Software Engineering Beyond our Planning Horizon: Automation for Computer-Based Systems

Luqi


*Downloaded from NPS Archive: Calhoun*
Software engineering beyond our planning horizon:
automation for computer-based systems

Luqia, Manfred Broy

Abstract

Software development capabilities lag far behind society’s demands for better, cheaper, more reliable software. Software engineering being originally very much restricted to the idea of programming develops more and more into a universal discipline of systems engineering. We give a general introduction to a software engineering workshop dealing with mathematics and formal methods that help solve practical problems in the engineering of computer based systems and engineering automation. Some of its 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.

1. Introduction

Software construction develops continually into a key technology for engineering complex technical systems of the 21st century. More and more critical infrastructures are crucially depending on the reliable functioning of software. Over the past few years, we have witnessed a slow but steady decrease in the gap between the theoretical and practical sides of the software engineering community. We are confident that this trend will continue and will accelerate improvements in the state of software engineering practice and theory.

Software problems have been quite visible to the public due to spectacular disasters in space missions or telephone black outs and are receiving increasing attention with the recent Y2K deadline. It is a good time to demonstrate concrete improvements in our discipline.

This research was supported by ARO (MIPR8GNPSAR042), NSF (CCR-9813820), ONR (N0001499WR20019), SPAWAR (N6600198WR00438), The Army Research Office, National Science Foundation, Office of Naval Research, and the Defense Advanced Research Projects Agency sponsored the 1998 Monterey Workshop on Engineering Automation for computer based systems.

* Corresponding author. Tel.: +49-89-289-28161; fax: +49-89-289-28183.
E-mail address: broy@in.tum.de (M. Broy).
The continued doubling of computing speed and memory capacity every 18 months and the improvement in the capacity of communication links implies that the only constancy for large distributed systems, technology, tactics and doctrine may well be the idea that change is always inevitable. The dynamic aspect of systems is not supported by current practice and is seldom emphasized in current research. Software evolution research is extremely important for achieving modifiable and dependable systems in the future. Improved methods for reengineering are also needed to bring legacy systems to the condition where they can benefit from improvements in software evolution technology.

Thirty years ago, when the term software engineering was coined, there was lack of theoretical foundation for many practical concepts in computing. That is no longer true. A solid body of foundational work is available now that addresses many challenging issues related to software and computing, including specification techniques for systems and data, logical calculi for concurrent, distributed, and realtime systems, logical concepts related to interactive systems, and formal models of programming language semantics with a variety of inference systems.

The challenge is to put these results to work, to develop theory that better supports engineering needs, and to improve practice. This will require cooperation and a concerted effort from both theoreticians and practitioners. We will need advances in education and improvements in theoretical approaches to meet the increasing demand of practical engineering for computer software. To be attractive to practitioners, formal methods, mathematical foundations and automated engineering tools need to provide return on investment. These approaches must be cost effective to successfully compete with other development methods, and the benefits they provide in terms of software quality must have sufficient economic value to justify investment in them.

These goals require some uncomfortable changes in the research community. Mathematical elegance is not enough for the success of an engineering theory: applicability, tractability, and ease of understanding are often more important in practice than logical completeness or conceptual elegance of the principles that guarantee the soundness of the methods. We must carefully separate the application of mathematics to demonstrate the soundness of a formal software model or to construct automated tools for engineers from the formal models that will be used “by engineers as design representations”.

The formal aspects of computing cannot be studied in isolation if it has a practical impact. The different aspects of technical, educational, and management issues are so closely intertwined in software engineering practice that it is risky and ineffective to study and develop them in isolation if practical applicability is a prominent goal. This puts interdisciplinary requirements on researchers and lends importance to interactions between experts from different specialties. We have to face the following facts:

- 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.
Software engineering being originally very much restricted to the idea of programming develops more and more into a universal discipline of systems engineering.

In many application areas such as telecommunication, business process engineering or production automation, software engineering models such as proposed by UML dominate the description techniques.

Therefore, to improve the impact, much better insight is needed to optimize our tools and techniques.

The remainder of this paper is organized as follows. Section 2 restates the main premises of the workshop and its papers. 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 workshop from which the papers were collected is the 6th in a series of international workshops with the general theme of increasing the practical impact of formal methods for software and systems engineering. The workshop took place in Carmel, California late 1998, hosted by the Naval Postgraduate School.

The objectives of the software engineering (SE) workshops are to encourage interaction between the research and engineering communities, exchange recent results, assess their significance and encourage transfer of relevant results to practice, communicate current problems in engineering practice to researchers, and help focus future research on directions that address pressing practical needs. Since 1990, the SE workshops in the series have focused on real-time and concurrent systems, software merging and slicing, software evolution, software architecture, and requirements targeting software. This workshop focused on engineering automation.

The broadest range of expert opinions and views were represented. Members of the academic, government, military and commercial world came to share their vision, insight and concerns. By synthesizing the expertise of these communities we hope to gain significant insight into the problems and solutions. The discussions ranged beyond the narrow confines of software and mathematics, to address engineering of systems containing hardware and people as well as software, and related issues that include requirements elicitation, management, and engineering education. Discussions at the workshop addressed technical advances in mature areas, such as a new decision procedure for a queue data type and novel types of model checking, as well as ideas for new directions, such as lightweight inference and coalgebraic models for interactive systems. The SE workshops have helped to reduce the gap between theory and practice, and to recharge the research community to address problems of immediate concern. Workshop attendees identified and discussed both the technologically dependent and technologically independent trends within the engineering automation of computer based systems for the near term and out to our planning horizon.
It is our pleasure to thank the workshop advisory, program and local arrangements committees, and the workshop sponsors, NSF, ONR, DARPA, and especially ARO, for their vision of a principled engineering solution for software and for their many-years tireless effort in supporting a series of workshops to bring everyone together.

The main premise of the SE workshops 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. 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 special issue). If the specification is wrong that satisfies the specification is useless.

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.

We have collected some excellent papers from the SE workshops. These articles are written by internationally renowned contributors from both academia and industry that examine current best practices and propose strategies for improvement, as well as a summary of the high points of the discussions at the workshop.

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. In the broader sense formal methods characterize the fundamental proceeding and techniques of the software engineer: in a systematic manner logical methods of its application are developed and implemented on computers.

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. Cleaveland and Sims present methods to improve the efficiency of generic, automatically generated model checkers. They 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 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. 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). Völker and Krämer 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). Cooke and Kreinovich describe a formalism for expressing implicit concurrency in data parallel computation, with applications to data mining.

4. Summary of the discussions

The discussions in the workshop addressed the full spectrum of software engineering with a particular emphasis on research topics of the future. The National Science Foundation is considering the impact of the PITAC report (http://www.ccic.gov/ac) 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:

(1) software;
(2) scalable information infrastructure (networking);
(3) high performance (peta-flops) computing, including software R & D;
(4) 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:

(1) fund fundamental research in software development methods and component technology;
(2) sponsor a national library of software components in subject domains;
(3) make software research a substantive component of every major IT research initiative; and
(4) fund fundamental research in human/computer interfaces and interactions.

Relevant research initiatives include accelerated strategic computing initiative (ASCI) and next generation internet (NGI). 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 society’s demand for software. We therefore need to

(1) empower end-users with domain-specific tools that create software;
(2) make component-based development a reality;
(3) automate software engineering processes; and
(4) 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

(1) understand what works and what does not;
(2) understand the science of software construction; and
(3) 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 cannot 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 10 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 cannot afford changes 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 efforts in 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 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.

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 cannot be realized in practice.

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 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. 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.

Software isolation is another problem. Much software is connected to communication, hardware, and other components. If we do not include models of 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 to understand because it is abstract. 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 21st century. For example, new models of interactive computing and quantum computing are on the horizon. Today’s computing environments cannot and will not be the environments of tomorrow.

Computing is a relatively new science. There is an 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 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% of money for basic science goes to physics, and only 1.7% 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 its lobby. They say, “We have this great vision. We need Congressional funding for astrophysics, etc.”, and then set up a lobby and get substantive money. We need to develop a similar vision and agree to work together toward that vision. One vision is that software engineering provides the ultimate modeling technique based on logic and algebra. Digital models provide, in contrast to the quantitative models of physics and continuous algebra, the qualitative models. These are models of data, discrete events, processes, causality, states and relations. They form the basis for knowledge engineering and engineering complex systems.

In computing science, we have not actually agreed on our goals and visions. 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”. The goals identified in the PITAC report are a good starting point for developing a shared agenda for the entire computing community.

Computing is today 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. 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.

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 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.