Formal methods in programming and microprogramming

Two of the three sessions in this area are about microprogramming, and the third is about programming in general. The dominant theme in all the sessions is the current attempt by researchers and practitioners to make programming a formal process. Roughly speaking, this means the use of mathematical rigor in defining, specifying, translating, transforming, analyzing and verifying programs (including microprograms).

The work is motivated not by a love of formal methods for their own sake, but by hard economic objectives. Users are now demanding increased confidence in the adequacy and reliability of delivered software, greatly reduced cost of production and maintenance, and greatly increased flexibility with respect to change, growth and transportability. These demands are not being met by the informal techniques that have accumulated over the years. The work on formal methods is intended to make major advances in the way software practice meets these demands.

With few exceptions, the old techniques—and the tools that support them, have aimed to help the programmer to describe, encode, recode, package and probe his large and complex product. The task of fully understanding the product and of ensuring that it and all the steps used in its production are correct, has been left to the human judgement of producers and consumers. With some serious exceptions, present practice has been generally successful due to the dedication of computer professionals, the effectiveness of social processes in dealing with errors in output data, and, of course, the great economic benefit of using computers for solving problems. This success has come at high costs that are no longer acceptable. Furthermore, new applications and market conditions have arisen that bring very high penalties for error, inflexibility and inefficiency.

The response of the computer science community has been to attempt to put mathematical rigor into the task of understanding what a program does (its "semantics"), or, in linguistic terms, what is the meaning of its text.

Of course, understanding precisely what a program does implies knowing precisely also what the translators, optimizers and interpreters do that may operate on the program. If the description and analysis of programs and their processors could be put on a firm mathematical basis, and if there were effective procedures for carrying out formal analyses, the consequences would be very significant. For example, one could build powerful optimizers, and be confident...
that they would not change the meaning of a program. Also, one could verify that a program implemented a requirement for all possible values of data in the intended input domain, without having to use test data samples (a most costly and unsatisfactory process if high confidence is demanded).

As long as ten years ago, theoretical results showed that a formal basis for programming could be constructed. Since then much effort has gone into creating methods of formal specification and analysis, and tools to implement the methods, that are effective enough to be incorporated in software engineering practice. At the present time, several methodologies are reported to be in practical use in special application domains such as microprogramming, or on an experimental basis, for general programming. The methodologies require skilled practitioners. Because of this, one cannot say that a methodology is "ready" or "not ready" without saying who is going to use it. Nevertheless, it is claimed that there will be enough practitioners of the requisite skills to make a very significant economic impact on software production, especially as the methodologies mature.

The papers and panel sessions of this area give the attendees an opportunity to judge how valid these claims are, and to assess the relevance of the ideas and tools of the new methodologies to their own practice.