Automated Software Engineering, 4, 255–270 (1997)
© 1997 Kluwer Academic Publishers. Manufactured in The Netherlands.

# Artificial Intelligence and Software Engineering: Breaking the Toy Mold

CHRISTOPHER A. WELTY weltyc@cs.vassar.edu
*Vassar College Computer Science Dept., Poughkeepsie, NY 12604-0462*

PETER G. SELFRIDGE pgs@research.att.com
*AT&T Laboratories, Room C037, 180 Park Ave., Bldg. 103, Florham Park, NJ 07932*

**Abstract.** The application of AI techniques to software engineering has suffered, from the perspective of practising software engineers, due to a tradition of testing ideas and theories on small, toy domains. At the IJCAI-95 Workshop on AI and Software Engineering we focused on this issue, and here we discuss some of the results of that workshop, identifying the major weaknesses in AI&SE research, and offer some insight into the future of the field.

**Keywords:** artificial intelligence and software engineering

## 1. Introduction

There is a long standing tradition in AI to develop and test techniques and theories in *toy worlds*, small convenient domains that are abstractions of the real world. Furthermore, some still think that the issues of carrying these techniques beyond this level are uninteresting (or, at least, not relevant to the goals of research). The application of AI to Software Engineering has been no exception, and this has led practitioners to doubt that AI is of any significance to software engineering.

In August of 1995, the Third Workshop on AI and Software Engineering was held at IJCAI (The International Joint Conference on AI). The theme of the workshop was *Breaking the Toy Mold*, and we sought to specifically deal with real problems of Software Engineering and how AI can be applied to them by adopting the *Industry as Laboratory* approach (Potts, 1993).

This special issue of *Automated Software Engineering* will serve to expand upon the presentations and discussions of the workshop, and to explore more deeply the kinds of research that can result from using real-world problems as the basis for research. Included in this issue are an article by Alex Quilici and Steve Woods that explores an analogy between plan recognition and constraint satisfaction, a paper by John Mylopoulos that discusses the contributions AI has made to real-world software engineering, a paper by Robert Filman that takes you through the real practice of renovating software, and a paper by Scott Henninger describing his work and insights doing domain analysis for the Union Pacific Railroad.

Before we segue into these articles, we provide a summary of the workshop to set the context, and then take a more in-depth look at some of the issues raised during the workshop discussions. In particular, we explore the relevance of comparing software engineering to other engineering disciplines, highlight software maintenance as both critical to software engineering practice and deficient in AI&SE research, and finally we discuss the field of AI&SE itself by reflecting on its history and hypothesizing about its future.

## 2. Summary of the Workshop

The workshop was held on the Sunday and Monday immediately preceding IJCAI-95, in Montreal's Palais de Congres. This was the third workshop on AI&SE, the previous having been held at the 1992 IEEE Conference on AI Applications (Selfridge, 1992), and the 1994 International Conference on Software Engineering (Kontogiannis and Selfridge, 1995).

Our goal was to solicit participation from people who are engaged in doing AI research motivated by some of the problems of Software Engineering in the 'real' world. We hoped, in addition, to attract people from industry who could present some of these real problems, and perhaps even develop an agenda of 'open problems,' much in the way mathematical research works.

These goals were lofty, particularly for a mainly academic AI research conference. The submissions and participation were still largely academic. We did make a special effort to invite several industry participants, who helped to add a more realistic and pragmatic flavor to some of the discussions, however it is fair to say that we only partially realized our goals.

In evaluating the workshop we find an interesting analogy to the industry as laboratory approach for doing research: when dealing with the real world, things rarely turn out as you expect, and you must continually modify your expectations and success criteria. This is a very valuable lesson for anyone considering adopting this research methodology, or anyone using the methodology who has become disheartened with their progress. Putting a positive 'spin' on results is not necessarily a bad thing, beforehand one rarely has enough information and understanding to accurately determine what success will mean. We found this to be true of our workshop: it was a success, and we will conclude this section with a brief discussion of why.

The workshop web page can be found at:

`http://ijcai.org/past/ijcai-95/workshops/aise.html`

This page is up-to-date, providing a list of participants, the schedule of talks, links to most of the extended abstracts, and a full workshop summary. In addition, the summary can be found in (Welty and Selfridge, 1995).

### 2.1. Opening Panel on Reuse

The first day began with a panel on *Industrial Problems with Reuse.* This panel was held jointly with the IJCAI Workshop on Formal Methods for the Reuse of Plans, Proofs, and Programs. Michael Lowry (NASA Ames) chaired the panel which consisted solely of persons who are actively engaged in applying techniques of software reuse to industrial problems.

This panel was informative, lively, and engaging. All those present, including those from the more theoretical Reuse workshop, appreciated being 'brought down to earth' by a discussion of some of the practical problems facing software reuse. Michael Lowry charged the panelists to address four questions:

1. What cognitive barriers hinder reuse?
2. What organizational barriers hinder reuse?
3. Do source code components provide the most leverage for reuse, or should software reuse focus on other artifacts in the software lifecycle?
4. Where can AI technology best leverage software reuse?

Dr. Lowry briefly discussed his own work with Amphion, a system used to develop software for controlling deep-space probes at JPL. Here, he stressed the importance of a formal model representing the axioms of the domain, and an extensive domain-specific user interface, in enabling reuse.

Scott Henninger of the University of Nebraska, Lincoln, discussed the crucial roles of domains, process models, and knowledge in the context of his work with the Union-Pacific Railroad Company. His article in this issue summarizes many of his points; he primarily focused his presentation on dispelling the myth that any design fits into one 'domain.'

Philip Newcomb of Software Revolution described transformational reuse: transforming software in order to facilitate reuse, which was based on his experiences at Boeing. Boeing is fairly committed to reuse, and has found that in practice very little reuse is possible without some change in organizations and how people work.

Finally, Jim Ning described Andersen Consulting's approach to component-based software engineering (CBSE). Andersen is working with Pacific Bell to leverage and reuse work products across two projects, and to make sure that the systems developed will be evolvable in the future to support more services (from telephony, to video, and interactive services). This is a large scale effort by Andersen, and the role of Jim's group is to establish a reuse-centric development process and a component classification and management system in the short term, and to apply CBSE technologies to repackage components and architectures to improve reusability and portability in the long term.

## 2.2. Keynote Address

The workshop broke for lunch after the joint panel, during which the organizers learned that 90 minutes is not long enough for lunch in Montreal. The workshop at this point separated from the Formal Reuse workshop and began its own sessions. Chris Welty presented his view of the goals of the workshop as a mix of industry and academic researchers and practitioners. The participants then got a chance to introduce themselves and their particular reasons for being there.

The first presentation was a keynote address by John Mylopoulos of the University of Toronto, who spoke about representing software engineering knowledge. First, he observed that most past efforts to apply AI to SE concentrated on developing intelligent tools to support the SE process. He then proposed that AI has a role even if no intelligent tools result: AI can help by delivering new concepts and ontologies to the SE community, new notations and languages (i.e. for requirements and for design), better semantics, and SE repositories. John then described 3 cases of this kind of delivery, including requirements modeling, semantics for activities, and capturing intentions in requirements.

His paper with Alex Borgida and Eric Yu in this special issue deals with these points and offers some evidence of their validity.

## 2.3. The Workshop Proper

After a short break, Alex Quilici chaired the first of the five workshop sessions. This first session also ended the first day of the workshop, focusing on Design and Domain Knowledge. All the sessions in the workshop were in a panel style consisting of three brief (ten minute) presentations and then thirty minutes of discussion, which permitted a lot of interaction between the participants. The first day closed with a dinner at one of the many great restaurants in Montreal.

The second day began with the session on Business and Industry, chaired by Loren Terveen of Bell Labs. Lewis Johnson of USC/ISI chaired the next session on Formal Methods, followed by a session on Reuse chaired by Daniela Rosca of Old Dominion University, and the final session of the workshop was chaired by Chris Welty, and dealt with Program Understanding.

The papers available at the workshop home page provide adequate summaries of the presentations made during these sessions. The key to the workshop environment, however, is not so much the presentations, but the discussions that they provoke. The organization and atmosphere of this workshop fostered a lot of discussion, which frequently spilled out into the hallways during breaks and continued during meals.

The focus of some of the presenters on their own application experiences led to the audience sharing anecdotes about design failures in general, which led to the inevitable comparisons between software engineering and other engineering disciplines. This discussion took a new twist when the validity of the comparison itself was challenged: is there something about software engineering that is fundamentally different from other engineering practices? We take a further look at this question in the next section.

The reuse panel, the separate reuse session, and the experiences of some of the participants in trying reuse in industry provoked a lot of discussion about reuse itself. The reality is clearly evident that software reuse has thus far been unsuccessful on anything but a very small scale, and of course this leads some people to question whether it will remain a pot of gold at the end of a rainbow.

The presence of people actually using various AI techniques in industrial software environments also influenced a discussion on maintenance. Maintenance is not an avoidable step in the software life cycle, but the most costly and time consuming. Many of the more theoretical techniques, however, seem to ignore it. The issue of maintenance is a significant one, and we consider it more deeply in section 4.

The final discussion was started by John Salasin of ARPA, who challenged the participants to identify the progress made in the AI&SE field in the past ten years, saying it seemed as if we were talking about the same things we were then. Most of the participants were unprepared to answer such a question, and the discussion bounced around wildly as different people tried to take a stab at an answer. After some more careful thought, we have chosen to address this question here as well.

### 2.4. Closing Talk and Evaluation

Peter Selfridge concluded the workshop in a rousing closing talk, summarizing the workshop issues and attempting to take a realistic look at evaluating it.

To begin with, there were clearly some successes. The attendance and talks were quite diverse, despite the small audience of forty. The systems described were more comprehensive than in previous years, a trend that is becoming more apparent as time moves on, and that gives good cause for optimism that the field itself is progressing. Scale and efficiency are starting to be taken seriously by researchers, and this is not only promising, but directly relevant to the workshop theme.

There were also clearly failures. There were few industry people, and most of these had to be specifically invited. While scalability was mentioned, there was a lack of consensus on how to demonstrate it, and little evidence of real data present.

The themes of the workshop were clear only at the end, since many of them were not specifically part of any presentations, but arose repeatedly during the discussions. Analogies to other engineering disciplines was certainly one theme. The importance of legacy systems in industry was emphasized, not for the first time, but perhaps with growing conviction – legacy software problems are *increasing*, not disappearing as many hoped. Most of the presentations acknowledged that the capture and representation of other kinds of information during the SE process is necessary, and while also not new, this notion has become more mainstream. Finally, the stated theme of the workshop, breaking the toy mold, was evident in that much of the research presented were empirical or task-driven efforts.

In evaluating the workshop, we must consider, in retrospect, a few things. First of all, as pointed out by Scott Henninger in this issue, there is a difference between *industry as laboratory* and *technology transfer*. The former simply means to use real world problems to motivate research, and the latter implies numerous problems that simply have nothing to do with research. This is not an academic view, it is a real view, and is further emphasized in the paper by John Mylopoulos, Alex Borgida, and Eric Yu in this special issue. We can not evaluate research, or a research workshop, by counting users or systems, and we can not require technology transfer in the form of full productization.

The two main goals of the workshop, again, were to attract people engaged in research that used industry as a laboratory, and to attract people from industry who might provide some real world problems to motivate further research. We were clearly successful in the first regard and we selected two papers that best represent this type of research to appear in this issue: Case-Based Knowledge Management Tools for Software Development by Scott Henninger, and Toward a Constraint-Satisfaction Framework for Evaluating Program-Understanding Algorithms by Alex Quilici and Steve Woods.

We were less successful in achieving the second goal, but the workshop did not entirely fail here either. There were few industry participants, but few is not none. These participants had a lot to say, and the paper Applying AI to Software Renovation by Robert Filman is a representative of this group.

We did not come up with an itemized list of open problems for AI&SE, but perhaps this was an unrealistic goal. Legacy systems and maintenance problems were mentioned several times, and in retrospect it is fair to say that these can be considered our open problems from industry. We discuss this future in section 4.

Finally, the true evaluation of a workshop is in whether or not the participants felt it was a worthwhile gathering. The feedback we received, at least from those willing to respond, clearly and unanimously indicated that this was so, and we therefore conclude the event was an unqualified success.

## 3. Comparing SE to Other Forms of Engineering

It is inevitable that during any discussion of software engineering someone will interject an anecdote about suspension bridge harmonics, the Intel Pentium floating point problem, a building collapsing, etc. A civil engineer can hear a story about a problem with a bridge and learn from it, thinking, “I’ll have to remember that.” A software engineer hears the same story and thinks, “I’ll have to remember that next time I write some software for designing bridges.” The software engineer hasn’t learned anything about software engineering, but about building bridges.

It would seem to be more meaningful to investigate these anecdotes to determine if there is anything software engineering researchers can learn from them. We begin this investigation by briefly reviewing some of the more familiar anecdotes and the comparisons made between software engineering and other engineering disciplines.

### 3.1. Civil Engineering

The most common anecdotes come from Civil Engineering, typically building bridges or buildings. It is likely that this comparison stems from the success of this engineering discipline in general: civil engineers have designed bridges for at least a hundred years and the success rate is very high, and mankind has been building bridges for millennia. (Note, however, an interesting confluence between bridges and software: maintenance becomes the critical task and, just like software, inattention to maintenance is responsible for serious and expensive problems.) Early work in software engineering suggested the field should be as disciplined as civil engineering, identifying the “basic building blocks” of which software systems should be composed (Garmish, 1968) and certainly this is the basis of most software reuse.

Another reason this comparison is so common is that the failures, and their anecdotes, are quite spectacular, and thus we hear about them. The famous movies of suspension bridges caught in a wind-generated harmonic and tossing cars around like toys are hard to forget. Stories of a building falling because the designers used the wrong order of magnitude when estimating the wind velocity are amusing in retrospect (to anyone not directly affected by the consequences), and make for good workshop banter.

## 3.2. Electrical Engineering

The discipline that is historically and culturally closest to Software Engineering, and with which we are all likely the most familiar, is the second most common source of anecdotes. Most of us have friends or read literature in this field, and so we know of many stories. This is also a very successful form of engineering, producing a large amount of new devices with remarkably few failures. The "black box" form of hardware design is widely viewed as the ideal model upon which software reuse can or should be based.

## 3.3. Generalizing the Stories

After the stories are told and everyone adds their little bits of trivia to them, we are left with the unspoken question, "What does this have to do with us?" Can we in fact learn anything from the successes and failures of other disciplines of engineering, or is Software Engineering fundamentally different?

While the individual anecdotes don't offer much to help software engineering solve its problems, there are some general lessons that can be extracted from them. It seems clear that engineering failures fall into two very broad categories: trying something new, and human error.

It is fairly well known that when you try something new, the only way to really know if it will work is to try it. When something goes wrong, you make incremental changes to fix it. Suspension bridge failures fall into this category. Confidence in a particular design comes from the length of time it has worked without failure, and from little else. One rarely *knows* a design will work. Most suspension bridges seem pretty safe, they have operated without failure for longer than any software system, but is there still a design flaw that we have yet to discover? What if there were a rift in the time-space continuum? These anomalies regularly wreak havoc on board the *Enterprise*, imagine how they would affect a suspension bridge, or something else even our most imaginative science fiction authors have yet to think up. Seriously, while the general public often takes the safety of public structures for granted, thoughtful engineers know better, and the occasional catastrophe should remind us that there is very little, in all of our engineering activities, that is known for sure. This also again highlights the importance of the issue of maintenance for ensuring adequate functionality and safety over time.

The second type of failures, those due to human error, are omnipresent in engineering systems. They are, however, unquestionably decreasing over time due mainly to two factors: evolving engineering processes, and automation. Engineering processes and methodologies have removed the ad-hoc nature of human endeavor, and require engineers to proceed in a well-defined and structured manner that takes into account all the possible failures the field as a whole has either encountered or imagined. Automation takes this one step further by imposing even more rigorous structure on the processes, and providing a reliable repository for cases and examples based on past experiences.

Each anecdote involving human error seems to have a different "solution", in which a particular technology would have prevented the error from carrying through to the disaster. Common sense reasoning, or more likely an accurate domain model, would have prevented a building from collapsing because the wind velocity was incorrectly entered into a modeling system. Better software testing may have prevented the Intel Pentium floating-point problem, in which there were errors in the scripts that translated the output of the formal system that the floating point algorithms were designed with to a PLA.

## 3.4. Applying Lessons to Software Engineering

The first, and perhaps most important thing we can learn from these stories is that Software Engineering *is* fundamentally different from other traditional engineering disciplines in at least one very important way: *software engineering very often involves doing something new.* Standard engineering experience tells us there really is no way to know if something new will work, only that something old does work.

Numerous electrical engineers have tried to discount differences between software and hardware engineering, pointing out that ultimately anything that can be done in software can be done in hardware (their point being that software engineering is no harder that hardware engineering, so what are we whining about). Ten years ago, in fact, most VLSI engineers believed that by now general purpose processors would be a thing of the past, and that computers would contain numerous application-specific processors. It can be argued that, in theory, anything that can be done in software can be done in hardware, but in practice application-specific hardware is uncommon. Electrical engineers rarely make something completely new, and ultimately any problems encountered are *pushed up to the software.* A good example of this is RISC, in which many pipelining problems are solved by the compilers.

The second thing we can learn from these stories is the importance of domain knowledge. Curtis, Iscoe and Krasner made the point in 1988 that the one thing that distinguished the most productive members of a software team was familiarity with the domain (Curtis, Krasner, and Iscoe, 1988). The stories about human failure, however, also reinforce that a formal representation of the domain is important, to help catch these kinds of errors.

The third lesson is the importance of process. Too much software engineering is ad-hoc (though many would prefer to call this simply *programming*). Every story from another engineering discipline always brings to light, despite the failure the story may describe, that these other disciplines have well-established methodologies.

Finally, these stories continue to reinforce that there simply is no *silver bullet* (Brooks, 1987), including Artificial Intelligence. Within the AI&SE community we often find disagreement about which part of the problem, or which technique, etc., is right or best. These disagreements can sometimes be productive if they expose obvious flaws, but we should all keep in mind that since there is no *one* solution, all approaches may have some merit. This is an important point to keep in mind when reading some of the criticisms in this paper, since they are truly meant to be constructive.

## 4. The Big Issue: Maintenance

All software *must* be maintained, in fact maintenance is by far the biggest cost in the lifetime of any industrial software system. Maintenance includes the evolution of the software as well as fixing bugs and testing, and in reality these tasks are typically performed by people with the least experience in the organization.

### 4.1. Make it go away

Most research in AI&SE seems to proceed with the naive assumption that the many problems of software maintenance can be eliminated by *fixing the earlier stages*, such as requirements elicitation or design. It was clear from the workshop that industry is far more interested in better maintenance tools (be they based on reuse, program understanding, formal methods, etc.) than better design tools. Most of the money is spent on software that is well past the design stage.

Many believe that in order for good maintenance tools to exist, the system must use the right formalism, or be maintained at the design or requirements level. Even if this were true, and it is not clear that it is, the systems that operate in this fashion do not have any tools for doing maintenance, and no one has shown that tools which support the early stages in the lifecycle will make it easier to maintain the software. Furthermore, all the successful high-level synthesis systems require extensive domain theories to be developed. What is important to realize is that *these theories are the system*, and they must be maintained as well.

The point is simply that maintenance does not go away, and can not be ignored, and if AI&SE systems are to be useful in industry, they must provide for maintenance.

### 4.2. Software Understanding

Another point that seems to have been given only minor consideration in the AI&SE field is that the biggest part of maintenance is *understanding* the system. While different high-level systems may simplify or make clearer the specification of a certain piece of software, none of them serve to make it easier to understand.

This point does not just apply to thirty year old legacy systems written in COBOL. Recent evidence has shown that systems designed and implemented using object-oriented technology are *harder* to understand than their imperative or procedural predecessors (Meyers, Reiss and Lejter). Object-oriented languages tend to delocalize information, since inheritance removes the need to specify all the properties of each object, and delocalizing information makes it more difficult to understand the software (Soloway and Letovsky).

The understanding problem has not been ignored by the AI research community, but the amount of attention paid to it is small compared to the need for it. One area of promise is *program understanding*, which has been growing slowly over the past decade or more. Most program understanding tools make use of common programming *cliches* to help simplify and explain what the program is doing.

Cliche recognition may ultimately be hindered by the fact that the number of cliches is very small in practice. In the paper by Bob Filman in this issue, he mentions that in his work with very large software systems at Lockheed they found two. The Quilici and Woods paper does provide some hope, though at the moment untested, by describing a far more advanced system for a programmer to identify cliches, and for a system to then recognize more of them.

Most of these techniques, however, are purely domain-independent. It may be useful to be able to identify cliches, but it is important to keep in mind that alone these techniques don't provide a lot of insight. The simple reality is that there are two parts to software understanding: understanding the code, and understanding the domain (Selfridge, 1991). The maintainer must be able to tie what is learned about the code to knowledge of what the program is supposed to do (i.e. domain knowledge), and this brings forward another very naive assumption made by a lot of AI&SE systems: it is *not* the case that the maintainer or developer of a system is a domain expert. Again in reality, because software maintenance is the lowest rung on the ladder, many maintainers are not familiar with the domain, and understanding that a piece of code is an "input-from-file cliche" may not be very helpful unless it can then be explained in the context of the domain.

This naive assumption about the relevance of low-level constructs is not made only in program understanding, but in most AI&SE research. The necessity of a domain model is well accepted in this field, and no system proceeds without requiring one, yet most systems do not provide for *understanding* the domain knowledge. Further, it would be undesirable to have multiple domain models, one for understanding and one for the specification, as each one would need to be maintained, so there needs to be a way for the domain knowledge to serve both purposes (Welty, 1995).

### 4.3. Maintenance as an "Open Problem"

The area of software maintenance is rich with problems that could be the basis of interesting research, and the software industry is literally crying out for help in this regard. Any researcher interested in the industry-as-laboratory approach would be well served by exploring maintenance issues. Maintenance problems can inspire research at all levels, from the abstract and theoretical to the directly applicable.

Additionally, research in software maintenance has a significant advantage over research in the early lifecycle stages: there is a wealth of real data available.[1] Most AI&SE research in design and requirements proceeds with only toy data because the researchers are unwilling to develop a huge software system just to test their research ideas.[2]

## 5. Defining AI&SE

The final discussion of the workshop was started by John Salasin of ARPA, who challenged the participants to identify the progress made in the AI&SE field in the past ten years, saying it seemed as if we were talking about the same things we were ten years ago. Most of the participants were unprepared to answer such a question, and the discussion bounced around wildly as different people tried to take a stab at an answer, but finally converged on three points (not all of which were really answers):

1. In the past ten years, AI&SE *has* been concentrating on the same things, from a purely abstract perspective. The progress has been in turning what ten years ago were purely theoretical ideas to reality. The field has progressed from theories to experiments.
2. What is AI&SE itself? How can progress be measured without knowing what is being measured?
3. Is there ultimately any difference between Software Engineering and knowledge acquisition?

The first point is perhaps the most defensible and accurate. The third point is the most provocative, and we hope for much discussion about this in the future. We now take a shot at fleshing out the second point, emphasizing the roots of AI&SE, observing some trends, and expressing some opinions about the future.

## 5.1. Roots of Artificial Intelligence and Software Engineering

Programming digital computers has always been an esoteric skill, requiring a combination of technical knowledge and the patience of a trained craftsman. In the early days, writing "software" also required a detailed knowledge of the hardware that would execute it. From very early on, programmers had a vision of "automatic programming" systems that would reduce the need for detailed hardware knowledge and automate various aspects of programming. Early compilers were seen as powerful "automatic programming" systems that eased the burden of programmers and facilitated the building of larger and larger software systems (see the introduction in Rich and Waters, 1986).

As is often the case with technology, the boundary between the mundane and the interesting, in this case "interesting" automation, recedes as progress is made. Thus, compilers that remove the need to do detailed register optimization were no longer seen as automatic programming systems and a new vision of automation became popular along with the rise in Artificial Intelligence research. The idea that one could "describe the goal" of the program and have a system create the program was an alluring, if naive, view. Of course, we have now learned that describing goals is extraordinarily difficult, and AI continues to tackle this idea, with steady if slow progress.

During the 70's two other trends contributed to the apparent desirability of automatic programming. First, large scale software systems became much more widely used by business, and with it, a growing frustration with the difficulty in managing large-scale software engineering efforts. Large software systems, it was discovered, were almost impossible to manage and plan in a predictable manner (Brooks, 1995). The second trend was the growth in AI research, primarily funded by far-seeing individuals at ARPA (particularly J.C.R. Licklider). These two trends (and the ancillary fact that programming is clearly an intelligent activity conveniently practiced by AI researchers) contributed to the idea that AI techniques could contribute to solving software problems by allowing the programmer to represent their programs at a higher level, and "intelligence" in the machine could handle the grubby details of translating this high-level specification into "correct" running code.

These trends continued into the 80's with software being one of the many "toy domains" used by AI researchers to test their ideas. Two related efforts are worth describing in a little more detail.

In 1983 an influential report was commissioned by Rome Laboratory (formerly the Rome Air Development Center) to address the development and maintenance of ever-larger and more complicated software embedded in weapons systems, such as advanced fighter airplanes. The report, titled "Report on a Knowledge-Based Software Assistant", started the field of knowledge-based software engineering as such, primarily because Rome Laboratory has funded this effort since the report came out. To closely paraphrase from an excellent summary of the early results of "KBSA" work:

> "The underlying concept of a 'Knowledge-Based Software Assistant' (KBSA) is that the *processes* as well as the products of software development should be formalized and automated. This could enable a knowledge base to evolve and capture the history of the software life cycle and support automated reasoning about the software under development. The impact of this formalization of the processes is that software will be derived from requirements and specifications through a series of formal transformations. Enhancement and change will take place at the requirements and specification level as it will be possible to "replay" the process of implementation as recorded in the knowledge base. KBSA will provide a corporate memory of how objects are related, the reasoning that took place during design, the rationale behind decisions, the relationships among requirements, specifications, and code, and an explanation of the development process. This assistance and design capture will be accomplished through a collection of integrated life cycle facets, each tailored to its particular role, and an underlying common environment." (White, 1991).

This superb vision (and Rome Laboratories funding) has been extremely influential. For example, a related research effort by Dick Waters and Charles Rich of MIT was the Programmer's Apprentice project (Rich and Waters, 1990). The results of some 10 years of research produced an impressive amount of technology and ideas, including the idea of program cliches, a plan calculus that further formalizes the cliche idea, numerous demonstrations of the use of cliches for understanding program fragments, one attempt to seriously implement a Programmer's Apprentice (Waters, 1985), and the architecture of a multi-level reasoning system for software development and maintenance (Rich, 1985). Rich and Waters' student Linda Wills continued this effort with her dissertation on automatic program recognition (Wills, 1992) and subsequent work.

Of course, there have been many other complementary approaches to using AI for software development, in particular, a large amount of work on strictly formal, theorem-proving-type approaches. These approaches typically have trouble scaling, but it can be argued that they are addressing fundamental issues of rigorous software construction and deserve serious attention as a piece of the AI&SE puzzle. The KIDS system (Smith, 1990; Smith and Parra,

1993) is one impressive demonstration of the potential of formal approaches when coupled with the idea of domain specificity.

## 5.2. Today

The workshop that is the subject of this report provides a snapshot of where we are today, but we will make a few comments on other trends we see. Other sources of information on AI and SE include the annual Knowledge-Based Software Engineering conference (printed by the IEEE Computer Society Press) and the Journal of Automated Software Engineering, published by Kluwer (this journal!).

### *5.2.1. Trend: more serious implementations*

It has been a long-standing complaint, not just to AI&SE but to AI in general, that the lack of serious implementations hampers the evaluation of the ideas behind them. This complaint is legitimate, although several early systems stand out as exceptions, for example, the Sinapse work by Elaine Kant (Kant, et al) and Dave Barstow's work in the 80's (Barstow, et al. 1982). However, we are now beginning to see serious prototypes and demonstration systems and, in some cases, work that is on the verge of having serious customers.

For example, the Rome Laboratories funded KBSA Advanced Development Model (ADM) appears to be approaching "critical mass" for a serious knowledge-based development environment (Benner, 1994).

The KIDS system, mentioned above, was developed by Doug Smith and colleagues at Kestrel Institute (Smith, 1990). It has been used to synthesize small programs for solving combinatorial problems and scheduling algorithms (Smith and Parra, 1993), among others, and is a valuable example of a serious, hard-core transformation system rooted in formal theory. Some of these programs solve problems much better than programs written by skilled humans. Other examples are included in the workshop papers at the Web site.

### *5.2.2. Trend: increasing concern with evaluation*

Any new technology goes through several phases. Initially, the ideas are so new that investigation can proceed unhindered by questions of scaling or applicability. After this initial phase, however, *some* evaluation is eventually called for. This is especially true of technology, like KBSE, that claims to be relevant to real-world software problems. Recently, the KBSE community has come to recognize this issue and address it (Sasso and Benner, 1995), and we can expect more work of this kind. Of course, this trend is dependent on Trend number 1 above.

#### *5.2.3. Trend: emphasis on process support*

True to the original vision of KBSE, there has been recent work focusing on intelligent process support as opposed to automation. Selfridge and Terveen's work on the Design Assistant (Selfridge, Terveen, and Long, 1992; Terveen, Selfridge, and Long, 1995; Selfridge and Terveen, 1996) is an interesting example of this trend. The Design Assistant became a fully deployed tool in a large (several thousand person) software development organization, with a novel "knowledge maintenance" sub-process also deployed. The article by Mylopoulos, Borgida, and Yu in this special issue in another example of this trend.

#### *5.2.4. Trend: (re-) emphasis on domain specificity*

There is a natural tension within AI, and AI&SE, between the generality of algorithms and techniques and the increased leverage that comes from domain specificity. This tension was played out first with Expert Systems, which attained their power by addressing narrow domains of human expertise.

The same tension exists in the world of software research, and the trend is to try to take advantage of more domain knowledge. [3]

### 5.3. The Future

Besides the above mentioned trends, we believe that there are three big challenges for the future: bridging the gap between academic AI&SE research and the world of real software engineering and real software; the need for continued research in the role of ontologies and knowledge representation for domain modeling; and the need to view the notion of *intelligence* in a broad, unbiased fashion.

The world of software is something of a paradox: vast changes are underway in software technology and the way it is deployed, yet software problems abound and one could conclude that there is very little progress being made. This discrepancy is only worsened by the increasing "niche-ification" of the research community and the seeming irrelevance of much SE research to practice. Indeed, this motivated the workshop and its goals. Until research and practice in AI&SE are more "in synch", we cannot be very optimistic that SE practice on large systems will itself become easier.

One hypothesis that we are optimistic about is the role of formal and semi-formal knowledge representation in the eventual appearance of AI&SE systems. While some argue that over-formalization hinders the development of useful tools, such work came about partly because of the preponderance of undisciplined frame systems in the 70's and 80's. The field of AI&SE needs to experiment with a variety of formal systems and test their usefulness in different circumstances. Surely they will play a large role in AI&SE research and eventual practice.

The final issue or challenge for the field is to examine the notion of intelligence itself. We need to become comfortable with the idea that intelligence is an emergent property of a whole system, including the technology, the task, the overall process, and the user. This view reduces the role of clever algorithms or implementations as such and increases the role of "environment", if you will. It also, we believe, emphasizes the need to combine multiple techniques, including formal approaches, knowledge bases of domain and process knowledge, flexible human interfaces, software development environments, and whatever else may be appropriate to advance our goals of better SE engineering and practice.

## Acknowledgments

We would like to thank Daniela Rosca, Neil Maiden, Eric Yu, Alex Quilici, Lewis Johnson, Loren Terveen, Michael Lowry, and Doug White, for their assistance in organizing the workshop.

## Notes

1. However, this data may not be readily available to researchers in other organizations, expecially academic ones, and additional artifacts such as design documents may be impossible to get.
2. There are exceptions to this, such as (Lowry and Van Baalen) and (Kant et al.).
3. One difficulty has been the tendency for AI researchers to think that software itself is a "domain". This is partially understandable because the word "domain" is incredibly overloaded and ambiguous. There is work in progress that seeks to clarify the overlapping concepts of domain (Welty, Terveen, and Redmiles, 1996).

## References

Barstow, D., Duffey, R., Smoliar, S., Vestal, S. 1982. An Overview of $\phi$nix, Proc. of the National Conference on Artificial Intelligence, Pittsburg, Penn. August 18-20.

Benner, Kevin M. 1994. Demonstration abstract: Knowledge-Based Software Assistant - Advanced Development Model Demonstrations, Proceedings of the Ninth Knowledge-Based Software Engineering Conference (KBSE 94), Monterey, California, September 20-23.

Brachman, R. and Schmolze, J. 1985. An Overview of the KL-ONE Knowledge Representation System. Cognitive Science. 9. Pp. 171-216.

Brooks, F.P. 1987. No Silver Bullet: Essence and Accidents of Software Engineering. IEEE Computer Magazine, April.

Brooks, F.P. 1995. The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition. Addison-Wesley, Inc.

Curtis, W., Krasner, H., and Iscoe, N. 1988. A Field Study of the Software Design Process for Large Systems. Commun. of the ACM, 31(11).

Huitt, R., and Wilde, N. 1992. Maintenance Support for Object-Oriented Programs. IEEE Transactions on Software Engineering. 18(12), December.

Kant, E., Daube, F., MacGregor, W. and Wald, J. 1991. Scientific Programming by Automated Synthesis. Automating Software Design. AAAI Press.

Kontogiannis, K.A., Selfridge, P.G. 1995. Workshop Report: The Two-Day Workshop on Research Issues in the Intersection between Software Engineering and Artificial Intelligence (held in conjunction with ICSE-16), Journal of Automated Software Engineering, vol 2: 97-97, March.

Lowry, M., and Van Baalen, J. 1995. META-AMPHION: Synthesis of Efficient Domain-Specific Program Synthesis Systems. Proceedings of the Tenth Knowledged-Based Software Engineering Conference. IEEE Computer Society Press. November.

Meyers, S., Reiss, S., and Lejter, M. 1992. Support for Maintaining Object-Oriented Programs. IEEE Transactions on Software Engineering. 18(12), December.

Naur, P., Randall, B., editors, 1968. Software Engineering - Report on a Conference - Sponsored by the NATO Science Committee, Scientific Affairs Division, NATO, Brussels.

Potts, C. 1993. Software-Engineering Research Revisited. IEEE Software, pp. 18-28.

Rich, C., 1985. The Layered Architecture of a System for Reasoning about Programs. Proc. of the 9th Int'l. Joint Conf. on Artificial Intelligence, Los Angeles, CA, August.

Rich, Charles, and Waters, R.C. 1986. Artificial Intelligence and Software Engineering. Los Altos, CA: Morgan Kaufmann Publishers.

Rich, Charles, and Waters, Richard C. 1990. The Programmer's Apprentice, ACM Press.

Sasso, William C., and Benner, Kevin M. 1995. An Empirical Evaluation of KBSA Technology, Proceedings of the Tenth Knowledge-Based Software Engineering Conference (KBSE 95), Boston, Massachusetts, November 12-15.

Selfridge, P.G. 1991. Knowledge Representation Support for a Software Information System, Proceedings of the 7th IEEE Conference on AI Applications: 134-140, Miami Beach, Florida.

Selfridge, P.G. and Terveen, L.G. 1996. Knowledge Management Tools for Business Process Support and Reengineering, International Journal of Intelligent Systems in Accounting, Finance, and Management, volume 5, pp. 15-24, January.

Selfridge, P.G., Terveen, L.G. and Long, M.D. 1992. Managing Design Knowledge to Provide Assistance to Large-Scale Software Development, Proceedings of the Seventh Knowledge-Based Software Engineering Conference (KBSE-92): 163-170, September.

Selfridge, P.G. 1992. Workshop Summary: Applying Artificial Intelligence to Software Problems: Assessing Promises and Pitfalls, IEEE Expert 7: 65-68, June.

Smith, D.R. 1990. KIDS - a semi-automatic program development system. IEEE Transactions on Software Engineering Special Issue on Formal Methods in Software Engineering 16(9): 1024-1043, September.

Smith, D.R. and Parra, E.A. 1993. Transformational Approach to Transportation Scheduling, Proc. of the Eighth Knowledge-Based Software Engineering Conference, Chicago, Illinois, September 20-23.

Soloway, E. and Letovsky, S. 1986. Delocalized Plans and Program Comprehension. IEEE Software. 3(3). May.

Terveen, L. G., Selfridge, P.G., and Long, M.D. 1995. "Living Design Memory - Framework, System, Lessons Learned," International Journal on Human-Computer Interaction.

Waters, Richard C. 1985. KBEmacs: A Step Twoard the Programmer's Apprentice, MIT AI Technical Report number 753, May.

Welty, Chris. 1995. Towards an Epistemology for Software Representations. Proc. of the Tenth Knowledge-Based Software Engineering Conference, Boston, Mass, November. IEEE Computer Society Press.

Welty, Chris and Selfridge, Peter. 1996. The IJCAI Third Workshop on AI and Software Engineering: Breaking the Toy Mold. SIGART Bulletin. July.

Welty, Chris, Terveen, L, and Redmiles, D. What in the World is a "Domain", Anyway?, draft in progress.

White, Douglas A. 1991. The Knowledge-Based Software Assistant: A Program Summary, 6th Annual Knowledge-Based Software Engineering Conference, Syracuse, NY, September 22-25. Published by the IEEE Computer Society Press, order number 2605.

Wills, L. 1992. Automatic Program Recognition by Graph Parsing, MIT AI Laboratory Technical Report 1358, July.