BIOLOGICAL SOFTWARE

Erik Sandewall
Informatics Laboratory
Linköping University
Linköping, Sweden

Abstract and introduction: The robot in the common science fiction novel appears as a man on the surface, but is built mechanically from wheels and levers inside. In A.I., we usually visualize an A.I. system as having a similar structure: it communicates in man's language (English), or performs other tasks which make it appear man-like, but it is in fact a large program, written in a programming language, and executed under a common time-sharing system.

The present paper argues on the contrary that A.I. systems *should not* need to appear man-like, *and* that it is necessary, both from the A.I. point of view and from the software engineering point of view, that the next level down in the A.I. system has quasi-biological properties, such as the ability to reproduce.

Remark: due to space limitations, this paper has been written in a condensed style.

1. *Reproduction is a mechanism for a species to adapt.* It is natural to make analogies between the biological system that forms the substrate of natural intelligence, and the computer hardware/software system that forms the substrate of A.I. Among characteristic properties of organisms we find homeostasis and the ability to reproduce. Do they have counterparts in software?

The common game of writing programs that create many identical copies of themselves, misses an important reason for biological reproduction: it is *a mechanism for adaptation,* from the point of view of the species. Human reproduction involves not only copulation and child-birth, but also two decades of work for training the new individual (sometimes called "social reproduction" by sociologists). In the training process, the new person inherits some knowledge, but also selects, reviews and reorganizes his or her intellectual heritage. Imagine how much progress there would be if humans lived forever and never reproduced!

2. *Software also needs to adapt, in response to changing user needs.* Conventional software does not *adapt autonomously to any significant extent: it is adapted* by a programmer who changes the program (*"maintenance"*) or throws it away and writes a new one.

Future software systems with a more complex behavior will have much bigger needs to adapt, and will be harder to change from the outside. They should therefore be enabled to change without programmer intervention, in response to requests from the end user. They can do so either by modifying their internal state, or by creating a modified copy of themselves. *Both of these methods are viable if the programming technique of* conceptual programming *is used.*

Remember that adaptation was a topic of interest in earlier A.I. work, but it has gone out of fashion because it seemed one could only do trivial things with it, such as adjusting numerical parameters. What one would really like to do was to adapt programs, but that seemed too hard, and had to wait for automatic programming to happen first.

But there is a trend in several areas of software to use specialized application languages (implemented either as interpreters i.e. general, highly parametrized programs, or as program generators) to develop applications. Examples of this trend are:
- commercial program generators for reports, screen layouts, data base queries, etc.
- the system for information-flow applications, developed in our group (ref. 1).
- special-purpose languages in A.I. research e.g. grammar languages.

These are signs of an emerging software technology where computer applications are implemented by selecting a small number of general-purpose tools, combining them in appropriate ways, and providing them with a description of the application in (several) specialized application languages. This style of programming has been called *conceptual programming* in ref. 2,3,4.

Self-modification in a system should clearly be relatively easy if it can be performed by modification of parameters or application-language expressions, as compared to direct self-modification of a conventional program. This is both because easier "maintenance" is one of the reasons for specialized languages, and because end-user requests for modification are naturally expressed in application-language terms.

3. *Should an A.I. system 'live' forever?* In other words, is internal self-modification sufficient as an adaptation mechanism, and is reproduction unnecessary? The answer is that perhaps it is not practical for software to adapt forever. For adaptation, new knowledge has to be added to a system, which means that other knowledge has to give way, otherwise the system will grow indefinitely. But whenever knowledge is removed, one encounters a well-known but yet un-named problem which I propose to call *the delete problem,* for example in the following simple form: suppose A has been asserted in a data base, and B has also been asserted by forward inference using "A implies B", and later A is to be deleted. Should B be deleted as well? There might be other, independent support for B, so that it should not be deleted. The problem is well-known on many software levels, from conventional garbage collectors to semantic networks.

The conclusion is that for a system to be indefinitely adaptible, it must contain a considerable overhead of information about the details of its own insides, such as back pointers from B to all its independent supports, in the example. Reproduction offers an alternative, with copying garbage collection as an example on the low software level, and training of off-spring as examples among animals and (I conjecture) for A.I. systems.

4. *Reproduction takes many forms; ours is a biological exception.* Reproduction in mammals has a simple structure: two individuals together generate an offspring which grows up continously. But butterflies, ants, fish, jellyfish, frogs, and funghi offer a rich reportoire of more complicated schemes, where the organism exists in more than one physical form during the reproductive cycle.

In several projects in our research group, I have seen needs for similar, non-trivial schemes for reproduction in software.
- *Background: some of the group's work assumes that the programming techniques that have been developed in A.I. research, are a significant spinoff result of that research, and should be used for other purposes as well.*

Two specific cases are of interest:

A) Use of the Interlisp system for development of pilot versions of application programs. (A pilot version is one which can easily be changed until the end users are happy). Development of the initial program went smoothly, but the subsequent user-initiated modifications were clumsy to make. Analysis: when conventional languages are used, one does debugging in the production environment (i.e. using the regular compiler). The Interlisp system provides a "dry-dock" environment for program development, which implies that a non-trivial effort is required to take the program into and out of the "dry dock". This is worthwhile if a lot of work has to be done there, and in particular if the program never really leaves the development environment, as is often the case in A.I. research. But the same programmer support did not seem worthwhile as the programmer's effort for transfer from development environment to production environment was much bigger than the effort of the update itself.

The problem was solved by allowing each application to exist in two forms: a development system and a production system. The development system was the one that really 'adapted', and it had the ability to create corresponding production versions.

This solution embodies the following basic attitude: a programming system should not be like a street-front clinic, where a program walks in to be operated on (for example, for being compiled, or debugged during

a debugging session), and then leaves again. It should instead be a permanent dwelling for the program. The programming system plus the program it accomodates, should be viewed as an organism, which contains a fair amount of knowledge about itself, and in particular has the ability to generate systems which are similar to itself although tuned for production use - like a queen bee does.

B) Modelling information-flow systems. Many data processing applications can be characterized as information-flow systems: there are specialized work-stations for different people or roles in the organization; information packets ("transactions") are generated in such stations, sent in channels from one station to the next, accumulated in files that are local to each station, and the task of the person at a station is to review the information that flows by, take actions, add more information to the transactions, and pass them on.

Recent work in our group has resulted in a prototype software tool which allows the various aspects of the information-flow application to be described in specialized languages, and which allows one to do pilot execution of the system. work on a generator of production systems is in progress. The system is designed with the programming techniques that are usual in A.I. programing, such as strongly parametrized programs, handles, and data-driven procedure calls.

In the terms of the previous sections, this system is "organism-like" in that it can create modified copies of itself, namely copies for each of the work-stations. These specialized production-phase copies have been obtained by selecting subsets of the parameter structure, and by specializing programs through "smart" compiler macros.

These and other applications in our group are using a general-purpose tool for accomplishing system reproduction, called ACTEMAN (ref. 5).

5. *Knowledge acquisition.* The reproduction in these two examples as presently implemented, does not involve any selective acquisition of knowledge by the off-spring. (There is however a real need already to have that). The major reason for the examples is to illustrate that multiple 'life-forms' and unusual (for us) reproduction structures may be useful for software systems.

With a system which is organized along the lines of conceptual programming, as mentioned above, knowledge acquisition by the young off-spring could be organized as follows:

- let the ancestor generate a basic system (for example a fresh instance of the Lisp system), and load the appropriate general programs into it. Also let the ancestor transfer the appropriate parameters and datadriven procedures to the descendant, but usually only a subset of what the ancestor "knows"
- let the descendant engage in work, usually by serving a human user, where in doing so one identifies missing parts in its structure and obtains the information by communication with senior computer systems, (in particular, systems similar to the "deveopment phase" system described above, which contain extra knowledge and which specialize in training new systems), and only in hard cases, communication with a human programmer.

In such an architecture, it is possible but not natural to let the computer systems communicate between themselves in human natural language. It is much more natural to exploit the properties of the computer medium, and let them communicate in terms of program segments, data structures, and so forth. (This does not preclude that some of the characteristics of naturallanguage communication will still be needed)

6. We can now proceed to the other issue mentioned in the introduction to this paper: *The emphasis in current A.I. research on systems with humanoid behavior.* Its main historical reasons are:
A) The training issue: it has been argued that intelligence presumes knowledge, and that the computer can only gain knowledge by talking to people and/or being among people, as a robot.
B) The usefulness issue: it has been argued that an intelligent computer system can only be useful to us people, if it can talk to us in our language (which is assumed to be natural language).

The training argument addresses the wrong bottleneck. It is true that a lot of work will have to be spent by people for transferring knowledge to computer systems, and it is conceivable that it will be facilitated if natural language may be used. But already scores of programmers have as their profession to transfer knowledge and ability to computers. The significant problem is instead that computer systems can not transfer knowledge to each other. If one computer system could train others, then it would not matter if the initial knowledge transfer to one computer system had to be done in a formal language

The reproduction chain can start when we have built systems which have the necessary properties for being useful, and for generating useful offspring, recursively. In order to start it, we ourselves will have to understand the reproduction process fairly much in detail. It then seems more convenient to create the

first generation by performing manually the operations that will later be done by the ancestor, rather than create a natural-language understanding capability only for the purpose of the first system generation.

The usefulness argument (B) is valid if we consider current research results as given, and search for possible applications. But suppose instead that we would focus on *current* data processing (D.P.) applications, and ask what role programmed intelligence could play in them. The role might not be dominant: the resulting thought product is not necessarily a computer intelligence incarnated into an application. A car with a micro computer in it is well described as a car and not well described as a computer, and intelligence might be very useful in a similar, subordinate role in the D.P. system.

7. *There is a significant area of common interest for A.I. and software engineering.* In A.I., conventional programming has traditionally been considered

- as an intelligence-requiring activity that is observed to occur in real life, and which therefore is a candidate A.I. application, and
- as a part of the support work for an A.I. lab and a temptation for the graduate student to waste his time on

It is then assumed that software engineering and A.I. are disjoint areas. The arguments of the present paper imply on the contrary that

- systems which adapt, internally and by reproduction, are necessary both for A.I. and for future software engineering
- conceptual programming techniques may facilitate design of large and complex systems (which is a topic of common interest for A.I. and S.E.) as well as adaptation and reproduction.
- such systems, performing conventional D.P. tasks, are a likely early application area for A.I. research. But intelligence can not then be obtained as an add-on: the system has to be built from the start using some of the programming techniques which are common in A.I.

8. *How does this fit in with other work in A.I.* Hewitt's actor concept is clearly related in the abstract sense that he also talks of structures formed by individuals with a local autonomy and initiative. However, he has mostly worked with actors of very simple structure, in particular as an elementary object in a theory of computation. We emphasize systems which are complex enough to contain intelligence; systems which are pseudo-individuals. Marr's work in computer vision has already started a trend to take the biological foundations of intelligent systems seriously again

Acknowledgements

Jim Goodwin and Sture Hägglund have contributed strongly to the ideas expressed in this paper.

References

1. Erik Sandewall: *A description language and pilot-execution executive for information-transport systems.* Proc. Very Large Data Bases conference, 1979.

2. Terry Winograd: *Five lectures on artificial intelligence.* Stanford A.I. Memo, 1974

3. Erik Sandewall: *Some observations on conceptual programming.* Machine Intelligence 8, 1977

4. Bob Wielinga: *A I Programming Methodology.* Proceedings of the AISB/GI Conference on Artificial Intelligence, Hamburg, 1978

5. Erik Sandewall: *Self-organizing Information and Operations for Reproduction in Distributed Programming Systems.* Internal report, Informatics Laboratory, Linköping University, 1979.