For several decades there have been serious and extensive efforts in building, refining, and extending formal tools in order to support the design and analysis of large software systems. An abundance of formal tools, theories, and methods have been proposed and discussed. Despite their having a value in their own right as theories, and having constituted a very rich direction of research in Computer Science, there has also been a permanent discussion and challenge to what extent the formalization and formal treatment of design and analysis problems could contribute to, or even bring about, solutions for real-world systems.

While originally a confrontation between theory and practice, the discussion is by far no longer between academia (theory) and industry (practice). There are serious efforts, in industry projects, to use formal methods for software development (e.g. for prototype verification). At the same time one finds innovative formal academic work that stems from practical design experiences (e.g. in real-time systems). Consequently there is a wealth of difficult and different problems and perspectives regarding the relevance of formal methods for practical software system design. The following gives the actors in their corresponding roles:

**Proponent:**

- **Prof. David Gries (Cornell University)**
  - correct program development
- **Dr. N. Shankar (SRI International)**
  - formal verification of fault tolerant systems
- **Prof. Kwei-Jay Lin (Univ. of Illinois, UIUC)**
  - formal methods for real-time systems design

**Respondent:**

- **Dr. Mark Ardis (AT&T Bell Laboratories)**
  - use of formal methods in industry
- **Prof. Joseph Urban (Arizona State University)**
  - Meta-Language Based Theory of Specifications
- **Prof. Horst Wedde (Wayne State University)**
  - theory of distributed systems, prototype refinement and verification

Since for our theme there are indeed more open questions than answers or viable solutions, an innovative format of dialogue, even controversial discussion has been chosen for our panel discussion. Prominent representatives of three formal approaches will present their findings and experience about the use and the usefulness of formal methods. Their deliberation is based on an extended written statement prepared by the respective proponents. Each of these statements, after possible revision through interaction with the moderators has been mailed to another scholar, each of which has been asked to act as a respondent. In this role, a complementary, even critical, evaluation of each formal method, has been prepared, condensed into another written statement that was made available to the corresponding respondent of each approach. During the panel session, each respondent will discuss his view immediately following the respective proponent’s presentation. So the discussion, while being open as any panel discussion, is both well prepared by the panelists and focussed.
Are Formal Methods Useful in Software Development?

David Gries
Computer Science, Upson Hall
Cornell University
Ithaca, New York 14853

SCOPE OF DISCUSSION. I will not address the question of the mechanical use of formal methods—for example, having proofs checked on the computer—since that topic is being addressed adequately by the other panelists. Instead, I will concentrate on the application by people of formal methods in software development. I limit my discussion to the following areas of formal methods:

(0) The use of formal notations (the predicate calculus, extensions to it, like Z) for specifying algorithms or program segments and for manipulating and analyzing specifications.

(1) The use of formal theories of program correctness and the principles for program development that arise from them, both for developing programs in a calculational manner and in proving programs correct. Examples of such theories are a weakest-precondition formulation of an imperative programming language, extensions to it to deal with data refinement, and the Bird-Meertens calculus for dealing with functional programs.

I do not claim that people can use or should be using formal methods to write every specification and every program. My thesis is the following.

THESIS. Today, Jack and Jill programmer—working in industry or as computer scientists who develop algorithms or programs on a regular basis—would be far more productive and produce work of far higher quality if:

(0) They had a real understanding of and facility with the propositional and predicate calculus.

(1) They had a real understanding of and facility with theories of program correctness (for imperative and functional programs) and the principals of formal, calculational, program development that emerge from them.

Jack and Jill would be far more productive not because they would use formal methods all the time but because they would have understood and mastered the (mental) tools of their trade. These mental tools would color almost all they do. They would use the formal ideas in an informal manner continually.

They would have the ability to be more formal when necessary—usually on the more complex parts of a specification or program.

People shy away from the formal development of programs because they are not fluent in formal logic (propositional and predicate calculus). The formal theories mentioned above should be taught in the first two years of every undergraduate computer science curriculum. Provided the teachers themselves were facile with the formal methods (which, in general, they are not), the upper-level curriculum would benefit greatly.

ON EDUCATION IN FORMAL LOGIC. The formal logic typically taught is inadequate. Sometimes it is taught by philosophers, whose viewpoint is so different as to make the course useless. Often, CS people teach logic as an object of study, and not as a useful tool, and too little time is devoted to it. The concepts are rarely used throughout the rest of the course, and the students come away with the impression that logic is simply an academic exercise, to be looked at and forgotten. They do not practice it enough to become really fluent. They don't see enough applications that show its real use.

One hears the complaint from time to time that students are unable to PROVE things. They don't have a solid understanding of the notion of proof, they lack the ability to manipulate symbols according to given rules, and they lack the ability to use formal ideas in an informal way while realizing that they are doing so. Students CAN acquire these abilities; we are not teaching them effectively. When we complain that students lack ability to write proofs, to argue coherently, perhaps we should take it as a sign of our own failings as teachers rather than as the failings of the students.

An equational logic should be taught, since it is more usable by people than, say, natural deduction. Natural deduction immediately turns students away, because it is not useful to them as a tool at their stage of learning.

EVIDENCE FOR MY POINT OF VIEW. I have not written a large programming system lately, so I cannot give first-hand evidence that formal methods can be applied to large programs. However, I do have the following first-hand evidence of the use of formal methods.
(0) As editor for IPL, I have helped countless authors vastly improve their papers simply by rewriting their algorithmic presentations, often using formal ideas in an informal way.

(1) I have seen a 300-page industrial specification of a large programming system. It was badly organized and riddled with errors and inconsistencies. Our investigations showed that it could be rewritten and made clear in a 200-page document. Had the programmers in the project been well educated, they would have refused to work with this specification; as professionals, they would have first worked with the (in-house) customer to make the specification clear, concise, consistent, etc. Education in formal methods helps one attain such an attitude.

EVIDENCE FROM STUDENTS. My assertions concerning formal methods have little weight if they are not backed up by others. At the end of my junior-senior level course on the "science of programming", I asked the students to write an essay on their view of the course. Here are some quotes from those essays. Many of the students wrote about the large picture regarding knowledge of formal methods, and not just about using formal methods in developing a program.

One of my weakest areas, with which I have always struggled, is doing proofs. No matter how well I knew the material, it has always been difficult to get from point A to point B logically and correctly. Being forced to do this, while keeping a clear and structured format, as we have continually done, has improved on this immensely. The instruction in formal logic coupled with its use in proving and developing programs has also been very beneficial. The use of formal logic in the class helped clarify my thinking. The subject I feel most confident with is propositional calculus. It helped me the most in a very costly problem I have: carelessness. Curiously, it also made me aware of how incoherent and verbose my speech is. I learned the importance of being rigorous. It does not make me reason better or more intelligent, but I am less prone to mistakes. The practice in using formal logic and reasoning, as well as the frequent examples of the practical use of these tools, helped me develop a clearer, more analytical approach to algorithm development and programming.

We all must learn to crawl before we walk, to walk before we run. So please teach students to write short programs correctly before making them write long, hard programs. This course should be modified to make it a bit less rigorous and taught to students of introductory courses. This course has altered every notion I had about programming... I finally came to grips with the idea that developing a program is not a spontaneous activity. ... I used to sit down, turn on the computer, and start typing. I learned there are more scientific ways of doing things. ... I found the course experience very enriching, despite the fact that I lost any confidence I had as a programmer.

The most important thing I learned is that, no matter how intuitive you are, you are always better off formalizing the problem than "fumbling in the dark", guessing statements. In the final analysis, this course should be required of every CS major because it encourages good habits.

There is only one solution ... education. A person doesn't have to be taught how to guess programs, but how to develop them. If the formal approach is taken right from the start, the student will develop a "formal instinct". The problem with the old methodology is that our ability to prove programs is far behind our ability to write them. This course should be taken by all CS majors; it teaches principles and methods that, when used, result in clear, concise, correct programs.

When solving problems, I like to have a solid logical framework within which to build my solution. I have had this sort of framework in my physics and mathematics courses and, even, to a certain extent, in the philosophy and government courses I have taken. But I never had a firm logical framework with which to solve programming problems. Thus, when I programmed, it was more difficult to reach a correct answer. ... Had I known the formal concepts I learned in this course, I would have saved myself many hours of frustrating debugging. ... Now, I use formal methods to aid in my programming. My mind has now been trained to think more scientifically as I program, which helps me even when I am not consciously applying the principles. The material is useful because it provides a fundamental framework in which to operate when programming. It gives meaning to the word "science" in the term "computer science". many formulas and equations that describe the concepts. Underneath this large set of ideas is a smaller, more fundamental one which gives the larger one cohesion. This small set serves as a framework in which to understand physics as a whole; without it, one's abilities to solve problems are diminished. The method taught in this class provides such a framework in the domain of CS.

EUROPE AND FORMAL METHODS. The Europeans seem to be ahead of us in the formal methods game. Their industry seems to accept formal methods much more readily. The U.S. seems to shun formal methods. I even saw a very short referee report for a grant proposal about education in formal methods: "formal methods won't work in the States." In the short run, this attitude probably doesn't hurt our competitiveness. Our good old Yankee enthusiasm and creativity and hard work seems more than adequate. In the long run, however, this attitude will hurt.
Are Formal Methods Useful in Software Development?
(Response to Position of David Gries)

Mark A. Ardis
AT&T Bell Laboratories, 1G-353
1000 E. Warrenville Road
Naperville, IL 60566-7013

David Gries proposes that all programmers would be more productive and produce higher quality products if they would learn two things: (1) predicate calculus, and (2) program correctness (including formal program development). I have no disagreement with the first item, but I object to the second item because it teaches the right skills the wrong way. Programmers should first learn to analyze programs formally before they learn to develop programs. In other words, they should learn to read before they learn to write.

Most programmers spend little time writing programs. Far more time is spent reading and studying programs to test, fix, or enhance them. Also, most code that a programmer studies was written by someone else. Formal analysis should make this job easier. But, most programmers have never been taught how to obtain a specification from a program.

The study of program correctness usually begins with the calculation of weakest preconditions. Although the specific task of calculation is better left to automation (e.g., verification condition generators), the process of code analysis is beneficial. It is worth teaching weakest precondition calculation if only to show which parts can be automated.

Formal, calculational program development is less valuable. While it may be possible to obtain a correct specification before writing the implementation, the development of correct programs from correct specifications is not a significant problem. And, errors that are made during this step are far easier to discover and repair than errors made during specification. (See, for example, the discussion of error introduction and removal rates in chapter 24 of Software Engineering Economics by Barry Boehm, Prentice-Hall, 1981.) But, the real waste in formal program development is its emphasis on refinement, rather than abstraction.

Abstraction is much more important to programmers than refinement. It is a fundamental skill that is used throughout the lifecycle, not just during the implementation phase. For example, most requirements begin as examples or scenarios. It takes considerable skill in abstraction to identify and describe the requirements in their true generality. Also, abstraction is exercised more often than refinement, since programmers spend more time reading code than writing it.

Unfortunately, abstraction is harder to learn and practice than refinement. When refining a specification into a lower-level language most of the effort goes into dividing the task into separate pieces. This is much easier than collecting many pieces to form one abstraction. Also, refinement tends to be bounded by the object language. It is more difficult to see the top than the bottom, since there are so many possible tops.

How should we teach abstraction? There are at least three steps to be mastered:

1. Students need to learn to distinguish between those details that are essential to the problem and those that are accidental to the solution.
2. Students need to learn how to bring together disparate pieces of information so that they can be generalized. The trick is in recognizing those pieces that fit together and those that do not.
3. Students need to express abstraction precisely and unambiguously.

The first two skills involve reading, not writing. The last skill involves writing, but writing in predicate calculus (or an equivalently precise and abstract formal language), not writing in a programming language.

It might be argued that programmers must still be taught how to write correct programs. They will still need to learn program calculus, even if it is only a small part of their job. Perhaps, but their efforts will be little appreciated if no one learns how to read their programs. More importantly, programmers must learn to express ideas at many levels of abstraction, not just as programs.