Transformational implementation

by DAVID WILE and ROBERT BALZER

Information Sciences Institute
Marina del Rey, California

Only modest gains in programming productivity have been produced in 25 years of software research, but the groundwork has been laid for major advances through rationalization and automated aids. This groundwork rests on two critical ideas: that specification must be separated from implementation, and that the separation between these two processes should be a formal operational abstract (i.e., very high level) program rather than a nonoperational requirements specification. Structured programming represents the first results of combining these ideas. It is a special case of a more general two-phase process, called Abstract Programming, in which an informal and imprecise specification is transformed into a formal abstract operational program, which is then transformed into a concrete (i.e., detailed low-level) program by optimization. Abstract Programming thus consists of a specification phase and an implementation (optimization) phase which share a formal abstract operational program as their common interface.

The concept of Abstract Programming is completed by adding the feedback loops required by Testing, Maintenance, and Tuning. In conventional programming where no abstract program exists, these feedback loops all operate on the optimized concrete program. On the other hand, in Abstract Programming, if an effective method can be found for guaranteeing the validity of an implementation (that is, the functional equivalence of the abstract and concrete programs), then the validation process can be shifted to the specification phase to show equivalence between the user requirements and the abstract program. Thus, validation could, and should, occur before any implementation. Furthermore, if the implementation process could be made inexpensive through computer aids, then maintenance could be performed by modifying the specification and re-implementing it rather than directly modifying the optimized concrete program, as is current practice. The importance of such an advance can be recognized when one realizes that optimization obscures clarity and thus makes it difficult for maintainers even to understand how the concrete program operates.

It is therefore clear that major advances in programming will hinge on the ability to provide an inexpensive optimization process with guaranteed validity so that maintenance and validation can occur in the specification phase on the abstract program rather than in the implementation phase on the concrete program.

Several researchers have espoused an approach which we will call Transformational Implementation (TI), in which equivalence-preserving transformations are successively applied to the abstract program to effect such an optimization. The key to this approach is that while optimizing transformations are selected by the programmer, the program is transformed by the computer system. Hence, the programmer designs the optimization; the machine ensures its validity and transforms the program.

In addition to guaranteeing the validity of the implementation, drastically reducing the time and effort required to implement a system, and enabling maintenance to be performed at the conceptual level on the abstract program, this approach would also allow programmers to experiment with alternative implementations to widen their experience base and improve their capacity to design optimizations.

We have undertaken a study to discover: what facilities are necessary for the TI approach; what transformations are required; how are they expressed; what categories do they fall into; how can they be named; what kinds of applicability criteria are required; what processes are required to verify that these criteria are satisfied; what instrumentation facilities are required to test a transformation’s selection criteria; what proof techniques are required to validate the transformation; etc. In short, the study will provide an experience base for the TI approach from which we can design and implement a prototype system.

REFERENCE
