Abstract

Background

Position

Current literature provides many reasons for reusing software [ ]. Many system level components have been identified and formally described as candidates for reuse [ ]. Software engineers look for software that does all or part of what they need so as to avoid wasting time on the design and implementation of programs or program parts that have already been done before. When such components are available, designers and developers alike can realize significant increase in productivity.

Having given our support to the notion of increasing productivity by reusing existing software components, we now want to point out two compelling reasons not to reuse software — first, when it is not completely clear what the software does and secondly, when the software is not correct. It is our position that clients can know what software does only when that software has been formally specified. As to correctness, we believe that the reuse community should strive toward the goal of mathematical proofs certifying that a given implementation meets its formal specification.

Several ways for both specifying and verifying programs appear in the literature, most falling into one of two classes. In the algebraic approach each program component is specified as an axiom system and arguments concerning correctness are given in terms of the theory described by that axiom system. A major drawback in this method is that one must invent a new theory for each component, hence raising such issues as soundness, expressiveness, and completeness.

In a model based approach the specification for a given component is written in a language entirely separate from the programming language. When possible the one describing the component uses an existing mathematical theory in the description, thereby doing reuse even at the mathematical level. The implementation is then written in some programming language and a set of proof rules for the constructs of that language are used in checking correctness.

For simple language constructs such as assignments, loops, and conditionals proof rules are quite well-known and have been used for a long time. However, when it comes to dealing with more complex program components, such as abstract data types, proof rules and their associated semantics are necessarily far more complicated, most such rules depending on a function that relates implemented objects to conceptual ones. It has been shown that this relationship between conceptual and realization objects is not necessarily functional, but may in fact be relational. Moreover, while proof rules for abstract data types have been proposed, those rules have not been checked for soundness and completeness. Indeed, to show soundness and completeness, one must first provide a formal semantics for the notion of what it means for an implementation to meet its specification.

Although this notion may seem to be clear intuitively, we must find a way to express this idea formally. Moreover, even at the intuitive level, there are some challenging aspects of this issue to consider. For example, suppose one implements a minimal spanning forest specification using Kruskal’s algorithm. A correspondence between the code and the concept is necessarily relational because at each step in the process of building the spanning forest, there may be several edges from which to choose.

There are actually at least three categories of implementations to consider.
• Those that have a one-to-one match with the concept. For example, if one specifies a stack as a string, and implements that specification with an array and top index, the correspondence is functional, matching the conceptual array to the concatenation of the elements in the array from the first to the top.

• Those for which a given implementation may have a relational correspondence with the concept, because for each object in the realization space, there may be multiple matching objects in the concept. The spanning forest example fits here.

• Those for which the realization may get away with a less complicated structure than the concept describes. For example, given a specification for a two state machine with operations change state and read state, in which the read state operation has a pre-condition that allows reading only when the machine is in the initial state, one might use an implementation with only one state. Here the correspondence can be given only by introducing and adjunct variable.

The one-to-one matching is easy to deal with formally, but the other two cases present technical difficulties when trying to provide a proof system that is both sound and complete.