Protective Interface Specifications

Gary T. Leavens and Jeannette M. Wing

1 Department of Computer Science, Iowa State University, Ames, IA, USA
2 Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA

Keywords: Specification languages; Underspecification; Partiality; Larch

Abstract. The interface specification of a procedure describes the procedure’s behaviour using pre- and postconditions. These pre- and postconditions are written using various functions. If some of these functions are partial, or underspecified, then the procedure specification may not be well-defined.

We show how to write pre- and postcondition specifications that avoid such problems, by having the precondition “protect” the postcondition from the effects of partiality and underspecification. We formalize the notion of protection from partiality in the context of specification languages like VDM-SL and COLD-K. We also formalize the notion of protection from underspecification for the Larch family of specification languages, and for Larch show how one can prove that a procedure specification is protected from the effects of underspecification.

1. The Problem

This paper seeks to explain and precisely define properties of “good” procedure specifications. These properties say when the precondition of a procedure specification protects the postcondition from partiality or underspecification in the vocabulary used in the specification. While we will precisely define protection for formal specifications, it can be applied and used in even informal specifications (with, of course, less precision).

To explain what a protective specification is, we start with an informal
example. Consider an (ill-defined) specification of an integer-valued factorial procedure, such as that found in Fig. 1. This behavioural interface specification is to be implemented in C++, which explains why C++ syntax is used to specify how it is to be called. The pre- and postconditions follow requires and ensures, respectively; when the precondition is satisfied, the procedure must terminate in a state that satisfies the postcondition. (The keyword informally in Larch/C++ [Lea97] signals the start of an informal predicate.) This specification is ill-defined because it is not clear what the procedure should return when \( x \) is negative. The problem is that mathematics does not define what “the factorial of \( x \)” means when \( x \) is negative, but for that case the specification seems to require a correct implementation to return some integer. Because of this, a correct implementation’s obligations when \( x \) is negative are unclear, making this a bad specification. Note that the problem with this specification has nothing at all to do with the particular mathematical formalism used to write the pre- and postconditions, or with any particular logic for reasoning about what they mean.

A better, yet still informal, specification of the factorial procedure is given in Fig. 2. In this specification the precondition requires that the argument \( x \) is nonnegative, and thus has a well-defined factorial. We say that the precondition of Fig. 2 “protects” the postcondition, because for all values of the arguments that satisfy the precondition, the vocabulary used in the post-condition is well-defined. Thus whatever the phrase “the factorial of \( x \)” might mean when \( x \) is negative does not matter.

The concept of protection, even in informal specifications, does have one subtle twist. It is that one part of a precondition may protect other parts of the precondition itself, so that the entire precondition is well-defined. Most programmers are familiar with examples where they must check that a number is nonzero before checking some condition involving a ratio or modulo calculation. The same idea applies in specifications such as the one in Fig. 3, where the first conjunct in the precondition (“denom is positive”) protects the second. That is, if the first conjunct is false, the entire precondition is false, and so the meaning of the second conjunct does not matter, as the implementation will not have any specified behaviour in such a case.

In the example of Fig. 3, the (informal) logic used to reason about the meaning of the precondition matters. In our informal argument we assumed that if the first conjunct in the precondition is false, then the entire precondition is false (and hence well-defined). However, since the precondition is informal, one could plausibly argue that since the “/” operator used in the second conjunct is partial,