Guest Editorial* . . .

RAYMOND T. YEH

Department of Computer Sciences,
University of Texas at Austin, Austin, Texas 78712

The phrase "reliable software" has two distinct shades of meaning. On the one hand, it implies *correctness*. A piece of software is correct if it meets its functional specifications. This implies that as long as the inputs satisfy the specifications, the software will produce the desired outputs. On the other hand, reliable software implies *robustness*. A piece of software is robust, or fault tolerant, if it can be expected to deliver a certain minimum level of services even when faced with an unexpected or hostile environment (such as hardware failure or bad data). Thus a piece of software is reliable if it is both consistent with respect to the stated specifications and able to withstand unexpected demands.

Software validation is concerned with analyzing software to determine the extent to which it performs the logical functions intended by its creator. Techniques in software validation can be classified into two main categories: testing and verification.

Software testing is concerned with analyzing a program by evaluating its response to a selected set of input data. Since any test data will necessarily be a very small sample of the possible inputs, testing is inadequate for achieving a complete understanding of the logical or the performance properties of real programs. However, testing may be a necessary step in program vali-

dation not only to clean up certain obvious bugs, but also to reveal unexpected behavior and thus aid a programmer's understanding of his program. A recent paper of Gerhart and Yellowitz [3] strongly emphasizes the need for testing. It should be pointed out, however, that techniques for testing remain rather ad hoc. The paper by Goodenough and Gerhart [2] is one of the first attempts toward a theory of testing. The paper by Fosdick and Osterweil in the September 1976 issue of *Computing Surveys*, the first of the two-part Special Issue on Reliable Software, represents another attempt to lay a theoretical foundation of detecting errors in software. In the Fosdick-Osterweil paper, the notion of path expression is used to describe actions on data and to classify data flow along paths within subprograms and along paths which cross subprogram boundaries. It is shown that fast algorithms, originally introduced for global optimization, can be used to implement this classification scheme and to determine data flow anomalies.

There are two ways of approaching program verification. The *static approach* considers a program and its specifications to be given. Mathematical proofs are developed to demonstrate that the logical behavior of a program is as specified, viewing this logical behavior as completely characterized by a set of formal assertions. The *constructive approach* lays stress on the correct development of a program. However, it seems that both approaches rely on the programmer's ability to abstract certain sufficiently strong

* [Editor's Note: This Special Issue on Reliable Software is published in two parts in *Computing Surveys*: Part I: Software Validation (September 1976 issue), and Part II: Fault-Tolerant Software (December 1976).]

invariant properties of the program, intended or given. These properties form the inductive hypothesis that incorporates certain inductions for proving the correctness of loops [4].

The paper by Hantler and King also in the September 1976 issue gives a tutorial introduction to the static approach. The basic idea of this approach stems from Floyd's observation [1] that a computer program can be thought of as a mathematical object, and hence its properties can be studied in a rigorous fashion. Floyd shows that it is possible to capture the "invariant" properties of a program at each node of a flowchart program by means of a formal assertion. To show that a program is consistent with its specifications, it is necessary to show that the input assertion implies the output assertion over all possible paths between the start and halt nodes. The Hantler-King paper discusses the essence of this approach using symbolic data and also demonstrates how the technique of "symbolic execution" can be used for program testing.

The notion of robustness, or fault tolerance, is concerned with providing design redundancy in software such that it can continue to perform, perhaps in a degraded mode, under changing requirements or hostile environments. Providing redundancy is, of course, a common practice in any engineering design. Buildings, bridges, highways, and other constructions, are built with certain tolerance factors. In the last decade there has been a great amount of research into the design of fault tolerant computer hardware. However, research efforts in software fault tolerance began relatively recently.

Comparing the different branches of software engineering, we find that compiler engineering has matured in the sense that certain principles and procedures have been established for the systematic construction of a compiler. No papers appear in this issue on compiler construction. On the other hand, operating systems engineering seems to be nearing its maturity, and hence, certain principles, techniques and guidelines have emerged. Two papers in the December 1976 issue, one by Denning and the other by

Linden, are concerned with structuring reliable operating systems.

Denning's paper emphasizes the importance of software/hardware integration in order to achieve system fault tolerance. Four principles have been singled out as effective guidelines for error confinement. These principles are discussed in the context of a capability-based machine to emphasize the point of hardware assistance in software fault tolerance.

Linden's paper is concerned with two specific concepts, small protection domains and extended abstract objects, as means of enhancing the security and reliability of software in general and operating systems in particular. The content of this paper overlaps that of Denning's contribution insofar as both regard the capability-based machine to be an effective hardware architecture through which to implement the software concepts for reliability. On the other hand, the two papers complement each other very well in that Denning picks out a set of principles as general guidelines whereas Linden surveys a wide spectrum of techniques utilizing these principles. Nevertheless, the principles and techniques discussed in both papers illustrate the outlook which a capability machine provides for total system fault tolerance.

Finally, the paper by Hecht in the December issue is concerned with the redundancy architecture design for real-time applications software. It focuses on the concept of how alternative modules can be added to enhance reliability. Hecht provides some sample computations giving information on the amount of improvement in reliability relative to the cost of the amount of redundancy added. These computations are very illustrative indeed.

To be sure, we are a long way from the goal of systematic development of reliable software. It is hoped, however, that this two-part special issue of *Surveys* will give readers a sense of perspective about the fundamental issues involved in reaching the goal.

I would like to take this opportunity to thank the following colleagues for their contributions in getting this special issue into shape: Lloyd Fosdick, Leon Osterweil,

Sidney Hantler, James King, Peter Denning, Herbert Hecht, Ted Linden, Karl Levitt, James Peterson, Maurice Wilkes, Dick Fairley, J. C. Huang, David Morgan, Donald Reifer, and William Howden.

REFERENCES

- [1] FLOYD, R. W. "Assigning meaning to programs," in *Proc. Symp. Appl. Math.*, 1967, Vol. 19, American Mathematical Society, Providence, R. I., 1967, pp. 19-32.
- [2] GOODENOUGH, J. B.; AND GERHART, S. L. "Toward a theory of test data selection," in *Proc. ACM 1975 Internatl. Conf. on Reliable Software*, ACM, New York, 1975, pp. 493-510.
- [3] GERHART, S. L.; AND YELLOWITZ, L. "Observations of fallibility in applications of modern programming methodologies," *IEEE Trans. on Software Engineering*. To appear in the December 1976
- [4] REYNOLDS, C., AND YEH, R. T. "Induction as a basis for software verification," *IEEE Trans. on Software Engineering*. To appear in the December 1976.