Foreword
What is AI? And What Does It Have to Do with Software Engineering?

TRADITIONALLY, the funniest part of a journal is the overview in which the editor pretends that the articles have something to do with each other. Actually, I believe that the articles in this special issue on Artificial Intelligence and Software Engineering really *are* related, at least within three rough groupings. Whether this belief is justified or simply continues the tradition of unintended editorial humor, I leave to the reader to decide.

What is Artificial Intelligence?

The term "artificial intelligence" (AI) has suffered a continuing identity crisis over the decades since it was coined by John McCarthy (inventor of the Lisp language [2]). One popular definition of AI is "getting machines to do things that people would agree require intelligence" [4]. By passing the buck to "people," this definition neatly sidesteps the issue of defining "intelligence." However, people are fickle: once someone gets a machine to do something, they no longer think it requires intelligence—after all, a mere machine can do it! For some intriguing insights into this phenomenon, see Pamela McCorduck's history of AI, provocatively entitled *Machines Who Think* [3].

This is not the place to trace the detailed intellectual development of the field, but the main lesson learned since the pioneering work by Simon, Newell, McCarthy, Minsky, and their colleagues can be stated in four words: *Intelligent performance requires knowledge*. Lots of it. Although early attempts at general-purpose intelligent programs like the General Problem Solver [5] appeared promising at first, they proved to be limited by lack of knowledge about the application domain, whether it was chess, medical diagnosis, or natural language understanding. Since then, AI research has focused on how to acquire, represent, organize, and apply such knowledge to a variety of applications. In fact, my current working definition of frontier AI research is *figuring out how to bring more kinds of knowledge to bear*. I like this definition because the word "more" reflects the changing nature of the endeavor: work considered AI several years ago would not be considered AI today, because the target has moved. For example, what was considered "automatic programming" 10 to 20 years ago is now just standard compiler technology.

The context-sensitive, time-varying character of this definition is illustrated by work on expert systems. Building the first few expert systems was properly considered AI research, since it addressed for the first time the problem of how to bring heuristic judgmental knowledge or "rules of thumb" to bear on problems like medical diagnosis [1]. Thanks to that work, the problem is now much better understood; consequently, building an expert system is no longer fundamental AI research *per se*, unless it extends our understanding of issues like expert system architecture or knowledge representation.

What Does AI Have to Do with Software Engineering?

Attempts to get the computer to shoulder more of the software engineering burden are probably as old as programming itself. The attempts reported in this special issue can be split into three categories:

- AI programming environments;
- studies of the software design process; and
- knowledge-based software assistants.

If AI is the science of bringing diverse kinds of knowledge to bear, then the first category deals with the experimental apparatus developed to support the science, the second category deals with discovering the kinds of knowledge used in the task of software design, and the third category deals with incorporating that knowledge in useful systems.

AI Programming Environments

AI has been called a forcing function for the rest of computer science, because its intensive computational demands have required progress on so many fronts: interactive programs, timesharing, multiprocessing, faster machines, larger memories, more flexible operating systems, higher-level languages, and so forth. In particular, AI has encouraged the creation of sophisticated tools to assist the rapid development of programs for the poorly understood applications characteristic of AI.

In this evolutionary development process, aptly called "exploratory programming" [6], implementation precedes specification. Typically an AI program and the understanding of the behavior it ought to produce evolve simultaneously and synergistically; one cannot write an accurate functional specification for, say, a medical diagnosis program without first developing and testing the program. This point is further explored in the paper by Doyle. To put it another way, AI programs are just like most software, only more so: the development of an AI program can be viewed as an accelerated form of perfective maintenance involving frequent specification changes. Traditional software engineering methods, both structured and unstructured, work poorly in such unstable conditions, so AI has required the development of alternative techniques to support rapid program evolution.

As every programmer knows, the ease of programming

in a given language depends at least as much on the programming environment as on the language itself. Subrahmanyam's paper compares the two most popular AI languages—Lisp and Prolog—in terms of how well the languages and their associated programming environments support the development of expert systems, both at present and potentially.

A difficult part of developing an AI program lies in figuring out how to express knowledge about the application domain in the terms permitted by the programming language. To narrow the gap, recent AI languages have escaped from the procedural paradigm common to traditional programming languages, and provide alternative paradigms like rules, logic, and object-oriented programming, in which certain kinds of domain knowledge may be easier to express. Bobrow's paper discusses what it means for a computing environment to support one or more programming paradigms.

Models of Software Design

Since its beginning, AI has overlapped with cognitive psychology. From the psychology side, a useful way to investigate a model of how humans perform some task is to implement the model in a program. From the AI side, a useful way to automate a task is to find out how people perform it. Three papers in this issue illustrate this approach in the case where the task is software design.

The paper by Adelson and Soloway describes a model of software design based on an experimental study of professional software engineers. The paper by Kant describes a model of algorithm design based on an experimental study of computer scientists. The companion paper by Steier and Kant focuses on the use of various execution and analysis techniques to guide the refinement of incompletely designed algorithms.

Knowledge-Based Software Assistants

The bulk of this special issue is devoted to projects aimed at developing knowledge-based assistance for software engineering. Here "knowledge-based" characterizes systems with explicit knowledge about the software engineering process, as opposed to tools like editors that cannot reason about the decisions they are used to implement.

Such systems can vary along many dimensions, including:

- *Scope:* What kinds of software are addressed?
- *Power:* How much is automated?
- *Level:* What part of the route from informal requirements to machine language is addressed?
- *Purpose:* What parts of the software lifecycle are addressed?
- *Knowledge:* What kinds of knowledge are explicitly used by the machine?

In general, these questions are easier to answer for a particular system at a given point in time than for a general approach. For example, should an approach be judged by the claims made for it, or by the results achieved to date? In fact, neither criterion seems appropriate; the real criteria for judging an approach are its inherent strengths and limitations, and those can be difficult to determine. Comparison of the reported projects is further complicated by the fact that they represent efforts at different stages of implementation by different numbers of people to solve different problems. Nonetheless, these questions are a useful way to classify the various approaches described in the papers.

Balzer's paper summarizes the past 15 years of experience of a large automatic programming effort at USC Information Sciences Institute (ISI). While the paradigm advocated in the paper spans the entire software lifecycle, and previous work investigated the problem of interpreting natural language specifications, the current emphasis of the project is on the notion of developing a formal, very high level (i.e., uncompilable) specification, implementing it by applying a series of correctness-preserving transformations, and maintaining the program by revising the specification and replaying the recorded transformation sequence.

Scope: general-purpose
Power: decisions made manually and performed automatically
Level: from behavioral specification down to compilable high-level program
Purpose: specify, implement, optimize, maintain
Knowledge: intended program behavior, transformations, recorded derivation

Fickas' paper describes the results of his dissertation research, carried out as part of the ISI project just described. His system automatically selected most of the transformation steps needed to refine formal specifications into programs a page or two long.

Scope: general-purpose
Power: most decisions made automatically
Level: from behavioral specification down to compilable high-level program
Purpose: implement, optimize
Knowledge: goals and methods of program development process

The paper by Smith, Kotik, and Westfold reports on a project by Cordell Green and his colleagues at Kestrel Institute, also based on using correctness-preserving transformations to refine a formal specification. The specification is expressed in a high-level abstract programming language which the group now uses for all its programming. The project has largely focused on deriving sophisticated, efficient algorithms and on using the system to derive its own implementation. Much efficiency is obtained by interpreting transformations as constraints and compiling them to eliminate runtime computation.

Scope: general-purpose
Power: automatic or interactive, depending on desired efficiency of generated code
Level: from specification to code
Purpose: implement, optimize
Knowledge: transformations, domain axioms, logic

Waters' paper describes the Knowledge-Based Editor in EMACS (KBEmacs), a demonstration system implemented as part of the Programmer's Apprentice project at M.I.T. Unlike the other projects, which are based on top-down refinement of an abstract specification, KBEmacs provides a library of programming "clichés" (fragments of procedures). The programmer then combines these building blocks into programs.

Scope: general-purpose
Power: interactive code-writing
Level: Lisp code and coding plans
Purpose: implement
Knowledge: programming idioms

Barstow's paper reports on a project at Schlumberger-Doll Research to automate the generation of application software for oil exploration. Work so far has centered on the analysis of the domain knowledge and implementation decisions incorporated in some existing Fortran programs, and the development of a framework for representing these decisions in a machine. A key aspect is the use of domain knowledge to translate informal specifications into precise functional specifications.

Scope: domain-knowledge-intensive
Power: interactive
Level: informal specification to code
Purpose: specify, implement, optimize
Knowledge: domain model, mathematics, abstract data structure implementations

The paper by Neches, Swartout, and Moore describes an effort at ISI to make expert systems explainable and maintainable by deriving them automatically and recording the rationales for the decisions made along the way. These rationales can be used to generate explanations for users and subsequent maintainers, and encourage more principled program design.

Scope: expert systems
Power: automatic
Level: informal specification to code
Purpose: implement, optimize, maintain, explain
Knowledge: domain model, problem-solving methods, programming knowledge

Conclusion

An increasing body of work in AI has direct relevance for software engineering. AI programming environments are making possible the rapid development of complex, domain-knowledge-intensive applications that would be expensive to develop and exorbitant to maintain using conventional technology. Experimental studies of software design are improving our understanding of how that still largely mysterious process is performed by humans and might be carried out or at least assisted by machine. Finally, research on knowledge-based software assistants offers the promise of dramatic improvements in software productivity, reliability, and flexibility.

Acknowledgment

This special issue owes its existence to the efforts of many people.

Cheerful and efficient editorial assistance by Audree Beal and Toshiye Aogaichi kept things running smoothly even when I relocated from ISI to Rutgers during the busiest period of preparation.

I would like to thank the referees for their expertise and thoroughness. They helped select an outstanding collection of papers and provided a wealth of useful comments, not only about the included papers but the others as well.

The referees for this issue were as follows:

Beth Adelson
David R. Barstow
Jim Bennett
Daniel G. Bobrow
Alex Borgida
Tom Brown
Stephanie J. Cammerata
Jaime Carbonell
Donald Cohen
Daniel Corkill
Jon Doyle
Lee D. Erman
Martin S. Feather
Stephen F. Fickas
Alan S. Fisher
Michael Fox
Susan L. Gerhart
Neil M. Goldman
Cordell Green
Sol J. Greenspan
Walter C. Hamscher
David Heckerman
Cliff Hollander
Lewis Johnson
Elaine Kant
Van E. Kelly
Casimir A. Kulikowski
Curt Langlotz
Robert London
Leo Marcus
John McDermott
Matthew Morgenstern
Mark Musen
John Mylopoulos
Robert Neches
Ramesh S. Patil
Tom Pressburger
Charles Rich
Paul Rosenbloom
William Scherlis
Ted Shortliffe
Douglas R. Smith
Reid G. Smith
Elliot Soloway
Randall Steeb
Louis Steinberg
William R. Swartout
Christopher Tong
Richard C. Waters
David Wile

I am especially grateful to the referees who reviewed papers on very short notice, and to the authors who responded with equally fast revisions.

The papers submitted for this issue were subjected to the most stringent reviewing; only a small fraction were accepted. In addition, I am pleased to have invited papers from an impressive array of leading experts; these were further improved by reviewers' comments. I am grateful to all the authors for contributing such excellent papers. In short, thanks to many people this is a very special issue indeed!

Jack Mostow
Guest Editor

REFERENCES

[1] R. Davis, B. Buchanan, and E. H. Shortliffe, "Production rules as a representation for a knowledge based consultation system," *Artificial Intell.*, vol. 8, pp. 15–45, Spring 1977.

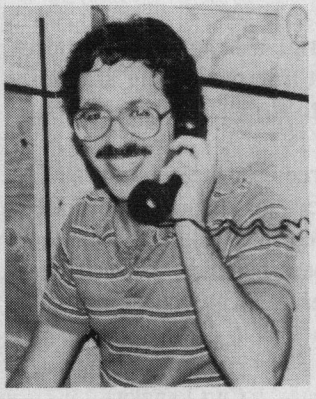
[2] J. McCarthy *et al.*, *LISP 1.5 Programmer's Manual*. Cambridge, MA: M.I.T. Press, 1963.

[3] P. McCorduck, *Machines Who Think*. San Francisco, CA: Freeman, 1979.

[4] M. Minsky, Ed., *Semantic Information Processing*. Cambridge, MA: M.I.T. Press, 1968.

[5] A. Newell, J. Shaw, and H. Simon, "Report on a general problem-solving program for a computer," in *Proc. Int. Conf. Inform. Processing*, UNESCO, Paris, 1960, pp. 256–264.

[6] B. Sheil, "Power tools for programmers," in *Interactive Programming Environments*, D. Barstow, H. Shrobe, and E. Sandewall, Eds. New York: McGraw-Hill, 1984, pp. 19–30. Reprinted from *Datamation*, pp. 131–144, Feb. 1983.



Jack Mostow received the A.B. degree in applied mathematics from Harvard University, Cambridge, MA, in 1974 and the Ph.D. degree in computer science from Carnegie-Mellon University, Pittsburgh, PA, in 1981.

After a postdoctoral year at Stanford University's Heuristic Programming Project, he joined the University of Southern California's Information Sciences Institute as a Research Computer Scientist. In 1985 he became Assistant Professor of Computer Science at Rutgers University, New Brunswick, NJ. He has authored over 40 articles, covering his work in machine learning, VLSI design, program transformation, and speech understanding, and has published reviews of current research in learning, design, hardware compilers, and knowledge-based systems.

Dr. Mostow serves on the Editorial Boards of *Machine Learning* and IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, reviews research proposals for the National Science Foundation, and teaches at the Institute of Artificial Intelligence in Los Angeles, CA.