IFIP participants debate programming approaches

Galen Gruman, Soft News Editor

The 11th World Computer Congress focused on providing tools — whether in the form of theory or tangible products — to computer researchers and developers. The International Federation for Information Processing's triennial conference was held Aug. 28-Sept. 1 in San Francisco and drew about 1,900 participants. The 11 tracks included one on software engineering and several with a substantive software component.

Balancing theory and practice. In the academic keynote address, Donald Knuth of Stanford University urged computing professionals to balance theory and practice. "Theory and practice are not mutually exclusive," he said, "They are intimately related. Neither theory nor practice is helpful without the other. The best way to improve our tools is to blend theory and practice."

Knuth decried the recent emphasis on funding for mission-oriented research. "This ignores the fact that the basic theory needs to be developed," he said, "We need a lot of small projects instead of a few large projects with predefined goals. Otherwise, we'll face a huge slump in our ability to work with future problems."

As an example of how theory and practice can be combined for high payoff, Knuth challenged the audience to make a thorough analysis of everything — the hundreds of thousands of operations — a computer does in one, random second. "I'm sure the results will be very interesting," he said, "and will help us to learn to make better computers."

Advocating standardization. In the industrial keynote address, John Young, president of Hewlett-Packard, advocated the broad use of standards and argued that they would enhance competition and innovation, not hinder it as some have argued, because they would let developers focus on better implementations rather than reinventing fundamentals like protocols and interfaces.

"They'll lead to a new richness," Young said, because they "define physical and logical connections among services, but they don't define *how* those services are implemented." He cited the automobile industry as an example of how standards let developers focus on innovation and implementation to the customer's benefit.

Young said the lack of standards has actually hindered progress by making people focus on low-level details like data formats and learning a different interface for each program. "White-collar productivity in the US has actually declined, despite rapid developments in computer technologies and massive investments in them. The service industries that have invested most in computer technology show the worst productivity performance," he said, citing several studies.

The recent emphasis on mission-oriented research ignores the fact that the basic theory needs to be developed, Knuth said.

"This [problem] requires a cooperative computing environment made of heterogeneous machines. This requires standards for how they communicate," Young said. However, he cautioned that standardization should be limited to

- be independent of existing systems,
- only what is truly generic and had broad application, and
- mature technologies.

He said standards — even privately developed ones — should be open, with reasonable licensing fees and equal access.

In his closing address, Bill Joy, Sun Microsystem's vice president of research and development and the chief architect of BSD Unix, echoed the call for standardization, saying that openness is real now that the industry has reached the "end of feudalism," where each company is self-contained and follows its own rules, as villages did during feudal times. However, he cautioned that when standardization "is taken to the limit, the result becomes politics where the last two or three viable alternatives battle to be the dominant one. That's a destructive thing. I think we can afford to support more than one [alternative]."

Reliability. A major thread throughout the software-engineering sessions was whether developers could produce reliable programs whose reliability they could have measurable confidence in. System designers can now evaluate their confidence of whether a system would fail during 10,000 hours of use, said Bev Littlewood of the City University in London. But designers cannot be confident that safety-critical systems specified at reliability levels of one failure per billion hours meet those requirements, he said. While formal verification and fault tolerance may help, they cannot achieve confidence at that level of reliability, he said. "Perhaps we should just build systems that require only modest reliability," he said.

Littlewood's assertion caused several panelists and members of the audience in that and later sessions to debate the appropriateness of numbers like one fail-

ure in 1 billion hours of use because no one could test for so long. Several people said such requirements were misleading because they confused real time with execution time. John Musa of AT&T Bell Laboratories cited the *Voyager2* mission, which in its 12 years required less than 10 minutes of execution time for its reliability-critical navigation software. For software with long execution times, using faster machines or multiple machines would also reduce testing time without compromising test coverage, he said.

Several people advocated specific approaches to building reliable software, including *n*-version programming, applying formal methods, large-coverage testing, fault tolerance, and verification and validation. (In the *n*-version programming approach, independent teams build their own versions of the program with the hope that each team's errors are different; you create the final version by merging the error-free parts of the initial versions.)

In a presentation, John Knight of the Software Productivity Consortium argued that the hardware reliability modeling that underlies *n*-version programming does not apply to software. "You could have faults even though there are no faults [in any of the versions] because the versions don't agree," he said. His presentation provoked much criticism from several members of the audience.

In a separate presentation, Knight also criticized fault tolerance. "The real question is 'Will [faults] be tolerated when we want them to be?' " he said. He concluded that "we can't avoid them, and we can't eliminate them with any degree of confidence, either."

Decrying narrow focuses on the effectiveness of specific techniques to make software reliable, David Parnas of Queens University listed the overly narrow focuses of previous presentations and panels and showed how they argued about "false dichotomies," not addressing the fundamental issues of reliability. "Can we write fault-free software? Probably, but we can't tell if we do. Can we count on it? Sure, if we're crazy," he said, "Can we build fault-tolerant software? Yes, if you give me a list of faults to tolerate."

"Is it better to verify software or to test it?" Parnas asked. "Yes. [The audience laughed.] Proving eliminates a certain class [of error], testing another class — they're not entirely disjoint. We should do both." He also said the debate over *n*-version software missed the point. While people should write correct code the first time, "people make mistakes, even with the best methods," Parnas said. In some cases, *n*-version programming works; in others, it can make things worse, he said.

Foundations. Ultimately, software engineers need a solid foundation on which to base their methods and approaches, panelists agreed. "Some foundations we have but are not being used very well. Others need to be built," said Harlan Mills of the University of Florida and the Information Systems Institute. "We need good mathematical and statistical foundations," he said.

But Mills reminded the audience that "software engineering is just one human generation old. When civil engineering was that old, they didn't have even a right triangle!" Still, Mills complained that it is too easy to be a software engineer: "[Just] take a three-day short course — with no exams," he said.

"You *can* prove correctness, even in 10-million-line systems," Mills argued, by scaling up mathematical methods of correctness. "Verification is finite, looking for bugs is not," he said.

Susan Gerhart of MCC's Software Technology Program urged developers to consider how the foundations of computer science fit in with the foundations of the application domains. Computer science's contributions include semantics, fundamental structures (like formal methods, invariance, and abstract functions), theories of how computers work, transformation methods, and process organization. "We have the general forms of the foundations, but a lot of the details are missing," she said. Most systems today are based on the foundation of the domain, not of computer science, Gerhart said.

"We're doing much too little in understanding how to model the problem domain before we get into the requirements specification and programming," said Dines Bjørner of Danish Datamatics Center and the Technical University of Denmark, "We don't know how to relate them to the software aspects. We need to provide the foundations that interrelate them."