AI and Software Engineering Will the Twain Ever Meet?

Moderator:	**Robert Balzer**[1]	**USC Information Sciences Institute**
Panelists:	**Richard Fikes**	**Price Waterhouse**
	Mark Fox	**Carnegie Mellon University**
	John McDermott	**Digital Equipment Corporation**
	Elliot Soloway	**University of Michigan**

Abstract

This session will explore the reasons for the lack of impact in four important areas in which AI has been expected to significantly affect real world Software Engineering. The panelists, each representing one of these areas, will respond to the conjecture that these failures rest upon a common cause — reliance on isolationist technology and approaches, rather than upon creating additive technology and approaches that can be integrated with other existing capabilities.

For the purposes of this debate, we've divided up the ways that AI could impact Software Engineering into four broad areas, each of which will be represented by one of the panelists:

- Development of smart applications (e.g. Expert Systems)
- Development of smart application components
- Providing intelligence within the system architecture (e.g. in expert databases or intelligent user interfaces)
- Using AI to develop software

The first three are product related and are concerned with including AI capabilities within delivered applications. The first two address embedding the AI capabilities within the application itself, while the third addresses embedding them within the run-time facilities used by the application. The fourth area concerns the use of AI in the process of generating software. This generated software may be conventional or it may include AI capabilities in one or more of the first three areas.

Our expectations for major impact upon the field of Software Engineering in these four areas, or on the field in general, have been largely unrealized. Most obviously, there has been no shift yet in the basic Software Lifecycle that I, among others, predicted would occur. The old Waterfall Lifecycle is still universally employed. In many ways, software is still being managed and produced as it was twenty years ago.

Nevertheless, many significant changes have occurred within the Software Engineering community, such as: structured programming; abstraction and encapsulation; distributed processing; object oriented approaches; the adoption of Unix and C; and most recently, formal methods and specifications. It's just that these changes have been been initiated and driven by others, not AI.

Our major effect on software engineering to date has been in defining, supporting, and getting accepted the iterative development style, especially within the context of prototyping. We are thereby also partly responsible for the increased role of prototyping in software development.

The argument is not that we've had no influence, but that the impact we've had has been much, much smaller than expected. Some of the blame for this lack of impact from our field undoubtedly lies within the Software Engineering Community, but we've chosen to focus this debate on our own actions, or the lack thereof, that have crippled this coupling between the two communities.

Conjecture: Common Cause — Isolationist Technology & Approaches

The central conjecture that this panel will debate is that the primary impediment to the impact of AI on Software Engineering is the adoption by the AI field of technologies and approaches which isolated us from, rather than coupled us to, the Software Engineering community. This isolationism has been manifested in several areas:

Idiosyncratic Language

Like other communities, we invented idiosyncratic languages for our own use. What differentiated us, was not the existence of such languages, but rather the nature of the differences between them and the

[1]The author's work is supported by the Defense Advanced Research Projects Agency under NASA-Ames Cooperative Agreement No. NCC 2-520 and contract MDA903-87-C-0641. The views and conclusions are the author's and should not be interpreted as representing the official opinion or policy of DARPA or the U.S. Government.

standard algebraic languages in use throughout the rest of the community and our interest in symbolic rather than numeric computation.

Idiosyncratic Environment

What really set us apart was the creation of idiosyncratic environments, programmed in our idiosyncratic language, and utilizing its symbolic computation facilities to represent and manipulate the computation structures of the application being executed and of the environment itself. This ability to manipulate those computation structures gave rise to a wealth of late binding mechanisms to dynamically control and alter their meaning.

In contrast, the rest of the community used far simpler, less powerful, (intentionally) less flexible, and much less encompassing environments.

Idiosyncratic Hardware

Having created our own language and support environments, and not content to merely ride the crest of the emerging hardware revolution, we took the opportunity afforded by this revolution to create our own machines.

The specialized architectures of those machines certainly provided us with an immediate benefit of increased power. But it also clearly launched us on a trajectory away from the rest of the computing community. Not only did we do our work in an idiosyncratic language, but we also now did it in separate machines that didn't run anything else. Moreover, this meant that the operating system itself had to be written in Lisp. This cut any remaining ties resulting from a shared reliance on a common operating system as we had had on the previous generation of mainframes. The primitive state of distributed system support that existed at that time virtually completed our isolation.

We not only created idiosyncratic, high performance, development machines, but we also created low-cost compatible delivery machines on which to run the applications we expected to be built and widely disseminated.

We all know now, with the advantage of hindsight, that this idiosyncratic hardware thrust was not sustainable — in part because the expected market did not develop, but also because of the huge development costs of keeping up with the rapidly improving VLSI state-of-the-art.

Recently, our isolation has been eased somewhat by the improving state of network support for distributed processing and by the appearance of co-processor plug-in boards for providing our specialized hardware support within conventional workstations and PCs.

It is interesting to note that no other language community chose to pursue a similar, specialized hardware, route.

Separate Infrastructure

In the natural progression of our separation trajectory, we built our own infrastructure. Our software development and execution environments are arguably the best ever created. But these environments also had a dark side. The better they became, the more we relied upon them. The more we relied upon them, the more insular we became, and the bigger the gap became between our support facilities and those common and accepted within the Software Engineering community.

We also created, in the expectation of major growth, our own service industry of software and hardware vendors and consultants to support our small, tight-knit community.

Large Footprint

One major problem resulting from our wealth of environmental infrastructure, was the ease with which it could be incorporated in our applications, and the difficulty of separating the part we depended upon from the rest of this infrastructure. This resulted in very large footprints (i.e. memory requirements) for all but the simplest applications.

Our response has traditionally been to just add more real memory with the knowledge that memory was always getting cheaper. But this hardware bail-out exacerbated our isolation by creating widely different hardware support requirements between ourselves and the mainstream computing community that relied on minimally configured workstations, or even PCs.

It is only recently, that we've taken this problem seriously enough to develop the necessary delivery technology to separate out just the portions of the environment really needed by an application, and thereby reduce the application's footprint.

Weak Interoperability

The all encompassing nature of our environments, the isolation induced by idiosyncratic hardware, and the dynamicism of our language all conspired to lessen our interest in supporting interoperability with other languages and/or machines.

We've certainly created connections between Lisp and other systems, but almost always in ad-hoc special purpose ways rather than building generic interoperability mechanisms. There don't seem to be any technical obstacles that are greater than those facing other communities — just less commitment on our part.

No Encapsulation

While the rest of the Software Engineering community has been engaged for some time in perfecting mechanisms for structuring applications by dividing them into manageable subunits with well-defined and enforcable interfaces between them, we have continued our

pursuit of flexibility. In that pursuit, we have largely ignored, and failed to support, the mechanisms used by others for early bindings and declarations of static structure. We have no interface definition languages, encapsulation mechanisms[2], or type-safe languages.

In the absence of such encapsulation mechanisms, we have no reliable means of breaking systems up into well-defined pieces. Hence, the understandability and maintainability of our systems is highly suspect, relying instead on conventions, good coding styles, and accurate documentation.

Egocentric Mentality

It is no coincidence that our major commercial thrust is called "Expert **Systems**" rather than "Expert **Subsystems**." We developed a technology for building a whole generation of stand-alone systems created and run totally within our own technology which solved some complete user problem. Rather than creating subsystems which could easily fit into existing systems, we temporarily avoided the issue by focusing on problems which allowed stand-alone solutions.

Another real user need we've ignored, beyond creating new expert subsystems and components, and potentially even more important, is making existing components smarter — that is, developing the technologic and methodologic base for incrementally integrating knowledge or rule-based capabilities into the existing structure of some component.

Marketplace Reaction to AI's Isolation

It is also no coincidence that the marketplace has rejected our isolationist and egocentric approach. They've demanded that both the generated expert system applications and the shells that produce them run on Unix Workstations and PCs, be coded in C (to improve performance and simplify interoperability), and be interfaced to the rest of the client's system and the environment in which the generated component will operate.

Likewise, the sponsors of the various software engineering consortia, institutes, and industrial research laboratories have rejected technology transfer utilizing Lisp based systems because they don't integrate or interoperate well, and because they don't have Lisp knowledgeable people to maintain and evolve these systems.

The commercial marketplace is moving quite rapidly toward adoption of a set of architectural and interoperability standards that support component-wise heterogeneity, and unless we can open up and unbundle our monolithic Lisp environments, we'll find ourselves even more isolated and rejected.

[2]Common Lisp's packages provide only the most primitive mechanisms for separating the public and private parts of an application, but no means of enforcing that separation

Questions to be Debated

- Do you agree that we've had little impact on Software Engineering? If not, what major impacts do you think we've had?
- Do you agree that the main cause of this lack of impact is our isolationist technologies and approaches? If not, what are the main causes?
- What impacts should we have had, given the technologic advances we've made?
- What technical problems should we have been addressing in order to have had more of an impact?
- Finally, what technical problems should we now be addressing, and what impacts do you foresee from working on those problems?