CHAPTER 13

Ease of Use

"For My yoke is easy and My burden is light."

— Matthew 11:30

In this chapter we consider the ease of use of the Sunrise system for proving programs correct. This includes the burdens of the annotations required for while loops and procedures, and the burdens of proving the verification conditions created. We also discuss the areas of the proof that the VCG supports.

13.1 Burden of Annotation

To prepare a program for submission to the VCG, the Sunrise system requires the user to attach a number of annotations to the program which have no direct impact on the program's execution, and serve only to help the VCG and the proof of the program's correctness. It is reasonable to ask how burdensome these required annotations are, how much is asked of the user, and how a user might be expected to generate such annotations in practice.

Most of these questions are similar to the ones raised in the debate over loop invariants, whether or not the user should be expected to contribute the loop

invariants, and the apparent difficulty of such a task. It has been argued that requiring the user to provide such invariants forces the user to think more clearly about why they should be true, and that they also provide a very useful form of documentation. We consider the question of the propriety of requiring invariants, and other annotations, to be a decision beyond the purview of this work. In this work, requiring invariants and other annotations is a pragmatic necessity. We now examine the difficulty of arriving at such annotations, considering each one in turn.

For while loops, two annotations are generally required, a loop invariant and a loop progress expression containing an expression whose value strictly decreases for each iteration of the loop. The invariant is used to prove the partial correctness of the loop, and the progress expression is used to prove its termination. Gries has studied the problem of generating loop invariants [Gri81] and arrived at a number of principles to guide this task. He has also described how to generate a progress expression (which he calls a bound function) so that each iteration makes progress towards termination.

For procedure declarations, we require several annotations:

1. Global variables
2. Precondition
3. Postcondition
4. For each procedure called in the body, a *calls* progress expression
5. If the procedure recurses, a recursion progress expression.

The burden of generating a complete list of global variables is not hard, but it is not as simple as scanning the body of the procedure. Instead, this should include all globals accessed from within procedures called from within the body of this procedure, either directly or indirectly, any number of levels deep. Thus, the globals list should be a list of all globals that can be read or written during the execution of the procedure body. If procedures are written in a bottom-up fashion, then this would be the union of the globals lists of all procedures called by the body, together with the globals actually used in the body itself.

The specifications of the precondition and postcondition are well-discussed in the literature, and will not be described further here.

The new specification of the *calls* progress expressions expresses a connection between two states, in some ways analogous to the connection expressed by post-conditions. Here, however, we need to take care to refer to the correct variables in the two contexts. The choice of these progress expressions is crucial to the proof of termination, for these are used to generate the path conditions while traversing the procedure call graph, and in creating the call graph verification conditions. These may be created by asking the question, "What sort of progress do I expect to achieve between the entrance of this procedure and the entrance of another called by this one?" We suggest first drawing the procedure call graph and examining it for cycles, to manually focus one's attention on the need to provide meaningful progress towards termination around each cycle. This progress is then expressed in the recursion expression of the procedure. The progress around each cycle then needs to be broken down into smaller steps of progress, which are distributed onto the various arcs of the graph. These smaller steps may in

fact individually show no progress, or even backwards movement as long as it is limited, as may be convenient. The requirement is that the accumulation of the progress of all the arcs around a cycle must show the forward progress of the recursive progress expression. Thus the choice of the recursive progress expression should precede the choice of the *calls* progress expressions.

The need to specify these calls progress expressions and the recursion expression in each procedure's header is welcome, for it compels the programmer to think seriously about the issues of termination for his program. For every possible path of recursion, there must be progress towards termination that can be identified and quantified. Usually this progress will be nascent within the programmer, as part of his design of the program, but the annotation requirements will force him to make these ideas concrete, and to examine them critically. In cases of great interaction among procedures, where the procedure call graph has many interlocking cycles, the expectation of having to prove termination may draw the programmer toward simplified designs with fewer well-chosen interactions.

This annotation structure was chosen as a compromise between the simple rigidity of Sokolowski's recursion depth counter, and the extreme flexibility of specifying the expected progress individually for each call, at the point of call. We chose to require every call issuing from one particular procedure to another to satisfy the same progress condition. This allowed us to partition the proof of recursion into two stages, where in the first stage the calls progress claims were verified by syntactic analysis of each procedure's body, and in the second stage, the recursion progress claims were verified from the calls progress claims by analyzing the structure of the call graph. This followed the compositional

paradigm, where the proof of each individual procedure was accomplished in relative isolation, and then the results of these proofs were brought together to verify the entire collection of procedures.

We feel this is a reasonable annotation structure, because if the programmer wished to prove termination, inherently he would have to describe how to prevent infinite recursive descent, and this leads immediately to a consideration of cycles in the procedure call graph. Each such cycle must be shown to terminate, probably by some form of a well-founded argument. Inevitably the programmer would have to supply information similar to what we have asked for in these annotations, and not having considered the issue beforehand, might choose a simple but overly restrictive system like recursion depth counters. Requiring our annotations at the beginning brings the programmer's attention to termination issues early, and clarifies the expectations of progress between procedures. Therefore this annotation structure would be a welcome element in good software engineering and modular design for implementation by a team.

13.2 Burden of Proof

The verification conditions presented by the part of the VCG that deals with analyzing the syntactic structure of the program appears to be quite satisfactory. However, the production of verification conditions sufficient to establish termination, created by analyzing the structure of the call graph, may allow for substantial reduction in the number of verification conditions generated. One such improvement is discussed in Chapter 14. This may be the subject of a future upgrade of Sunrise.

13.3 Areas of VCG Support

To briefly mention the concepts proven automatically by the VCG without user involvement, the user need not be concerned with proving

1. well-formedness
2. proof by stages of partial correctness
3. precondition maintenance
4. *calls* progress
5. recursive progress
6. termination
7. total correctness

All of these follow from simply proving the verification conditions. We do not mean to imply that the proof of the verification conditions is trivial or easy. They may well contain the bulk of the weight of the proof. However, the above concepts are not themselves trivial, and we contend that this VCG as presented does accomplish a significant task in reducing the difficulty of proving programs totally correct.