ICSE assesses the state of software engineering

Galen Gruman, Soft News Editor

Software costs mount. Schedules slip. There seem to be more bugs delivered than functionality. Researchers and developers seem far apart. Government agencies are counting on software for critical tasks, so workshops are held to address the software crisis. 1989? No, it was 1968 and 1969 — these issues led to the seminal NATO workshops in West Germany. Yet 20 years later, at this year's 11th International Conference on Software Engineering, the same issues remained.

This year's conference, held May 11-14 in Pittsburgh, combined a retrospective on the NATO workshops — where the term "software engineering" gained legitimacy as a goal if not a reality — and a look at unsolved problems. The basic question was "What is software engineering?"

"I think we've made remarkable progress" since the NATO workshops, said John Buxton, a computer-science professor at the University of London and a member of the NATO workshops' organizing committee. Those successes include development of the software life cycle, quality assurance, quality circles, design tool sets, and design methods.

But "we've made very little progress in some areas," said Larry Druffel, director of the Software Engineering Institute and conference chairman. "Our legal system doesn't protect software. There are no standards for entry into the profession. There is no shared standard of ethics. We have not firmly established software engineering in its important role of systems engineering. We're professionals trying to identify our profession," he said.

The three days of paper sessions and panel discussions produced no magic answers for the 1,200 people attending.

What is the field? During the software crisis of the mid-1960s, "we coined the words 'software engineering' and 'computer science' as wishful thinking," said Jack Munson, the Unisys manager of all software support for the space shuttle. "Academia should help put more science in computer science and more engineering in software engineering," he said.

Despite this widely acknowledged need, merely realizing that software engineering is different than programming has helped the field progress greatly, said Bernard Galler, a computer-science professor at the University of Michigan. But "in my department, software engineering still has to be accepted as a discipline, as a subject for theses. We've made progress, but unfortunately we're still talking about many of the same problems," he said.

The field should simply be named "computing," Buxton said. Computing has various facets, including the relationships to mathematics (computer science), to electronics (hardware design and architecture), and to the rest of the world (software engineering), he said. "Software engineering is the principal user interface to computing. It's not a separate discipline," Buxton said.

Formalism's role. An undercurrent throughout the conference was the role that formalism should — or should not — play. "We shy away from formalism. Most introductory texts do not teach programming principles," complained David Gries, a computer-science professor at Cornell University. "We ask for no kind of precision from the programmers," he said. "The emphasis is on studying the logic — predicate calculus — not on using it," he said. "We should pay more attention to formal concepts. We should teach it much earlier — [although] not necessarily so deep at first," Gries said. "We shouldn't accept a contract where it doesn't make sense to us. We should apply more rigor," he said.

Gries acknowledged that formal methods "require a change of habit, a change of thought." But he urged people to adopt formalism and "invest in the time over the long term, not the short term."

Many people disagreed with a strong emphasis on formalism. "Intuition and guessing will take you a lot further than formal methods. Software is as ornery as the human systems we are used to dealing with," said Nicholas Zvegintzov, a consultant and *Software Maintenance News*'s editor.

Others suggested a considered application of formal methods. "I think that technical correctness depends greatly on using the language that most directly says what you want to say," said Michael Jackson, a consultant whose Jackson Structured Programming method is widely used. "Correctness and formality have to be fitted to the application at hand," he said.

State of the practice. Most speakers acknowledged that software engineering is not yet a true engineering discipline. "One sign of our immaturity is our willingness to be sold panaceas. Remember that Cobol was supposed to eliminate programmers," Jackson said. Another panacea, data modeling, "is a catastrophe," he said. "The structures of the problem and the solution are almost always going to be very different," he said, "so we convince ourselves that they're the same rather than seek new models."

But software engineering can advance despite this immaturity, he said. "We need to specialize. There are no 'physical engineers,' but there are automotive, civil, and electrical engineers. We need the same for software," Jackson said, although he admitted that it is hard to determine what the specializations must be. "If you think about software design like marketing, you realize there's a need for many different formalisms." By considering software as a manufacturing process with parts, materials, tools, operations, and methods, "we can advance [development] tools and methods," he said.

"We may as well abandon the dream of getting the whole under control, Zvegintzov countered, "Various methods will work for localized problems. You will always be working on parts of the system."

Regardless of eventual scope of software-development tools, "there are

good, hard technical areas to work on" now, said Susan Gerhart, a senior technical staff member at the Microelectronics and Computer Technology Corp. These areas include transformations, higher level languages, delta theories to give people a technical handle on change, high-performance verification and testing, and laws of software structure.

The slow advances in software are not the fault of programmers but are due to outside factors, argued consultant Tom DeMarco. He cited Xerox, AT&T, and IBM, which were sued by the US government over antitrust concerns and forced to give up computer-related patents or license their computing technology to competitors. "Now don't expect much [from Xerox] for the next 10 years, since it is no longer funding fundamentals like Xerox PARC," DeMarco said. The results of the antitrust policy were to remove the incentive for US firms to develop computing technology and to make this technology available to competitors, especially the Japanese in the hardware area, he said. This policy weakened "the three giants of the information age," he said, "Nothing that happened in the software community had the effect that this has."

But there are "shining lights," DeMarco said. "MCC and SEI seem to have a hope of getting [the US] back into the research business," he said. The technical successes of small companies like Sun and Cray are encouraging, he said, but "we've got to build a software infrastructure — we can't hope for brand-new start-ups like Next and Sun to do this regularly."

But it's not all bad news: "A lot of progress that has taken place is not very visible," said Mary Shaw, a computer-science professor at Carnegie Mellon University. "We know that there are three types of software-engineering decisions: design, which we cannot quantify; empirically quantifiable, which we can count; and structurally quantifiable, which we [understand]. The last has the most long-term potential."

Technology transfer. Problems with the state of the practice have as much to do with the lack of technology transfer as with the lack of a solid science and engineering foundation, several speakers said. "We have been least effective in teaching what we already know, Cornell's Gries said.

Universities are not alone in not teaching what is known: "Lurking within companies are breakthroughs that don't transition from project to project," said Ilene Brikwood, vice president for quality at Tandem Computer. Why? "The problem is that you can't rock the boat. But to improve, you've got to change what you're doing," said Al Pietrasanta, a consultant.

To change the way the do things at Tandem, new methods and technolgies are prototyped on a single project before being disseminated, Brikwood said. This gives other projects confidence about the new technology's usefulness, she said, since developers "have got to see their neighbor using it successfully. No one believes anything from another company or from research," she said.

Researchers versus developers. Les Belady, MCC's director, told one panel that "research and development are nonconnecting, stable subcultures. What can we do about it? We need to find something else that focuses on transfer." He cited the Software Engineering Institute as a successful transfer endeavor, but "we need more."

"These [researcher and developer] cultures tend to be antagonistic to one another," Pietrasanta agreed, "That limits progress." While most visible between academia and industry, this cultural split also occurs between corporate research and development staffs, the panelists noted. "But both groups have common needs and need each other to achieve these outcomes. We need help from anthropologists, sociologists, and psychologists on getting the two cultures together," Pietrasanta said.

"The issue is that the researcher has to focus on software product and process. The development organization has the goal of getting that product out the door," said Victor Basili, a computer-science professor at the University of Maryland. Furthermore, "research is heading to high levels, but the development community is struggling to get out of bottom level," Pietrasanta said, "We've got a mismatch."

One way to address this mismatch would be to promote application-specific research in academia. "If you do software-engineering research, you have to look at the artifact. You have to get in there; you can't do that in isolation in a lab, no more than an experimental physicist can without applying the laws of nature," Basili said. "My speculation is that application-specific research will have tremendous pay-off," said Bob Martin, vice president for software technology and systems at Bellcore. "But research people are not willing to become an expert in a domain to make a contribution with a high pay-off in our field," he said.

"Can [developers] learn that experimentation is a normal process?" Basili asked. "The problem with developers as experimentation vehicles is that they have short-term constraints. They don't want to be a playpen," Pietrasanta replied.

CASE tools were a commonly cited example of the researcher/developer split between academia and industry. "Very few of the CASE tools come from the academic research community, so you have many CASE tools without computer science behind them," MCC's Gerhart said. "There's a lot of 'computer-aided' and not a lot of 'software engineering' in the term 'CASE,' " Martin agreed.

"The investment in [CASE] tools is tremendous," Unisys's Munson said, but "you will get nothing if all that you have to use is a new set of tools and methods. We should not convert the Tower of Babel of languages into a Tower of Babel of CASE tools."

However, "the act of education to use them is a huge benefit," Tandem's Brikwood argued. At her company, those learning CASE tools "had to think in a new way and take training they wouldn't have taken before. Plus, graphics made communication across sites better since no one actually reads the specifications," she said. "I think the emphasis on CASE tools is healthy, not because the tools of today are in any way appropriate but because they will lead us to appropriate tools," consultant Jackson said.

Management. "How do you handle projects so big that no one can understand them?" the University of Michigan's Galler asked, "Programming is one thing. Managing the process is another."

"In software, we don't control our own environment," Brikwood said. "There are so many areas that have zilch to do with software that affect us," she said. Brikwood would like "to know what effects that changes have on the schedule, ... to have estimation and measurement tools tied to design tools, plus tied into performance tools on the total distributed system, plus all tied in to the marketing database so you can see what effect that will have on the marketplace, plus a quality system that takes measurements across the project from beginning to end — measurements everywhere. We have tons of data, but we don't have good information," she said. The University of Maryland's Basili cautioned, "You have to do measurement to know what's going on. But the project [you measured] has to be willing to change" when the results are in.

A key to managing software development is understanding how develop-

ment itself works, said Bill Curtis of MCC's Software Technology Program. MCC has studied how developers develop, and Curtis presented the findings to a plenary session. "People feel guilty because they have a model of they feel they ought to be working that doesn't match the environment of the projects they work on," he said.

"Design is a process of satisfying requests within constraints. These include constraints imposed by the environment under which you are developing," but the environmental effects are often overlooked, he said. "A process model may not match the way people behave. It's points of accountability, not an imposed routine," Curtis said.

The MCC study showed that a developer's focus is "bouncing back between levels, not the [traditional linear] analysis of requirements to decomposition to implementation. Insights in the solution level depend on going back to the problem level," Curtis said. Therefore, "you have to have tools that let you move across levels of abstraction easily," Curtis said. The traditional top-down model is merely "a special case for when you already know the solution," he said.

The study found that "gurus have the deepest, richest understanding of the application domain — objects, constraints, behaviors — and are more able to translate this to computational architectures. They have an understanding of what you are writing a program about. They can [do] a sophisticated mapping," Curtis said. This bolsters the case for seeking or developing domain specialists, he said.

Selecting developers for a project team requires more than accumulating gurus or team players. "Team building is not [the same as building] a consensus," Curtis said. For example, "if [only] a few agree, they can lead the project. But other team members challenge assumptions and encourage explanation of alternatives," he said. Managers must make a trade-off when selecting team members: "Little diversity equals rapid design," Curtis said, while "great diversity equals thorough exploration."

Unfortunately, when a project is successful, its successes are rarely transferred to other projects, Curtis said. For example, "[design] rationales are not transferred from team to team," he said. Furthermore, "organizational buffers restrict the ability to make design decisions," Curtis said. For example, "scenarios of product use are not transmitted to programmers," he said. Programmers may not thus understand why some needs were important to design decisions since "you can't use information effectively until you understand the domain," he said.

Dealing with change is a perpetual problem for managers and developers alike. But people must accept that, "for large systems, change is a normal process — technology advances, needs change," Curtis said. "In the real world, the specification process is a negotiation process. I want to satisfy the customer's needs, not the specification. A formal specification is the beginning of the negotiation process," he said, not the final step. "The customer is learning at about the same pace as the developer is. That's why the specification is only a best guess at the time you wrote it," Curtis said.

The study also found that "most people are doing other things than writing programs," he said. These include learning, forming a consensus, negotiating, communicating, coordinating, and socializing, Curtis said. "These behaviors are very important — 70 percent of a programmer's time can be spent in communication," he said. But software-development textbooks leave out these important functions and so management does not recognize their importance, he said.