Advanced: Deductive verification

An alternative approach to verification is *deduction*. A formal semantics is defined for program constructs and then a formal logic with axioms and inference rules is used to deduce that a program satisfies correctness specifications, expressed, for example, as assertions. The advantage of deductive verification is that it is not limited by the size of the state space because the deduction is done on symbolic formulas; the disadvantage is that it is less amenable to automation and requires mathematical ingenuity.

A deductive verification of the program in Listing 2.2 is given in Section B.4 of *PCDP*; it was partially automated using the verification capabilities of the SPARK system [3].

For an overview of deductive verification, see Chapter 9 of *MLCS*; an advanced textbook is [1].

2.2 Verifying a program in SPIN

Consider the program in Listing 2.3 that has an error in the second alternative (line 5). When `a` equals `b` a random simulation is just as likely to take the first alternative of the **`if`**-statement as the second. In fact, even if we run the simulation repeatedly, it is possible – although unlikely – that the same alternative will always be taken. In other words, no amount of simulation can ever verify that the postcondition is true, because it may become true if one alternative is taken, while it is falsified in the other alternative.

Listing 2.3. Maximum with an error

```
1 active proctype P() {
2         int a = 5, b = 5, max;
3         if
4         :: a >= b -> max = a;
5         :: b >= a -> max = b+1;
6         fi;
7         assert (a >= b -> max == a : max == b)
8 }
```

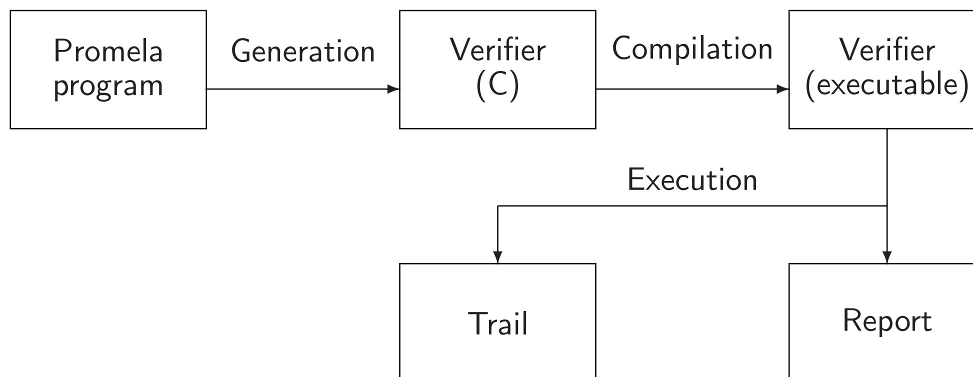
The only way to verify that a program is correct is to systematically check that the correctness specifications hold in *all possible computations*, and that is what model checkers like SPIN are designed to do.

In a deterministic program (with no input), there is only one possible computation, so a single random simulation will suffice to demonstrate the correctness of a program. For a concurrent or nondeterministic program, checking all possible computations involves executing the program and backtracking over each choice of the next statement to execute. One of the ways that SPIN achieves efficiency is by generating an optimized program called a *verifier* for each PROMELA model. Verification in SPIN is a three-step process (Figure 2.1):

- Generate the verifier from the PROMELA source code.
  The verifier is a program written in C.
- Compile the verifier using a C compiler.
- Execute the verifier. The result of the execution of the verifier is a report that *all* computations are correct or else that *some* computation contains an error. (The *Trail* shown in the figure is explained in the next section.)

Fortunately, there is no need to examine the C source code of the verifier; you simply perform these three steps within a script, or use JSPIN, which invokes SPIN, the C compiler and the compiled verifier.

Fig. 2.1. The architecture of SPIN



jSpin

Select `Verify`. The commands that are executed are listed in the message pane. The report of the verifier is displayed in the right pane.

Command line

Run SPIN with the argument `-a` to generate the verifier source code:

```
spin -a max.pml
```

Check your directory; you should find files `pan.*` including `pan.c`, which contains the source code of the main program. (The file name pan is historical and is derived from *p*rotocol *an*alyzer.) The next step is to compile this file; for the `gcc` compiler the command is:

```
gcc -o pan pan.c
```

Finally, run the verifier:

```
pan
```

You may need to enter this command as `./pan` or `.\pan`.

Verify the program in Listing 1.6 for the maximum of two numbers; you should get `errors = 0`. (For now, you can ignore the rest of the output.)

Next, verify the program in Listing 2.3 that contains an error; the report will be:

```
pan: assertion violated
   ( ((a>=b)) ? ((max==a)) : ((max==b)) ) (at depth 0)
pan: wrote max1.pml.trail
(Spin Version 4.2.8 -- 6 January 2007)
Warning: Search not completed
   ...
State-vector 24 byte, depth reached 2, errors: 1
   ...
```

SPIN does not bother to search the entire state space; instead, it stops as soon as one assertion is violated because the existence of one counterexample is usually sufficient to locate an error in the program or the correctness specifications.

Advanced: Continuing past the first error

The argument `-e` to `pan` causes trails for all errors to be created.
The argument `-cN` causes the verifier to stop at the *N*th error rather than the first, while the argument `-c0` requests the verifier to ignore all errors and not to generate a trail file.

2.2.1 Guided simulation

You may hope that your first attempt at verifying a model will succeed; however, this is unrealistically optimistic! Almost invariably it takes a long time to understand the interactions among components of the model, and between the model and its correctness specifications, in order to achieve a successful verification. Thus, a primary task of a model checker is to assist the systems engineer in understanding why a verification has failed.

SPIN supports the analysis of failed verifications by maintaining internal data structures during its search of the state space; these are used to reconstruct a computation that leads to an error. The data required for reconstructing a computation are written into a file called a *trail*. (The name of the file is the same as that of the PROMELA source code file with the additional extension `.trail`.) The trail file is not intended to be read; rather, it is used to reconstruct a computation by running SPIN in *guided simulation mode*.

---

jSpin

After running a verification that has reported errors, select `Trail` .

Command line

After running a verification that has reported errors, run SPIN again with the `-t` argument:

```
spin -t max.pml
```

An examination of the guided simulation for the program in Listing 2.3 will show that the bad computation actually occurs when the alternative with the mistake (line 5) is executed:

```
Starting P with pid 0
  0:    proc  - (:root:) creates proc  0 (P)
0 P     3   b>=a
0 P     5   max = (b+1)
```

As a check of your understanding of assertions, write the postcondition for the program in Listing 1.7 that computes the greatest common denominator of two integer numbers; verify that the program is correct.

2.2.2 Displaying a computation

When examining a computation produced by a random or guided simulation, we need more than the output that results from the print statements.