CAISOR

Cognitive Autonomous and Information Systems Open Reference Archive 1988-4
Edited by Erik Sandewall

Erik Sandewall

# Future Developments in Artificial Intelligence

Related information can be obtained via the following WWW pages:

| | |
|---|---|
| CAISOR Website: | http://www.ida.liu.se/ext/caisor/ |
| Comments about this article: | http://www.ida.liu.se/ext/caisor/1998/04/ |
| The author: | http://www.ida.liu.se/~erisa/ |

I was invited to make some remarks about future developments in Artificial Intelligence (not **the** future development **of** artificial intelligence, fortunately), and I would like to address that question from a particular perspective, namely seeing A.I. as a *community that generates new software technology.* Within our own field, we are of course used to seeing A.I. as systematic design of intelligent machinery, and or as the scientific study of intelligent systems using a particular, mechanistic paradigm.

For the outside world, however, the major practical significance of A.I. is that new software technologies sometimes emerge in the A.I. research environment. We have had many examples,[1] such as **time sharing, data base technologies, workstations, graphics interfaces** in workstations, **programming languages** such as Lisp and Prolog, **programming environments, expert systems shells,** and **natural language query systems.**

But the elements of this list are not *only* spinoffs; they have also been necessary steps in our research. As tools, and as ways of defining *where the problems are* which were then addressed by theoretical or other methods.

I will propose today that this list should be extended in the coming years, and that A.I. has good reasons to contribute again to pioneering new software technology, but maybe we should use a different approach than before.

Let us look back at the course of events in some of the previous cases. A.I. research groups have started first, or among the first, to develop and use all these[1] types of computing. But then the path from early experiments to widespread practical usage has been a difficult one to go. This problem has been very evident in recent years, where we have seen how difficult it is to bring Lisp based expert system shells into broad popularity, and at the same time we have also seen how difficult it is to get results if you try to evade Lisp and to transfer expert systems technology to the world of conventional procedural languages.

Maybe it's just that every new software technology faces an acceptance hurdle, and that that's something A.I. has to take into account as a generator of new such technologies. In this address I will first discuss how that hurdle can be dealt with for those applications of A.I. which are presently being considered, namely knowledge based systems.

Then I will go on to discuss forthcoming A.I. spinoff technologies, and in particular the practical use of mobile intelligent robots and other autonomous agents in the real world. I will discuss the problems of how A.I. techniques can be combined with the basic software techniques for real time processing, that are needed for those kinds of applications. I will also argue that these problems are not only important for commercial use of A.I., they are also very significant for basic research in our field simply because they define a number of important research topics.

The problem of accomodating older and newer software technologies is not a new one. When time-sharing was new, the difficulty of having batch

[1] Words that appears in boldface in this manuscript are those which were displayed on the screen during the presentation of the address. Anaphora in the following lines ("they", "that") often refer to the boldfaced items. In general, the present text contains several stylistic idiosyncracies which make sense when it is spoken more than when it is read, for example the lack of sub-titles, sentences which are ambiguous in the absence of intonation, etc. It is hoped that the reader will bear with that.

processing and time-sharing on the same computers was perceived as a big obstacle to using time-sharing at all. Workstation users have had to fight for liberation from mainframes. But in several of these earlier cases there has been a Hegelian development, going through the steps of thesis, antithesis, and synthesis. Separate time-sharing technology and separate workstations technology were able to build up sizable user communities and sizable markets of their own, and the synthesis could come later.

A similar strategy has been tried for expert systems. It has been assumed that one could first build a market for separate expert systems, based not only on separate software tools, but also separate hardware such as Lisp machines, and separate programming languages of an A.I. type. After the knowledge based systems had established themselves as an antithesis of the theses of conventional computing, the synthesis could evolve.

Also the synthesis was often perceived as the result of the new technology getting the upper hand. Remember how in a batch system one can embed a time-sharing service as one job which runs forever and which serves a number of users, while in a time-sharing oriented operating system, a batch job is a particular kind of session which doesn't do any interaction with a terminal. So it is one or the other, either batch or time-sharing is the dominant technology in such a system.

Similarly, optimistic scenarios such as the Japanese Fifth Generation proposal described how the knowledge based system could be the dominant technology, and how the functions of conventional computing such as data base operations or interprocessor communication, would become subservient technologies. They would have to be performed in ways which fit the general KBS paradigm, just like the batch job which must accept the rules and conditions of time-sharing in order to be accepted there.

That view is of course no longer practical. Knowledge based systems are not like time sharing; they are not going to build a large market of free standing expert systems which can later assimilate other types of computing as subfunctions.

The present state of KBS technology, as you can see it for example at the industrial exhibit here at ECAI and in the brochures of the software vendors, is characterized by the words **standard** and **interface**. A mature expert system shell runs on a standard workstation with an industry standard operating system; it has an interface to standard data base systems, and also interfaces to standard programming languages.

This is probably the best we can do in the short range. It is worth doing, and by all accounts it is much better than what the market had before, and the market is also fairly receptive to it.

From the A.I. researcher's point of view however, it seems that a lot has been lost in the process. Those interfaces are quite narrow, and get into the way for what you really want to do. As just one example, rules about how to default missing information in a relational data base are of use both for the data base processor, and for a logic-based knowledge base system that contains schemata and other meta information. The interfaces in contemporary systems are not exactly transparent for such default rules. So the present product generation *leaves us waiting for a better solution.*

Another alternative that we might consider is by analogy with embedded computers. If microprocessors are used in sewing machines or automobiles, they obviously are in a subservient position there. A car which is equipped with microprocessors is still a car, it is not a computer on four wheels. Maybe we should start thinking about embedded A.I., so that a data base system with an A.I. component in it is still a data base system and not a knowledge based system?

Actually it is very educational for us in the A.I. community to think more often in those terms, as a way of having a more realistic perspective of our own importance and in order to balance the KBS 'hype' of recent years.

But technically speaking I don't think it will work to embed real A.I. in conventional software. All evidence that we have, confirms that the programming techniques which are fundamental for A.I., require facilities which are not offered by conventional systems software, conventional programming languages particularly. So whatever A.I. spinoff methods can be rendered as embedded A.I. in the conventional framework will only bear a pale resemblance to the real A.I. thing as we know it. It might still be practically useful, but it also will leave us waiting for a better solution.

So what shall we do then? Doing conventional computing in terms that fit A.I. software, as illustrated by the left diagram in figure 1, is not practical. Embedding A.I. in conventional software, as illustrated by the right diagram, will kill the A.I. in it. Interfacing the two technologies, as in the middle diagram, leads to baroque designs which are passable but not more.

For the longer range, I think the right way to go is that A.I. should engage itself in the systems software level, or what I like to call the **Systems Software Borderland**, and we should look for natural allies for reform there.

What is the Systems Software Borderland? The conventional design for operating systems is interdependent with the conventional design for language specific systems (programming languages, run time systems, programming environments). For example, the operating system defines what a file is and how it is to be identified; the programming language contains operations for opening, closing, and accessing files, and the run time system implements those operations, usually by systems calls.

The resulting interface between OS and LSS (language specific systems) is extremely difficult to change, since new programming languages and systems must fit existing operating systems, and *vice versa*. This deadlock is particularly problematic since the division has been established based on the needs of traditional, "third generation" programming languages, which maybe are technically a bit passé today.

It is difficult to see anything on the market or in the computing industry that suggests a sudden change in this very stable deadlock situation. But that does not mean that change is not possible, and when it comes it could be so much more dramatic, because by definition it must affect both the operating system and the programming language at the same time, when some functions are reallocated from one to the other. The area that will be affected when this border is redrawn is what I call the Systems Software Borderland.

A.I. has an interest in such a change, since current technology in the borderland does not meet our needs, but if *only* A.I. would benefit then the change probably could not happen. But fortunately *the technology of fourth generation software would also have an interest in such changes,* and furthermore the reforms that would be useful to them are the same ones as would be useful to us in A.I.

Now if we are going to cooperate with the $4^{th}$ generation community, we must think not only of our needs, but also of their needs. Let us consider what the possibilities are if we could start from scratch, and design a new operating system with the needs of fourth generation software in mind: spreadsheets, data base query systems, outline processors, E-mail systems, hypertext systems, rule-based expert systems, and so on. What would be an appropriate design for an operating system that would support such software on a workstation?

Many ideas come to mind. The traditional file directory would be one of the first things to go; it should be replaced with a repository for "notecards" or "hyperobjects". Each definition in the spreadsheet could be represented as one "card" or "hyperobject" in the repository. Each stored data-base query would be a hyperobject; each E-mail message and expert system rule could be a separate hyperobject also in the same repository. All these different types of service software could be much simpler to use, and much more powerful, if they can be implemented on the assumption that the operating system offers them a hyperobject repository as a resource, and does not only give them a conventional file system.

So the proposed change is essentially one of **reducing the granularity** and **increasing the referability** of stored data: we would like to replace one conventional file with many hyporobjects in the repository, and to replace the mnemonic name (which you have difficulties remembering anyway) with access paths of many kinds.

In similar ways we would like to reduce the granularity of **processes** whose operation is supervised by the operating system. Traditional third-generation software technology will allow and encourage programs to be very modular in their source code form, but when execution time comes then the modules are again supposed to have been tied together closely, using a linkage editor of some kind. Each time an application wants to work with looser coupling between modules at run-time, that is supposed to be done within one job: within one programming environment, or within one Lisp system. The operating system usually takes its hand from all dynamic recoupling of procedures (including of course dynamic invocation).

In the new scenario, we would prefer the operating system to take on this responsibility, so that it can happen in a language independent way, and so that dynamic recoupling could be seen as a natural resource in the whole computing environment (I mean across all languages and software tools). The invocation mechanism should not be limited to the conventional one, where one procedure invokes another by calling its explicit name. Instead we need provisions for indirect calls of procedures, and for specialized interpreters or so-called superroutines, and for message passing techniques. Just like we replace the single mnemonic name for a file with multiple access paths to hyperobjects, we would like to do the same thing for processes.

Another key concept is that we would like to increase the range of globally available data structures which are administrated by the operating system of the new kind. All operating systems of course administrate certain global data structures, but in the traditional operating systems the only ones which are really publicly available are the file system and maybe a data base. My favorite global data structure in the new design would be the *agenda* which defines the future tasks for both the computer and its user.

Essentially, the agenda is a set of action descriptions which is partially ordered for time. If the agenda is administrated by the operating system, it could be a standard resource for the user (as a place to store *his* or *her* agenda), and for the computer as a replacement for the old-fashioned batch control file.

Sometimes the things that the computer has to do, relates to things that the user has to do, for example because the computer has to prepare material that the user needs for a particular occasion. Also *the way* the computer does the task may depend on when and where and how the material is going to be used. Therefore it makes sense to have a common place to store the plans for the user's future activities and the computer's future and ongoing activities.

All of this make sense already from the perspective of fourth generation software. Suppose now that such systems software already existed, so in principle there were two kinds of operating systems: third generation operating systems like Unix, and fourth generation operating systems along the lines I have described here. Third generation OS would go well with applications written in third generation programming languages and fourth generation OS would go well with fourth generation software - bypassing the conventional programming language.

Neither of these types of operating systems would be A.I. specific, of course. Suppose you were given the choice which of them you would use as the basis for A.I. software. If you go for the third generation O.S. you would put your favorite Lisp or Prolog system on top of the O.S. just like today.

If you go for the redesigned fourth generation O.S. you have an option - which you probably want to use - to also redesign the programming language. Maybe you want to let the *oblist* (the symbol table) be a global data structure in the O.S., so that all programming languages could rely on that symbol table for their data objects, and of course also for their procedure names, data type names, etc. It makes sense from a software engineering point of view to let the symbol table be a global service, but after such a reform it would not make sense for the A.I. programming language to maintain its own symbol table the way Lisp traditionally does.

Similarly, you would presumably rely on the dynamic invocation mechanism of the O.S. instead of doing the same thing again in the language specific software. Many of the features which we think of as characteristic of A.I. programming languages would make sense in an operating system for fourth-generation software. In general, reforms in the operating system parts of the systems software borderland will necessarily influence the design of A.I. programming languages in the same borderland.

As you can see, there is plenty of research opportunity if we go in this direction. Fourth generation software already has a solid position in the marketplace. At the same time the contemporary fourth generation systems are quite *ad hoc*. If we can develop a new technology for basic systems software which is significantly superior to the present one as a basis for fourth generation systems, then that innovation would have a fair chance to establish itself in the world of practical computing.

If the same technology could be seen as a good basis for implementation of A.I. systems and bringing them to practical application, then we have found a way to bypass the hurdle which now stands in the way of broader use of A.I. techniques.

In fact, the relevance of these basic software issues for A.I. are threefold. They are important for the application of our research. They are important for the research itself because it provides us with new tools. Finally they are also important because they define new and interesting research problems.

For example, the agenda that I suggested would be a useful O.S. resource, could readily be understood in terms of a logic of time and action. This would shorten the step from theoretical application description to practical implementation.

For another example, suppose that the top level executive of the operating system in a workstation is a problem-solver in A.I. sense. It maintains a set of goals or policies which reflects the user's wishes. It receives or recognizes information about what the user wants done, and information where it itself realizes that it needs to take action in order to prevent badness from happening. It deposits the plans in the systemwide agenda, where they will be executed in due time. If and when something goes wrong during the execution of the plan, then an account of the problem is returned to the problem solver which has to revise the plan in order to deal with the new difficulty.

In such an intelligent O.S. (intelligent operating system, or intelligent office system) one of the key considerations is that *the user and the intelligent software operate on the same data.* — Therefore the system must use data structures which are **flexible** enough that the user can express what he or she wants to express in that format. At the same time the data must be **constrained** or structured enough that the automatic operations which the problem solver invokes, can have a well defined job. Here we have a very concrete challenge for the *Representation of Knowledge* in the true sense of the word - how to represent knowledge in a way that carries meaning for both man and machine. Also we have a challenge which emerges from a practical situation (relatively practical, at least) and not from a set of logical exercises.

These are the reasons why we need to engage in basic software issues, just like before our field was engaged in technological issues of garbage collection and spaghetti stacks. But we must do so in a cooperative manner, not trying to define all the rules ourselves. We need to think about "software technologies" as entities, about "cooperation" between technologies, just in the same way as cooperation and joint interests between corporations. In this way we have an alternative to the earlier, Hegelian and $5^{th}$ generation scenarios where A.I. style technology is supposed to dominate and assimilate older technologies.

Let me proceed now to the topic of *Robotics and real world systems* which are important for A.I. and where the problems of basic software technology are at least as great as when A.I. technology is used in workstation type environments. For real world systems there are severe additional problems: requirements for response in hard real time, and very high volumes of sensor input data, to name just a few. Still it is interesting to consider whether new developments in the Systems Software Borderland can be a significant strategy for A.I. robotics as well.

Again let us begin by looking backward, because robotics has a long history within A.I. It is true that there have always been A.I. researchers who worked on machine realizations of the Oracle of ancient Greece, from the General Problem Solver and early Question Answering systems, to today's expert systems. But already during the 1960's there were also A.I. researchers who argued that an A.I. system must be able to move around in the world and experience it by itself. One reason was for what we now call knowledge acquisition: an A.I. system must have access to large volumes of knowledge about the world, and rather than us spoon-feeding all the knowledge to the computer, it would be better if the computer could go out and find that knowledge for itself.

Also, much if not all of what we humans say and write to each other uses our shared experience of the world as a frame of reference. Therefore (it was argued) the best way to make sure that a computer system could share as much as possible of the same frame of reference, was to design robots that could experience and handle real-world situations.

It is important to remember this today, when some groups of A.I. critics talk about "tacit knowledge" as something that A.I. researchers have missed entirely. Their argument is that there are important aspects of human knowledge and human intelligence which are never dressed in words, and which therefore is *a priori* outside the scope of a device that reasons logically from knowledge that was given to it verbally.

The answer of course is that the A.I. community has realized that all along, and that it is only the current wave of spin-off applications (namely knowledge based systems) which do not emphasize tacit knowledge as an issue. Robotics is very much about tacit knowledge. The major early A.I. projects started "Arm", "Eye", and "Cart" projects in the 1960's.

In the case of knowledge based systems, I have argued that A.I. should not attempt to introduce the necessary new software base alone. The same holds of course when we consider robotics. Every application you look at there, requires substantial contributions from **sensor technology, automatic control technology, mechanical engineering,** and so forth.

This means for one thing that A.I. will have to cooperate with those other fields, I think to a much larger extent than today. For example, one of the basic principles of automatic control is that in order to control a system - a device - you must have a model for it. In A.I. we are familiar with the idea to build models of real world phenomena; that is what Representation of Knowledge is all about. But our models are qualitative and logic based; the models used in automatic control are based on partial differential equations. One important research topic is how the different kinds of models that they use and we use can be combined and integrated.

But we also have to cooperate with those fields when it comes to software methods. All of them are already big and competent software users, and they have developed their own tools and techniques to meet their needs. And the needs are often very different from the ones we are used to: rapid response; delay from input to corresponding output must be not only short but also reliable; large volumes of data.

Those requirements are *as* essential for the total system as whatever requirements that the A.I. part may have, and they dont't combine easily with the peculiarities of A.I. software methods.

Well then, couldn't we just leave each other alone? Let the control engineers build the control part of the robot control system, let the sensor people build the sensor systems, and we'll build the A.I. part? Then we'll ask the systems hackers to implement an interface between Lisp and whatever language our colleagues want to use, and we'll be done.

The reason that that is not going to work is that it does not integrate the participating disciplines, and their respective softwares well enough. Consider figure 2, which recurs in one form or another in many papers about the software architecture of autonomous agents. It illustrates the idea that you have a low level data flow from sensors to actuators. This is the data flow for the algorithm that keeps the car on the road, and other basic motoric activities.

In parallel, you have higher, conceptual levels of processing, using qualitative models of the complex environment, planning in the A.I. sense, etc. The crucial difficulty in this architecture is of course at the point marked by a $\times$ in the figure. If the higher level has decided about a qualitative change in the agent's behavior, and wants to put it into effect while the lower level process is going on, then by what means should the higher level intervene? You must make sure that the intervention doesn't throw the whole system off balance. And, remember, we are not the only interested partners to this problem, because the continuously operating processes in the lowest layers are the ones that control theory people and sensor technology people know how to do right. So the problem is really under what circumstances, and how, should *our* software be allowed to intervene into the operation of *their* software?

The best way to deal with that problem, I think, is to make the software structure (even on the lowest level) very explicit at run time. We can use the same principles as for knowledge based systems:

**smaller granularity**

**greater referability** for both programs and data, and

**use of public data structures** supported by the operating system.

The following is one way of how those principles can be put to work, for the case of robotics software. We require that the operating system shall maintain a state vector, which you can think of as a simple blackboard. The components of the vector should be simple parameters of the observed objects and of the robot itself, such as speed and position of itself and the other agents.

The responsibility for the operating systems should be: (one) to make sure that the state vector exists at successive time points, with interval $\Delta t$; (two) to administrate and invoke elementary processes which compute some components at the next time-point from values at earlier time-points; (three) to dynamically modify this structure, and in particular to allocate and deallocate state vector components and process modules.

For example, if one more object enters the current agent's field of vision, state vector parameters for characterizing the new observed object must be allocated, and attached to appropriate procedures for continuously updating them.

This is the basic idea; the scheme needs to be elaborated in a number of ways in order to be practically useful. In particular, there is a need to have several layers of state vectors with different update frequencies; there may be a need to selectively suppress some of the computations in the network when they are not needed and computing power is scarce; and it would be reasonable to identify the higher (lower-frequency) layers, with the agenda structure that was discussed before for workstations.

The basic point that is made here, however, is that such a design could satisfy the needs of both automatic control and A.I. The calculations involved in Kalmann filtering methods, for example, could be plugged in straightforwardly as process modules. And from an A.I. point of view, we can recognize this as a kind of blackboard architecture, but where the responsibility for the blackboard has been brought down to the operating system level – which is necessary in order to meet the performance and response time requirements, and in order to open up to other types of programming languages than our own.

The explicit and modular representation of data blocks and processes also provides a handle whereby higher-level A.I. software can intervene and modify the lower layers while they are running. Such intervention would then be mediated by the operating system; as it should be.

From the theoretical point of view, one could see this as a computational paradigm which relates straightforwardly to temporal logic. This suggests that also these necessary elaborations could be understood as elaborated logics. It also suggests that we could eventually have a firm theoretical grasp of the transformation from the model of the environment to the executable structure within the computer.

Usually we think of computational paradigms as belonging to the realm of the programming language, but what we encounter here is how issues of computational paradigms are raised in the design of the operating system. This is but one more consequence of the realignment of boundaries in the Borderland.

Comparing what I said before about software techniques for knowledge based systems with this scetchy proposal for software for real world systems, we see that the same principles are put to work: smaller granularity, greater referability, and public data structures.

There is one major difference, however: for KBS we could rely on the existing, successful fourth generation software technology. The puzzle is, where is the fourth generation software for process control? If it exists, it is at least not well known. To find the reason, we must *chercher la femme*, her name is ADA. At exactly the time when $4^{th}$ generation software for personal computers emerged, the attention with respect to real time computing went to the promotion of ADA, which is a late third generation language (probably going to be the *last* third generation programming language), and which had essentially been designed in the 1970's.

However, if you look more closely into actual software products and software tools that are used for real time applications, there are in fact some which are analoguous to $4^{th}$ generation software for personal computers. It just does not get the same attention in the discussion and analysis of software trends. And just like for $4^{th}$ generation systems, these tools have *ad hoc* designs, and a lot could be done to make them more general, and to relate them to general, theoretical and software engineering principles.

So in summary, with respect to A.I. in the real world or robotic software, I propose that there must be considerable interest, not only in our own camp, for a new design in the Borderland for real-time systems software, where again

the **real time operating system** would accept more responsibilites than today;

the **programming language** would be designed differently from now because it can make use of the new operating system services;

and the **A. I. software** e.g. for planning and plan execution, would make very active use of the O.S. services, which is the channel through which it relates to the software provided by the other participating disciplines.

Let me return now to the original question about my view of future developments in artificial intelligence. I take for granted that A.I. will continue to be guided by the unifying vision of an intelligent electronic and mechanical device. During the 1980's we have seen a strengthening of the theoretical part of A.I. - the part which is founded on logic and discrete mathematics. That development has of course been very good for our field.

But my argument here today is that logic *alone* will not achieve the vision. Logic plus technical implementation work by hackers, or structured programmers, or hardware designers will not achieve it either. Instead I have argued that a lot of work also needs to be done in the area where A.I. and systems software research meet. The borderland is not merely between programming languages and operating systems, it is a borderland between programming languages, operating systems, *and A.I.* And just like in the 1960's and like in the 1970's, A.I. should continue to generate new ideas for how systems software ought to be organized. The areas of computer science called operating systems and programming systems run a certain risk of just extrapolating existing approaches, if they are left to themselves. A.I. can help by proposing alternatives to some of the established patterns.

So, when artificial intelligence serves as a generator of new software technology, our rôle is to focus on the very challenging long range problem of intelligence, and to identify how to organize all layers of software in order that intelligence can be embedded there. Not only to write intelligent software, but to *organize all layers of software in order that intelligence can be embedded in it.*

It has been our tradition to reject conventional basic software, because it is not suitable for embedding intelligence in, and to build our own from scratch as often as possible. One of my key points in this address has been that that strategy is not viable any longer. We must begin thinking about technology partners, i.e. other technologies which have similar needs for software reform as we have, and we must begin thinking about how we can interact and cooperate for example with the fourth generation software technology.

In summary, therefore, I think that A.I. will continue to be a field where theory and experimental practice co-exist and interact in a constructive way. I think that A.I. will also continue to be a field that upsets other parts of computer science, by disturbing established patterns of thoughts and by insisting that computer intelligence is a valid scientific and technical objective.