Presented to International Workshop on AI and Software Engineering,
Exeter University, April 12-14 1989.

SOFTWARE ENGINEERING AND ARTIFICIAL INTELLIGENCE

(Formal Methods vs Intelligent Insight?)

Aaron Sloman,
School of Cognitive and Computing Sciences,
Univ of Sussex, Brighton, BN1 9QN, England

NB: Presented at The International Workshop on AI and Software Engineering at the University of Exeter, England, 12-14 April 1989, sponsored by AAAI, DTI and Ablex Publishing Corp.
This is a "position paper" designed to provoke discussion. No claims are made for rigour, depth, scholarship, originality, etc. I take the central question to be:

How can computers help us with software production and maintenance?

In particular, how much of the task can be automated?

We need to think about different subtasks, and what problems they generate, before we can hope to implement automated systems. I'll list some subtasks that play a crucial role. I suspect that only a subset have even been thought about by software engineers. Maybe the least important subset?

The order of tasks given here is not meant to be a chronological ordering. An application might be thought of some time after the relevant tool has been implemented.

Task T1. Notice the possibility of a new application.

This involves noticing that it would be a good idea to try to have a system that can do X (e.g. where X = format text, monitor heart patients, advise insurance salesmen, translate English into Fortran, distribute electronic mail, etc.). T1 requires a deep understanding of the scope and limits of current technology and the relevant application area in order to see the possibility of a match long before there is either a specification, or a design. How do people do this? How were spread-sheets, or WIMP systems first thought of? Thinking up new technological objectives requires creativity on a par with great art or science.

Task T2. Come up with a good specification for X.

Task T2 requires a detailed knowledge of the environment in which X will exist, and how that environment is likely to develop. This includes knowledge of possible processes, constraints, resources, goals, problems, etc. Where human beings are involved it requires deep insight into their cognitive capabilities and motivational processes, including the ways in which these will change as a result of exposure to X.

Moreover, if the system has to cope with wide variation in environments (including, for instance, enormous variation in individual human abilities, preferences, motivations, learning speeds, etc), then task T2 requires creative insight into an appropriate level of generality that will support a variety of forms of accommodation and tailoring.

The longer X is going to last, the more the environment is likely to change.

An alternative to getting the specification right, is finding a way to make the specification itself adaptable, even to the extent of allowing re-programming by users. I.e. design not just software but VERY SOFT WARE.

Task T3. Produce a good design.

This requires an intricate bridge to be built between syntactic knowledge of available building blocks (i.e. what hardware and software facilities are available for assembling into a new design - including languages, subroutines, packages, techniques, etc.) and semantic knowledge about the functions to be achieved, the constraints, the properties of the environment, etc. It is much harder to produce precise mathematically manipulable descriptions of the latter than the former. How is human knowledge about objects, properties, relationships, events, processes, and constraints in the world encoded? How is it used in producing or checking good designs? How can ideas about what makes a satisfying work environment for different kinds of users be mapped into a design? Much of the knowledge used by a good designer is very hard to articulate, like our knowledge of English syntax, or the intuitive knowledge about human vision used by a good painter.

Even when if the knowledge is formalised, for a system of any complexity the potentially relevant building blocks and potential ways of putting them together will generate a combinatorial explosion that totally rules out exhaustively trying different combinations and checking them. How can insight and experience defuse such explosions?

Using higher level building blocks (libraries, OOP classes, higher level languages, etc. etc.) can reduce the combinatorics by orders of magnitude, but

> (a) at the cost of eliminating many possibilities (e.g. some potentially useful aeroplane designs can't be assembled from existing aeroplane parts - and similarly with software)
>
> (b) even reductions of several orders of magnitude don't help with exponential functions of large numbers. (2**100) is so big that dividing by 1000 will still give you a number larger than the number of electrons in the universe.

So, only by having a very high-level view of the space of possibilities which enables one to prune HUGE subspaces without any detailed consideration, will one have any hope of finding good designs. Good human designers seem to achieve this. How? What representations do they use? How do they manipulate them? Are these the right questions to ask?

Task T4. Produce an implementation of the design

There is no sharp boundary between design and implementation. Even the simplest bit of code-writing is designing. Severe combinatorial explosions can arise here too because of the number of components in a typical program and the number of possible ways of assembling program expressions. Lots of people seem to think that object oriented languages will make a big dent on this problem. This is just another one of those fashions followed in lieu of an analysis of true nature of the problem.

Another mirage is the use of formal methods. Yes in principle. Assuming that you can test whether a program meets a specification (see below) you can easily write a program that automatically generates programs, by enumerating all legal programs in the language to be used, in order of increasing length and alphabetical order for a given length, and checking them in turn against the specification, till a solution is found. Unfortunately this theoretically elegant solution is totally defeated by the combinatorics. What reason is there to believe that mere use of more sophisticated formal methods (and even some heuristics) will overcome the combinatorial explosion? Once again, even improvements of several orders of magnitude are too small to be useful.

Task T5. Check that your implementation meets the specification

This requires the semantic gap to be bridged between a specification described in terms of the external world and a code description that’s essentially all syntax. Data-structures may be semantics relative to the language formalism, but they are still syntax relative to the function of the software in the world.

If you produce the specification in terms of kinds of objects and processes inside the computer (data-structures, transformations of data-structures, etc) or even inputs and outputs defined in the way that the program defines them, then you still have the semantic gap between THAT specification and the real specification the user/customer etc. is interested in.

Of course, no GENERAL method for proving that a program meets a specification is possible. (E.g. the halting problem). Do we know which interesting and useful families of sub-problems are decidable? How are ‘interesting’ and ‘useful’ to be defined? Could the definition be mathematically formulated?

Even when the problem is solvable in theory, it may still be intractable if so many different combinations of external circumstances have to be considered that we again have something like an exponential function from number of factors to possible combinations. Finding good ways to move to an appropriate level of abstraction, or good ways to factor the possibilities, requires great insight and intelligence. Do we know how good mathematicians do it?

Task T6. Damage limitation and bug-fixing.

Given that specifications will often turn out not to define what people want, that designs will often turn out not to yield good implementations, that implementations will often turn out not to be free of bugs, we have a variety of damage limitation and bug-fixing tasks.

All of these have potentially the same combinatoric problems as the previous tasks, though in particular cases they may be trivial. In real life the user who reports a bug may not in fact have described it at the level of generality that gives any clue as to what the source of the problem is. All she can do is give one, or more, examples of its manifestation. My experience is that there is an enormous difference between the expert de-bugger who usually goes fairly quickly to the source, and the majority of programmers who flail around trying to narrow it down. What knowledge is the expert using? How is it represented? How is that knowledge manipulated? How is it acquired and extended?

Task T7. Comparing alternative specifications, designs, implementations, etc.

It’s one thing to produce a good one. What about producing the optimal one? What does "optimal" mean? Different criteria of optimality have to be considered: space, time, generality, cost, ease of maintenance ... Trade-offs have to be considered, e.g. going for speed at the cost of space, or minimising garbage collections to get uninterrupted performance, rather than optimising speed or space. Compare ease of implementation and maintenance vs ease of use and tailorability by end-users. The trade-offs will, in general, be different for different types of applications, so an ’automated’ designer would need to be able to generate the appropriate sub-categorisations for each task. The trade-offs will depend to some extent also on the target language and hardware. How does a good human designer think about trade-offs? At what stage in the design process does knowledge of trade-offs play a role? How? Clearly it would be daft to try to produce all possible designs and then assess them. So how can the search space be pruned sensibly?

SHOULD CASE TOOL DESIGNERS BE COGNITIVE SCIENTISTS?

My main worry about what seem to be current software engineering or mathematical approaches to the design of new tools or methods or formalisms to aid the production of good software, is that they start from some abstract idea of what the problem is and what the form of a solution must be like ("a program that will transform a specification into a program"). They do not try to get any deep insight into how the human geniuses work and how their mental processes might be modelled, or replicated. The result is likely to be tools and techniques that work on TOY problems and explode on anything even as complex as a compiler, let alone a networked operating system, or natural language interface.

They don't do the kind of AI that is intimately bound up with cognitive science and attempts to find out how human beings avoid these problems. Trying the latter might show that we have a much longer way to go than we thought. Looking for your lost keys where the streetlamp is may make the search easier, but doesn't necessarily increase your chance of finding the keys.

What are the methods by which those rare and precious software designers with vast experience, deep insight, and unerring aim produce good code and KNOW that they have done so? (Notice that this is not the same as knowing that they have not left any minor bugs in the system. A good design with a few minor bugs may be worth a lot more than a perfect implementation of an inferior design.)

Is there any reason to believe that the techniques currently being explored have any hope of giving computers similar abilities?

My hunch is that for a long time to come we are going to be dependent on the very best human designers and implementors. So, in the short run, our best strategy is not so much to try to design automated tools but to try to understand how the best humans work, and then look for ways of helping them do their jobs with less time and effort, and ways of helping the less good humans become more like the best.

This may require a combination of training techniques, good language design, and a host of new kinds of computer aids. These aids must be based on deep analysis of what help people most need with the main tasks, not apriori assumptions about what will help.

What is not always realised is that a language is as much a tool as an editor or browser or a library of re-usable design schemata. How many computer scientists have tried to design programming languages to meet the cognitive processing requirements of human programmers, as opposed to the needs of computers or the criteria of mathematical elegance and rigour? What are the cognitive processing requirements of human programmers?

Note: PDF version created using Groff on Linux (Fedora 33).
Slightly re-formatted on 4 Oct 2022.
