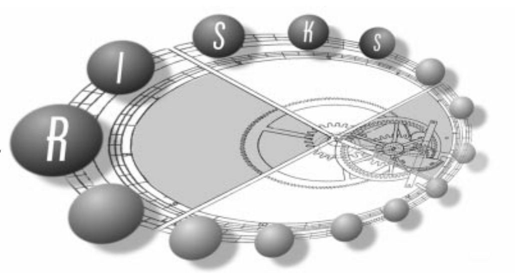

Fred B. Schneider

On Concurrent Programming

Concurrent programs are notoriously difficult to get right. This is as true today as it was 30 years ago. But 30 years ago, concurrent programs would be found only in the bowels of operating systems, and these were built by specialists. Today, concurrent programs are everywhere and are being built by relatively inexperienced programmers:

- *All sorts of application programmers write concurrent programs*. Freezing up a PC when you pull down a menu or click on an icon is likely to be caused by a concurrent-programming bug in the application.
- *Knowledge of operating system routines is no longer required to write a concurrent program*. Java threads enable programmers to write concurrent programs, whether for spiffy animation on Web pages or for applications that manage multiple activities.

A concurrent program consists of a collection of sequential processes whose execution is interleaved; the interleaving is the result of choices made by a scheduler and is not under programmer control. Lots of execution interleavings are possible, making testing of all but trivial concurrent programs infeasible.

To make matters worse, functional specifications for concurrent programs often concern intermediate steps of the computation. For example, consider a word-processing program with two processes: one that formats pages and passes them through a queue to the second process, which controls a printer. The functional specification might stipulate that the page-formatter process never deposit a page image into a queue slot that is full and that the printer-control process never retrieve the contents of an empty or partially filled queue slot.

If contemplating the individual execution interleavings of a concurrent program is infeasible, then we must seek methods that allow all executions to be analyzed together. We do have on hand a succinct description of the entire set of executions: the program text itself. Thus, analysis methods that work directly on the program text (rather than on the executions it encodes) have the potential to circumvent problems that limit the effectiveness of testing.

For example, here is a rule for showing that some "bad thing" doesn't happen during execution:

Identify a relation between the program variables that is true initially and is left true by each action of the program. Show that this relation implies the "bad thing" is impossible.

Thus, to show that the printer-control process in the previous example never reads the contents of a partially filled queue slot (a "bad thing"), we might see that the shared queue is implemented in terms of two variables:

- *NextFull* points to the queue slot that has been full the longest and is the one the printer-control process will next read.
- *FirstEmpty* points to the queue slot that has been empty the longest and is the one where the page-formatter process will next deposit a page image.

We would then establish that $NextFull \neq FirstEmpty$ is true initially and that no action of either process falsifies it. And, from the variable definitions, we would note that $NextFull \neq FirstEmpty$ implies that the printer-control process reads the contents of a different queue slot than the page-formatter process writes, so the "bad thing" cannot occur.

It turns out that all functional specifications for concurrent programs can be partitioned into "bad things" and "good things." Thus, a rule for such "good things" will complete the picture. To show that some "good thing" does happen during execution:

Identify an expression involving the program variables that when equal to some minimal value implies that the "good thing" has happened. Show that this expression (a) is decreased by some program actions that must eventually run, and (b) is not increased by any other program action.

Note our rules for "bad things" and "good things" do not require checking individual process interleavings. They require only effort proportional to the size of the program being analyzed. Even the size of a large program need not be an impediment—large concurrent programs are often just small algorithms in disguise. Such small concurrent algorithms can be programmed and analyzed; we build a model and analyze it to gain insight about the full-scale artifact.

Writing concurrent programs is indeed difficult, but mental tools exist that can help the practicing programmer. The trick is to abandon the habit of thinking about individual execution interleavings.

FRED B. SCHNEIDER'S book, *On Concurrent Programming* (Springer-Verlag, 1997), discusses how assertional reasoning can be used in the analysis and development of concurrent programs.