Engineering Automation for Computer Based Systems

Luqi

Elsevier

Electronic Notes in Theoretical Computer Science 25 (1999)
URL: `http://www.elsevier.nl/locate/entcs/volume25.html` 9 pages

Engineering Automation for Computer Based Systems [⋆]

Luqi

Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

1 Introduction

Software development capabilities lag far behind society's demands for better, cheaper, more reliable software. Since the gap is so large, and widening, it is unlikely that "business as usual" will be able to meet this need. Engineering automation based on sound and scientific methods appears to be our best chance to close the gap.

This is the sixth in a series of workshops whose common goal is helping to increase the practical impact of formal methods in software development. These workshops have succeeded in gradually bringing the theoretical and practical sides of the software engineering community closer together, focusing them on fulfilling the promise of scientific improvement of software engineering practice. The progress made in this direction at this workshop was larger and more readily apparent than in previous years, giving us hope that the effort will eventually succeed.

The remainder of this paper is organized as follows. Section 2 restates the main premises of the workshop. Section 3 gives an overview of the papers. Section 4 summarizes some of the discussion at the workshop, and Section 5 presents some conclusions.

2 Premises of the Workshop

The main premises of the workshop are that mathematics and formal methods can help solve practical problems in the engineering of computer-based systems, and that engineering automation is a promising way to accomplish this.

[⋆] This research was supported by ARO(MIPR8GNPSAR042), NSF(CCR-9813820), ONR(N0001499WR20019), SPAWAR(N6600198WR00438).

We use a broad definition of "formal method." Webster's Dictionary says that *formal* means definite, orderly, and methodical; that *method* means a regular, orderly, and definite procedure; and that *model* is a preliminary representation that serves as a plan from which the final, usually larger, object is to be constructed. Thus, to be formal does not necessarily require the use of logic, or even of mathematics.

In computer science, the phrase *formal method* has taken on a narrower meaning, referring to the use of a formal notation to represent system models during program development. An even narrower sense refers to use of a formal logic to express system specifications, and proofs to check correctness of implementation code - i.e., that it satisfies the specification.

The broader definition of formal method is appropriate to this workshop because it fits the theme of engineering automation. Processes need to be definite, orderly, and methodical to be successfully and reliably automated. Thus, formalization of engineering processes in this broad sense is a prerequisite for engineering automation.

The narrower sense of formal method - checking whether or not the code satisfies a particular requirement specification in a formal logic - is inappropriate for this purpose, because of the well known fact that the majority of software defects are requirements errors (see the paper by Berry in this Proceedings). If the specification is wrong, we do not want code that satisfies the specification.

The broader interpretation of formal method opens the door to other approaches, such as requirements elicitation via prototyping and the automatic synthesis of correct code from requirements models formulated via domain-specific notations. Note that a formal model is required to generate an executable version of a prototype, and practical prototyping requires extensive automation of the prototype design, analysis and implementation process. Such tools depend on extensive formalization of the processes involved. Similarly, the design of a domain-specific program generator depends on extensive domain analysis, culminating in the formalization of problem domain concepts, corresponding problem specification notations, and a library of solution methods for each domain. All of these activities are formal methods in the broad sense.

The reader is cautioned that not all of the authors use the phrase *formal method* in the broader sense recommended here. For example, Berry states that formal methods do not help in identifying requirements. This is true under the narrower interpretation of the phrase, but not necessarily the broader one.

3 Overview of the Papers

Several concept papers assess the applicability of formal methods to engineering practice. Berry notes that formal methods must be cost effective to be of practical use, that requirements are the central practical issue, and that most formal methods do not help to identify requirements. He also conjectures that formal methods help when they do because they provide a second iteration on conceptual formalization. Robertson analyzes observed failures of formal methods and their causes.

Another group of papers addresses automated reasoning and analysis. Bjorner presents a decision procedure for queues. Manna, Sipma and Uribe describe a method for combining deductive inference and model checking that can provide proofs about infinite state systems using algorithmic finite state methods. Cleaveland and Sims present methods to improve the efficiency of generic, automatically generated model checkers. Narasimba, Cleaveland and Iyer present a model, logic, semantics, and model-checking procedure for probabilistic systems. Kwak, Lee, and Sokolsky give a method for symbolic schedulability analysis that links to efficient equation solvers, which could be used to synthesize designs by solving for values of design parameters that would make the design achieve schedulability guarantees. Berzins analyzes the inference requirements for engineering automation and identifies the need for lightweight inference methods: sound, very efficient, typically restricted or incomplete.

A third group of papers report on engineering aspects and practical experiences in the application of formal methods. Polak reports a successful application of automatic program synthesis in a specialized domain (satellite control systems), and analyzes the reasons for the project's success. Kosiuczenko and Wirsing formalize a common design notation for communication among distributed systems (message sequence charts) using timed rewrite logic, and use the formalism to test a specification by executing it, revealing a fault. Gelfond and Watson describe the application of logic programs with non-monotonic semantics to realize automated decision support for a complex domain (space shuttle operation in the presence of multiple equipment failures). Volker and Kraemer describe the successful application of the higher order logic HOL to the development of a verified library of function blocks for a safety-critical domain (industrial control). Gafni, Feldman and Yehudai present a real-time design language for large scale applications and explain the associated design process via an example (cruise control). Cooke describes a formalism for expressing implicit concurrency in data parallel computation, with applications to data mining. Zhang, Lee, Friedel, and Keyser describe statistical methods for generating facts from raw data to provide decision support for an engineering task (diagnosis and repair of phased array antennas).

Peter Wegner presented the idea that interactive systems fundamentally change the nature of computing, and that this change has far-reaching effects that have not been fully integrated into current theories of computing and engineering science. The main ideas are summarized here because there is no corresponding paper in the proceedings (for more details, see Mathematical Models of Interactive Computing, http://www.cs.brown.edu/people/pw).

The difference is that the input to an interactive machine is not fixed in advance, and could depend on the partial output produced by the machine up to that point. A difference in expressive power due to this effect is claimed. This view of computation leads to different kinds of formal models, such as co-algebras; and different modes of reasoning, such as co-induction, which are relevant to the analysis of open (extensible) systems of the kind common in the current practice of object oriented design. The proposed change in viewpoint stimulated discussion as well as some controversy about the details and their philosophical interpretation.

4 Summary of the Discussions

The National Science Foundation is considering the impact of the PTAC report (http://www.ccic.gov/ac) and its impact on national research priorities, as summarized below. The report's major recommendation was to make software research an absolute priority. The four major research priorities identified are:

(i) Software

(ii) Scalable information infrastructure (networking)

(iii) High performance (peta-flops) computing, including software R & D

(iv) Socio-economic and workforce impacts

The report finds that software demand exceeds the nation's capability to produce it, that we must still depend on fragile software, that technologies to build reliable and secure software are inadequate, and that the nation is under-investing in fundamental software research.

The report makes the following recommendations:

(i) Fund fundamental research in software development methods and component technology;

(ii) Sponsor a national library of software components in subject domains;

(iii) Make software research a substantive component of every major IT research initiative; and

(iv) Fund fundamental research in human/computer interfaces and interactions.

Relevant research initiatives include ASCI (Accelerated Strategic Computing Initiative) and NGI (Next Generation Internet). The internet is making the next step, with major implications for software research. Yesterday's environment is not tomorrow's, and many issues need rethinking within the future context.

We are at a unique point in IT history: agendas are being set and recommendations are being made. The field needs a research agenda, a plan for research management, and action to build public support. Consequences of not acting include negative economic impact and loss of global leadership and competitiveness. One issue is that we are not currently able to meet the demand for software. We therefore need to:

(i) empower end-users with domain-specific tools that create software;

(ii) make component-based development a reality;

(iii) automate software engineering processes; and

(iv) produce more well-trained professionals.

Another issue is that we cannot produce high-confidence systems, and cannot even produce routine systems routinely. We therefore need to:

(i) understand what works and what does not;

(ii) understand the science of software construction; and

(iii) create a discipline of software engineering.

The problems identified in the PITAC report have many facets, including unresolved practical problems, rapid change, immaturity of the science, a gap between theory and practice, fragmentation of the research community, and inadequate infrastructure for technology transfer.

The recurring horror story is that we can not afford to build software systems using current technology. This has been true for many years despite improvements in the state of practice. We have not made a convincing case that we have done much. Some of the reasons for this are increasing demand and rapid change, lack of effective technology transfer, and lack of the right kind of science.

The practice of software engineering is moving very fast, in an attempt to keep up with demand and stay ahead of the intense competition. Time to market is vital in the commercial world. Many developers jump on aggressively marketed software fashions, although they often include ad hoc methods and worst practices along with some improvements.

Despite these difficulties, the commercial world has made progress. For example, Java is an improvement over previous practice. Networking and communication are coming together, and succeeding in reusing resources. Commercial systems engineering is improving. We can successfully educate professionals in about ten years.

Other commercial steps have been less effective. UML had the benefit of lots of talent with inconclusive results. The semantics of C++ remains controversial. Component technology is in fashion although it is still difficult to make components work together.

There is a widespread attitude in the commercial world that academic results are impractical and that theoretical results take too much time and cost to incorporate into practice, especially in a highly competitive world. Some parts of the theoretical computing community take the attitude that practical engineering is irrelevant. The result is ineffective technology transfer and engineering practice with a weak scientific basis.

This is an area where improvement is possible. Instead of a struggle between theory and practice, there should be a supply chain, and a coherent vision of problems flowing up the supply chain and solutions flowing down the supply chain. This should be a continuous, orderly, and effective process. Currently, it is not. We can not afford change in random directions.

There are multiple causes for the current situation, including immaturity of the discipline. The problem goes deeper than a lack of communication that could be resolved by the current practices of our educational systems. Many issues that arise in engineering practice have not been addressed by the scientific community. There is growing awareness of these issues and increasing resolve in the scientific community to address them by developing a more robust and principled basis for future software engineering technologies.

Past emphasis on formal methods in response to this problem has been a mistake. We should instead speak of and insist on effective, rational methods to achieve goals. The Latin for *method* is "via ratio," a rational path. It is not convincing to say, "We are on the right side because math and formulas are what matters." A shift of paradigm is now needed. The quality of the result and the cost of producing that result are what matter. For progress in engineering, it is essential to automate the process. The solution must be a highly interactive, adaptive, automated system. We must admit that, even if we build an advanced system, it will be at a cost of not doing it again.

As science is currently inadequate to support automated engineering, our community needs to understand and develop the science needed to bring the engineering to this level. Formalization is useful to the degree that it contributes to this goal by enabling automation or systematization of engineering processes.

There are two kinds of science: theoretical science focuses on understanding and prediction, while engineering science focuses on empirical validation of theory-based predictions, and learns mostly from failures - as, for example, in seismology. A finer interplay between mathematics and empirical science is needed to achieve progress. Many good ideas have been proposed, but often without a plan to evaluate success. The only basis for rational judgement is empirical science. Many ideas that sound good in the abstract can not be realized in practice. Good empirical computer science is needed, but no one has been able to do it well so far.

To focus effort where it is needed, it may be useful to distinguish engineering science from theoretical science. Recognition of the category *engineering science* is important because research funding agencies typically support science rather than engineering. The aim of engineering science is to improve the capabilities of practicing engineers. The aim of theoretical computing science is to improve our understanding of computing. Automation is a primary goal for engineering science, but not necessarily for theoretical computing science.

Advances in theoretical computing science can contribute in the long term to software engineering by providing better conceptual models and better principles that can be used to build tools for engineers. However, significant effort is required to identify, reformulate, extend, and package the relevant results from theoretical computer science to make them useful for engineering. For example, theoretical advances are often made using simplified models that avoid issues and details that are inescapable in practical engineering. These issues are in the realm of engineering science, and are vital for progress.

We need technology transfer from relevant new engineering science to make things work. Nobody has the responsibility for this now. There should be an "Expedition Center" to envision what the world is going to be like in 100 years, and a "Transfer Center" to transform those visions into reality. We have to be careful about what kind of technology we transfer: it must be relevant to practical problems. There is much irrelevant material from former type theoretical computer science and others: e.g., How do you get a theorem to find oil?

The various parts of the community must interact more closely than they have in the past to achieve practical impact. Software isolation is a problem. Much software is connected to communication, hardware, and other components. If we do not include these components, we have not solved the problem. Results from other disciplines are relevant also. Software development is a special case of product development. Software is hard because it is abstract; it cannot be visualized. We can learn much from design theory and product management.

Rapid change affects the scientific community as well. The nature of computing may change substantially in the 21-st century. For example, new models of interactive computing and quantum computing are on the horizon. Today's computing environments can not and will not be the environments of tomorrow. Computing is a relatively new science. There is opportunity, but also a need to educate people about what computer science is and what it can be. There is also need for periodic reality checks to ensure feasibility of long-term visions. These exercises can help improve the credibility of our field, can provide course corrections for research agendas and can evaluate readiness for technology transfer as we learn more about what can be done at what cost. DARPA and other agencies have challenge problems that could serve these functions. For example, the automatic theorem proving community has a standard set of benchmark theorems.

There is a tension between long-term goals and short-term goals. Funding agencies require that goals be achieved on a yearly basis. This is an issue that must be faced by all branches of science, not just in computing. We can ask how the issue is handled in other disciplines, such as particle physics. Physics has a history of setting up visionary programs. In Italy, 72 percent of money for basic science goes to physics, and only 1.7 percent goes to information science. Why is this - a good part of the answer is that the physics community behaves in a political way, i.e., it has a lobby. They say, "We have this great vision. We need Congressional funding for astrophysics, etc.", and then set up a lobby and get real money. We need to develop a similar vision and agree to work together toward that vision.

In computing science, we have not agreed on the goals. This has been aggravated by the rapid rate of change, which has spawned computing schools of thought, and intense competition for scarce research support. We need to identify our goals and stick together, instead of "dissecting ourselves to death." Computing research does not have to be a zero sum game. The goals identified in PITAC report are a good starting point for developing a shared agenda for the entire computing community.

Computing is the most successful technical discipline, in that it has come to relevance and has been applied in a relatively short time. Decidability and computability ideas appeared only at the beginning of this century. We had a vision of software engineering in 1968, but people were not aware of how much is hidden behind that vision. The digital point of view brought in a whole new view of the world, as opposed to physics. There is a basic difference between the root of physics and the root of computer science. The foundations of computer science are very simple - i.e., Turing machines suffice, with some modifications. NP completeness is not the most central problem. The real problems come on the macro level, in building systems and with human factors. The roots of physics are different, more involved. The theory of digital models may become much more than it is today.

We should be happy to work in a scientific field that has such a high level impact. We should also understand that there is a real push in progress, and appreciate that scientific push. What we have done wrong is to engage in too much infighting, much of which is due to not understanding the inherant positions imposed by the disciplines of our colleagues. What we have gained over the last 20 years could not have been done without deeper understanding. What we actually do in practice is not called *formal methods*, yet we have made more progress than we realize. It is important to make the field more transparent. We are just at the beginning.

5 Conclusions

The technical presentations and the engineering experiences reported at the workshop support the premise that engineering automation can lead to significant practical gains. Some of the papers detail the circumstances under which such gains can be realized using currently known techniques, thus providing a snapshot of the current state of the art in the area.

Another outcome of the workshop was a change in the attitude of the participants. For the first time there appeared a broad consensus that we should work together and agree on a larger common vision that we can all contribute to from our individual specialties. Most participants accepted the idea that theoretical work should contribute to engineering over a medium- to long-term time horizon. A working approximation to that vision is the improvement and application of computing science to, in turn, improve and automate processes for developing reliable computer-based systems.

This consensus suggests a direction for action. The common vision needs to be supported by a more detailed research and development plan, providing explicit intermediate goals on the way toward the ultimate end. We should interleave our specialized scientific efforts with periodic application and integration of results from our different disciplines, with assessment steps and identification of unsolved problems that lie between the solved fragments, and with validation and adjustment of the assumptions used as the basis for the next round of basic research. Applications of new and sometimes deep theories rarely happen spontaneously. For best success, those researchers who originate new theories should spend part of their effort identifying and developing applications of those theories, perhaps in cooperation with groups whose primary focus is empirical engineering science. Some of our most valuable lessons have come from the analysis of failed attempts to apply existing theories.

We must work together to agree on how these threads will fit together into a coherent whole, and to form a more detailed vision that addresses society's long-term needs. Technology transfer and public relations are part of this puzzle. We need to better communicate to the public how engineering automation and the basic research it will take to achieve that goal will alleviate the difficulties associated with computer-based systems that currently touch all of our lives. We need to make concrete progress in this direction, and to demonstrate the practical impact of that progress in a systematic and coordinated way. It is important to put past disagreements behind us to work together for the common good of both the computing discipline and society at large.