractor MODEX and for the extended version of SPIN is generally available today (see Appendix D). A more detailed description of the use of this tool, and the recommended way to develop a test harness definition, can be found in the online user guide for MODEX.

A related attempt to systematically extract abstract verification models from sequential C source code is pursued at Microsoft Research in the SLAM project, see Ball, Majumdar, Millstein, and Rajamani [2001]. The Bebop tool being developed in this project is based on the systematic application of predicate and data type abstractions.

The problem of detecting whether or not a property is vacuously satisfied was dealt with in the FeaVer system by implementing an automated check on the number of states that was reached in the `never` claim automata, see Holzmann and Smith [2000,2002]. A high fraction of unreachable states was found to correlate well with vacuously true properties. A more thorough method of vacuity checking is described in Kupferman and Vardi [1999].