A panel session—Formal methods in programming—When will they be practical?

SESSION CHAIRMAN—KARL N. LEVITT
SRI International

Panel Members
Donald I. Good—University of Texas
Ellis Horowitz—University of Southern California
Barbara Liskov—Massachusetts Institute of Technology
Lawrence Robinson—SRI International

PANEL OVERVIEW—Karl N. Levitt

Formal methods, i.e., use of mathematical rigor, have been employed by research computer scientists in their attempt to develop general results for many aspects of computer science, e.g., computational complexity, undecidability, numerical analysis, programming language semantics. Much of this work has had little impact on those charged with producing working software systems. However, in recent years numerous researchers have suggested that by applying formal methods to the realization of systems, the quality of such systems could be significantly improved. Such formal methods could be applied in the structuring, specification, verification, and analysis of performance for systems. The position statements below explore the use of these techniques in the production of systems. The general opinion is that formal methods will ultimately assume a vital role, but for the present their use will be restricted to particular systems produced by skilled individuals. The use of the formal methods will gradually increase as the techniques are refined and applied to a larger variety of systems, as tools are developed to support their use, and as the general community becomes better educated in formal methods.

FORMALISM CAN HELP YOU—Lawrence Robinson

There is much resistance on the part of the programming community to the use of formalism in programming methods. However, there is also a problem in achieving reliable and maintainable software that meets its requirements. Despite the resistance, the application of formalism to programming techniques shows much promise in attaining a software product of high quality. At present, some advanced techniques are ready for limited use.

First it is useful to state exactly what formalism is. Formalism is a means for making logical and consistent arguments about a particular area. To formalize an area requires a set of definitions and axioms to characterize the area, and a set of rules of inference to validate or disprove logical arguments. Formalism can be used to understand, analyze, and reformulate both a problem and its proposed solution, and to verify that the proposed solution actually solves the problem at hand.

Other engineering disciplines make use of formalism to some extent (e.g., Laplace transforms in electrical engineering and partial differential equations in structural engineering). In fact, some aspects of software already make use of formal techniques, such as the exact specification of the syntax of a programming language. However, software engineering, because it is still a very young field, lags behind other engineering endeavors in the use of formal techniques.

In addition, the special problems inherent in software engineering require an even greater use of formalism than do the problems of other engineering endeavors. Two factors are responsible for software engineering’s greater need for formalism:

• The complexity of large software systems exceeds that of other engineering systems by several orders of magnitude. Any engineering practices that are developed for software must place considerable emphasis on the management of this complexity. Current software engineering methods have not adequately done so. Formal mathematics has been the classical tool for managing complexity, by combining many cases into canonical groups, and by structuring complex concepts by the use of abstraction and formal definition.

• Software has a requirement for total exactness that is unique among engineering disciplines. Whereas the products in other fields are required to fall within continuous tolerances, the smallest error in a piece of software (a single bit) will sometimes result in the catastrophic failure of a system, rather than in degraded performance. Software engineering techniques must be able to guarantee the exact fulfillment of requirements. Formal techniques provide the only possibility of exactly stating the requirements of a piece of software, and for mathematically proving that the software meets its requirements.
Because software is so difficult an area, its development requires a great deal of attention to the analysis and planning that precede the coding. Current techniques underemphasize these early aspects of the software development process. Some new techniques require formal analyses of requirements and formal design specifications before coding can proceed. These techniques are ready for use by advanced systems programmers. A methodology developed at SRI International (HDM—the SRI Hierarchical Development Methodology) has been used for formally specifying the designs of several operating systems and many intermediate-sized systems; some of the systems have been implemented.

However, these techniques are not currently ready for everyone, and considerable training is required in order to use them. The major obstacle, however, is a resistance to change. But the education process (at universities, for example) is making these techniques available to people entering the software field. Gradually, programming will become closer to mathematics—more of a science than a black art. The changing process will be painful and fraught with resistance, but it hopefully will result in better and better software products.

SOFTWARE DESIGN BY ALGEBRAIC SPECIFICATION—Ellis Horowitz

I’d like to adopt for the moment an extreme point of view so I can describe the range of application of algebraic specification to software design. A recent advance in software construction has been the recognition that one should begin by supplying a formal specification of the intended task. This notion applies both to programming systems as well as to individual programs. The debate now centers on the way such a specification will be phrased, and as with programming languages practitioners are developing strong biases for and against certain approaches. My own bias centers on algebraic specification.

Algebraic specification is not appropriate for specifying all possible tasks. For example, the technique does not seem natural for describing concurrency. What it does seem ideally suited for is the description of data types (data structures). Many computer applications can be conceived of as the transformation of data from one form into another. Then, the programming task becomes one in which the logical data structures must be precisely defined and then represented via other data structures which are available on a computer. The essential ingredient which makes the algebraic approach so successful for data type specification is the fact that a data type is composed of a set of related operations. These operations create, build-up or make smaller instances of the data type; they may decompose it into its constituent parts or answer questions about its form and content. All of these operations are best viewed together and algebraic specification naturally exposes these relationships.

The relationships between operations of a data type are given via a set of equations, usually called algebraic axioms. These equations show the effect that the operations have on instances of the data type. One important virtue of these equations is that they imply no representation. Thus the meaning of the operations of a data type is defined abstractly (i.e., without regard to the eventual representation), and the design process is broken into two logical phases: data type definition and implementation. My own experience leads me to conclude that an algebraic axiomatization is about as easy to compose for a data type as it is to describe the type in English.

The ease of specification might alone be sufficient to recommend algebraic axioms. But it turns out that there are other payoffs when one adheres to this software design scheme. In particular the testing and verification of the resulting software are all facilitated when one continues to design and implement all data types via the algebraic discipline. A software “environment” can be created which aids the programmer as he builds his system. At the USC/Information Sciences Institute several of us (D. Musser, J. Guttag) have created the Data Type Verification System (DTVS) which supports program creation by algebraic specification.

We begin by algebraically axiomatizing a single data type. Is our axiomatization correct? Before attempting a proof some testing would be desirable. DTVS will automatically synthesize an implementation from the axioms. This will permit testing at the level of the axiomatization without the trouble of implementation. Naturally such testing will be costly in terms of execution time, but experience indicates that only small cases are sufficient to turn up most errors.

When testing is completed, a representation for the data type in terms of other data types is devised. Then implementations of the operations in terms of the operations of the implementing data types are given. Once again testing is repeated, but this time DTVS will use the implementations as the semantics of the data type. Once we are satisfied, a proof of the correctness of the implementation is carried out semi-automatically by DTVS.

Specification is a process we continue to explore. It is possible to build a complex software system by strictly adhering to algebraic axioms. By doing so one gains both the virtues of a formal approach and the testing and verification environment. This offers much more than an ad hoc strategy. Finally, this approach is not inconsistent with the eventual use of a conventional programming language.

PRACTICAL BENEFITS OF RESEARCH IN PROGRAMMING METHODOLOGY—Barbara Liskov

In the past few years, considerable progress has been made in the area of programming methodology. Although all research in this area is interrelated, two main research directions can be distinguished. One direction is the study of software system structure, in particular study of desirable kinds of modules and module interconnections. The other direction is the study of the process of developing correct software having such desirable structure.

Study of software system structure has been particularly effective within the framework of research on programming
languages, especially the languages CLU\(^1\) and Alphard.\(^2\) This language work has succeeded in identifying new kinds of modules, and has provided precise rules governing the implementation and use of such modules. Each kind of module supports a kind of abstraction found to be useful in constructing software. The most important new kind of module is that supporting a data abstraction; however, there are other kinds of modules, and in addition rules governing the interaction of modules. These latter rules constrain and reduce the interconnections among modules.

Earlier work in structured programming\(^3\) and stepwise refinement\(^4\) made evident the advantages of top down development of software. Recent work has elaborated the top down development process, taking into account the work on software system structure discussed above, and clarifying the way that top down development proceeds. The most important contribution has been the recognition of the role played by program specifications. A program specification is a description of the behavior of a module, the behavior that will be depended on by any user, and that must be provided by any implementation. At any stage of design, the goal is to identify lower level abstractions useful in implementing the current level. As these abstractions are identified, their behavior is specified. The specification is given in advance of the implementation, and it provides a complete description of the interface of the module that will later implement the abstraction. The presence of the specification permits the question of how to implement the lower level modules to be deferred until a later stage of design.

As programming methodology has become better understood, there has been increasing interest in defining formal methods to support it. In particular, there has been much recent research in formal specification techniques,\(^5,6\) which permit the specifications discussed above to be expressed in a formal language, i.e., one with a well-defined and unambiguous syntax and semantics. The advantages of such formal specifications are twofold: they are more precise and concise than informal specifications, and therefore may serve better the role of interface descriptions described above, and, in addition, given formal specifications, a formal proof that a module's implementation satisfies its specification is possible. However, formal specifications are more difficult to write than informal ones, and our current understanding of specification and verification techniques is insufficient to permit all useful abstractions to be described.

It is my belief that the present and near future construction of software systems can best be helped by popularizing the methodology. This can be done in a way that relies neither on a particular language, nor on the as yet incompletely understood specification and verification techniques. Instead, the following two ideas must be made clear to programmers:

1. What constitutes good modularity.
2. What constitutes good design practice.

Rules about good modularity are best explained by developing conventions, or better yet preprocessors, for existing languages in actual use; such conventions would permit a limited use of data abstractions and would prohibit current bad practices (such as non-local use of data). By expressing the rules in this way, they are explained in terms the programmers can understand, and furthermore, a tool is provided that helps in the development of a well-structured system. Good design practice then consists of top down decomposition into the kinds of modules that the programmers already understand, with emphasis on the role of (informal) specifications in the process, and especially on the necessity of specifications being given in advance of implementation. To aid programmers in understanding what specifications are, it is helpful to establish a specification standard which describes the kind of information that should be included in the specification, and gives a format for expressing that information. However, it is too early to require that specifications be given in a formal language. Formal methods in system design are not yet ready for practical use, but I believe use of the methods in an informal way can have considerable practical benefit.

REFERENCES

5. Guttag, J., E. Horowitz, and D. Musser, Abstract Data Types and Software Validation, Report ISI/RR-76-48, Information Sciences Institute, University of Southern California, Marina del Rey, August 1976.

BEYOND FACTORIAL—Donald I. Good

Factorial is probably the single most thoroughly verified program ever. It, and other small examples such as stacks, typically are used in the literature to illustrate formal program verification methods, and this tends to create the impression that these small examples define the limits of effectiveness of formal verification. We must be careful not to paint an overly optimistic picture, but the view that formal verification methods are effective only on small examples, such as stacks and factorials, is equally overly pessimistic. Formal verification methods have been applied successfully to a variety of programs in the 1000-lines-of-code range and strong verification clauses are now being written into some software development contracts with the U.S. Government. Although there is much yet to be done, formal verification methods show considerable promise of becoming an effective approach to building reliable software in actual practice. Basic research over the last ten years has done much to develop formal methods of program verification. The prob-
The problem of actually verifying a 2000- to 5000-line program, however, is much larger than just verification methods, and we must begin to focus some of our efforts on solving the "whole" verification problem if verification methods are to become effective in actual practice. The breadth and depth of the problem that is faced by a person charged with producing a substantial verified program is substantial. It encompasses the following issues: verification methods, specification methods, programming methods, specification language, verification language, programming language, specification tools, verification tools, programming tools, development strategies.

Verification methods are the central issue, but are by no means the only issue. A verification demonstrates the consistency of a program and a specification. Therefore, attaining a verification requires using specification and programming methods that are both compatible with the verification methods. (By "programming methods" we mean the use of program constructs that are verifiable, as opposed to program "development strategies"). We must know what kinds of things can and should be specified about various program constructs and we must know how to verify that the program construct meets the specification. We also must know how primitive verifiable program constructs can be put together to form large verifiable programs. Given the basic methods, precise languages for expressing the specifications, program, and verification are required. The specification and programming languages express the actual program and its desired properties, and the verifications are done in the verification language which normally is some variation of the predicate calculus. The development of suitable languages involves all of the well-known problems of language design. Given these languages, effective tools are required to apply these languages to an actual problem. Without adequate tools, the languages and the methods they embody become effectively inaccessible. Finally, these tools must be used according to some reasonable strategy for step-wise program development. Just as independent compilation is of major importance in developing a large program, independent verification is crucial even for a program of modest size; thus step-by-step verification must follow a sound program development strategy. Attaining a verified program of substantial size requires effective solutions to all of these problems and, just as importantly, it requires that these solutions be effectively integrated so that they can be used together in a systematic and coherent development of a verified program. In actual practice, an inadequate solution to any of these problems or an ineffective integration of individually effective solutions can completely defeat the verification of a 2000- to 5000-line program.