Software Engineering and Artificial Intelligence in New Generation Computing

SPL Insight, a company of the British SPL consortium devoted to advanced information technology studies, has given its 1984 Award to Professor Bob Kowalski for his achievements in Fifth Generation computing. Professor Kowalski delivered an Award Lecture on May 15, 1984 in London. He and SPL Insight have kindly given permission to reproduce his lecture here.

ROBERT KOWALSKI
Dept. of Computing, Imperial College of Science and Technology, 180 Queens Gate, London U.K. SW7 2BZ, U.K.

I am especially pleased and honoured to receive this award because of SPL Insight's concern with practical matters. I would like to see my work and that of my colleagues as having not only academic interest, but also economic and human value.

In this talk I would like to look at the applications of Artificial Intelligence technology to Software Engineering, and in particular to the systems analysis stage of software development. I shall argue that Artificial Intelligence allows us to execute systems analysis; and in some cases the execution is efficient enough to remove the need for separate specifications and programs. I would like to support my case by looking at the British Nationality Act as a particular example, which is closely related to data processing – the execution of rules and regulations whether they have legal binding authority or they are simply the rules an organisation follows for its own convenience. I would like to mention some of the other interrelationships between Software Engineering and AI. And finally I shall tread on ground I haven't tread on before and discuss some of the human implications of the technology.

I am afraid that not all of the consequences of the Fifth Generation are going to be beneficial; and that we will not be able to avoid some of the worst of these consequences, unless we are aware of the potential dangers.

The Fifth Generation

I would like to start by looking at the Fifth Generation.

The Japanese have identified the focal importance of Artificial Intelligence applications (see Fig. 1) They have identified logic programming as the underlying software technology; and they have identified new kinds of computer architectures. Certainly their new applications can be understood by the person on the street; and the electronical engineer can understand the computer architectures. But until the Japanese drew attention the the logic programming software, most computer scientists had either rejected it or knew nothing about it. The computer scientist's view of computing is conventional (see Fig. 2) not only with respect to the applications and the hardware, but also with respect to software methodology:

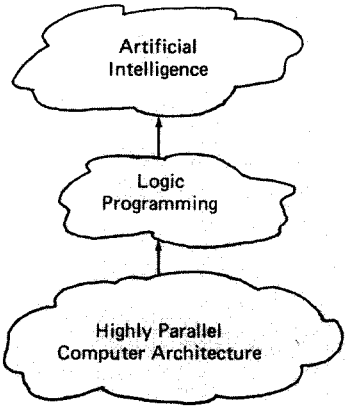

Fig. 1. The Japanese View.

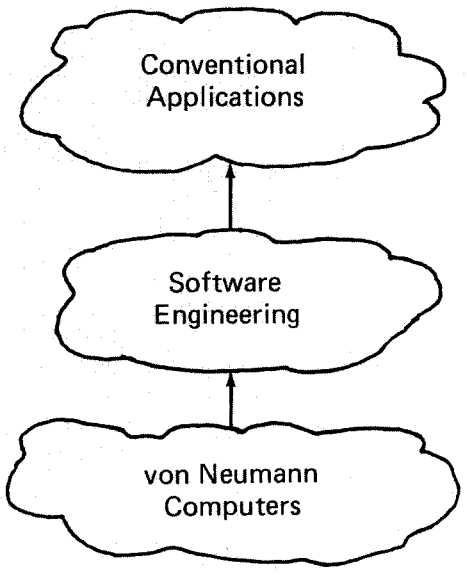

Fig. 2. Conventional View.

conventional number-crunching, scientific and commercial applications, executed on boring computers that run sequentially, have to be told every step, and cannot make any decisions for themselves.
And how do we bridge the gap between the boring application and the boring computer? By using boring software engineering techniques.
Feigenbaum has tried to correct our impression of the Fifth Generation (Fig. 3). Feigenbaum and McCorduck in their book on the Fifth Generation emphasize a very valuable, focal part of the Fifth Generation, that is its novel expert systems applications. They downplay the new computer architectures and the logic programming software. I do not want to argue that PROLOG should be regarded as

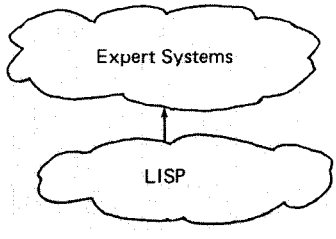

Fig. 3. The Feigenbaum View.

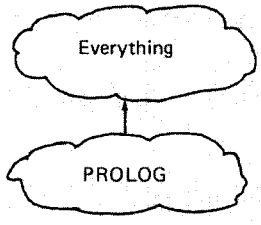

Fig. 4. Not My View.

the correct substitute for LISP or that PROLOG is suitable for all applications today (Fig. 4).
I would like to argue, however, that the new technology associated with logic programming and other declarative languages supports not only new applications but old applications as well (see Fig. 5). Thus I would argue that we must distinguish between technologies and applications. The new software technologies, of which logic programming is the most representative, not only enable new applications in areas such as expert systems and natural language processing, but also facilitate the implementation of old applications as well. They support various software development methodologies, not simply old ways of programming but also new ways of developing programs. I shall concentrate in this part of my talk on the application of the new software technology to the systems analysis phase of software development that is prior to both software specification and programming.
So what is this new technology? The new technology is characterised by the fact that it allows knowledge to be represented explicitly. You can see what the knowledge is; and that knowledge is separated from the way it is used to solve problems. It disentangles what the computer knows from how the computer uses it. The computer uses its knowledge to solve problems by reasoning deductively in a manner which simulates human reasoning, and which is congenial therefore to human thinking and to human-machine interaction. This means that the new software lets us see the knowledge and therefore understand it. This means that we can develop knowledge in an incremental fashion, because it's not all tangled together with the way it's used. And it is easy to modify, if we've made a mistake or if the knowledge changes, as it does very frequently in applications such as the formalization of legislation.

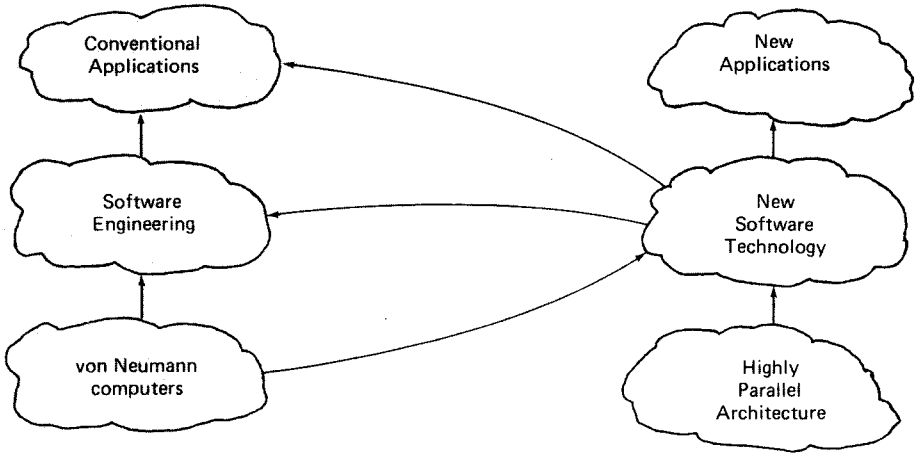

Fig. 5. Another View.

The New Technology in The Software Development Life Cycle

Let's look at the place of the new software technology in the conventional software development life cycle, as pictured for example by DeMarco (Fig. 6).
I want to draw particular attention to the bottom path of the diagram which is concerned with software. We start with the user requirement, namely the problem the user has or thinks he has. We analyse the requirement, come up with a functional specification of a computer-based solution to the problem, and then design a software system which we eventually implement as a program in a well structured top-down manner.
The *data flow diagram*, which describes the software development life cycle, is a convenient tool for the use of systems analysts to interact with users. It's a language which systems analysts have developed to communicate better with people. But data flow diagrams can also be interpreted as an alternative, graphical syntax for rule-based programming. Take for example the following rule which expresses

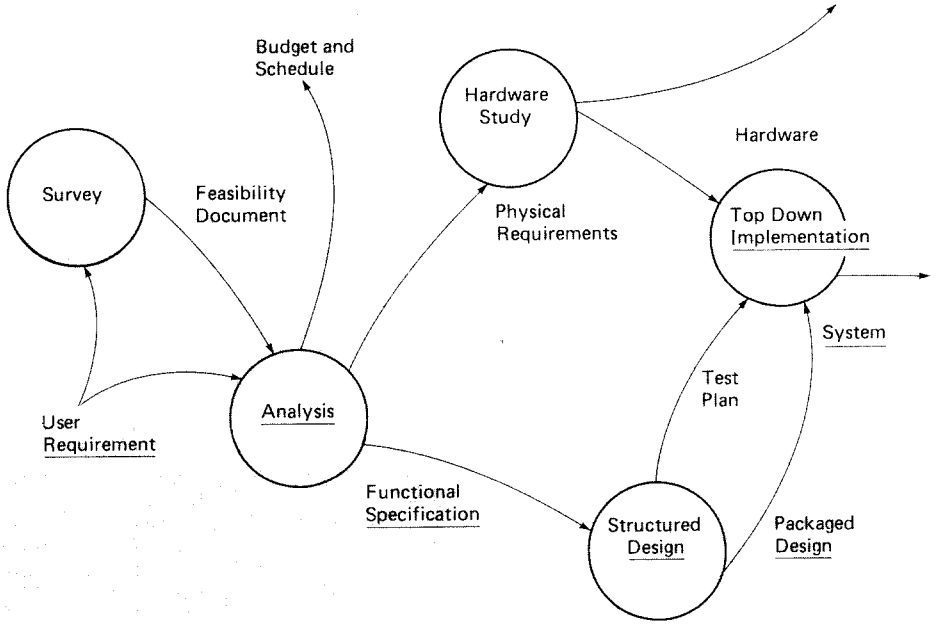

Fig. 6. Software Development Life Cycle (according to DeMarco).

that company x is a potential customer for product y if x has some type of work z and product y is suitable for z. A systems analyst would express this in terms of processes and data flow between processes and picture these graphically (Fig. 7).
I would like to argue that the data flow diagram is different only in syntax from the logic based language which has been chosen for the Japanese Fifth Generation project. It is equivalent in semantics to a language of rules, a language of conclusions and conditions.

x is a potential customer for product y
if x has work of type z
and y suitable for z.

The title of the diagram explains the purpose for which the processes in the diagram are to be used. In this particular example the purpose is to find products to sell to customers. This constitutes the *conclusion* of the rule, namely that some product y is suitable for the potential customer x. The processes which are represented inside the diagram, which are drawn whithin circles, constitute the *conditions* that have to be satisfied for the conclusion of the rule to hold. The first process finds some type of work z which the customer has. The second finds a product y which is suitable for z.
This example shows that rule-based, logic-based programming is not necessarily programming, or even formal specification. In this example rule-based programming is an *executable analysis* of the user requirement. Therefore it can assist the conventional software development life cycle at the earliest possible stage. The user requirement can be analysed and executed before we derive a functional specification, design, or program. We can execute the analysis to see whether it conforms to the user's view of the requirement; and therefore we can eliminate misunderstandings at the earliest possible stage, before they give rise to further misunderstandings.
How is it that we can execute such rules? Rules give rise to procedures. If we know the potential customer x as in the data flow diagram and we want to find something to sell him, then the procedure we obtain by using that rule backwards in a targetted, goal-directed fashion reduces the problem to two subproblems: Find what kind of work the customer has and find something that is suitable for that work. That at least is one procedure. It is a procedure expressed in human terms, which reduces problems to subproblems until eventually they need no further reduction. I can communicate such a procedure to a salesman who might not care about computers at all. Moreover, as far as computers are concerned, the two subproblems can be solved sequentially on a von Neumann computer or they can be solved in parallel on a Fifth Generation Computer in the future.
But there is more to it than that; there is more than one procedure here. The data flow diagram has done disservice to the knowledge. It's not simply that this is a procedure which takes a customer and finds something to sell him. The same knowledge can be used to find customers to sell particular products. If we want to find a customer x to sell product y, find out what type of work the product can be used for and find some customer who has that kind of work. The knowledge can be used more flexibly than the systems analyst has seen and more flexibly than the user has required. What's wrong with software engineering this instance is that there is more knowledge hidden away in the user than simply his perception of the user requirement.
Structured systems analysis has its *strengths*; and the use of new software technology to execute systems analysis adds to those strengths. Among its strengths are the fact that data flow diagrams themselves are a convenient, graphical language for communicating with users. They are sufficiently precise for the systems analyst to express what the user thinks he requires – so precise in fact that they can be translated automatically into rules which execute as procedures. Dataflow diagrams also provide a powerful tool for controlling scale and complexity.
DeMarco's rule is that you limit the size of a data flow diagram to a single sheet of standard sized paper (A4

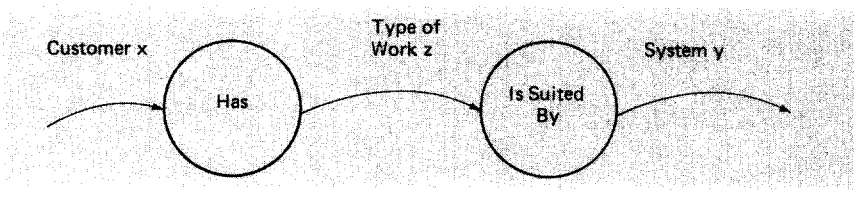

Fig. 7. *X* is Potential Customer for System *Y*.

in the metric system). As soon as you need to go outside the sheet of paper you expand some process by means of a lower-level data flow diagram on another sheet of paper. I doubt whether Software Engineering has any much better solution to the problem of controlling complexity than that.

What are its *weaknesses*? Users don't know what they want; and often, when they do, they don't need what they want. So we have to determine what is the case by starting from what users believe to be the case, and from what problems they think they have, and abstract to find out what knowledge is locked inside. I believe that the declarative form in which knowledge can be expressed using AI software technologies gives us a way of liberating users from their mistaken conceptions of their problems and of using the knowledge they have for bigger and better purposes.

What are some of the solutions for the weaknesses of structured systems analysis proposed by such critics as James Martin? Perhaps the most popular is *rapid prototyping*. You prototype the solution to your problem as quickly and early as you can in the software life cycle. But how do you implement the prototype? In most cases, with a programming language, which was designed for the final stage of software development. The new logic-based software technology allows us to implement prototypes using languages designed for systems analysis, the first stage of software development after the preliminary feasibility study.

If you don't use a programming language you use fourth generation program generators. And what are they? In many cases they are simply generic, parameterized programs which can be tailored for a particular application by the user himself selecting a particular combination of answers to a predetermined menu of options. In other cases they are based upon the database approach. So let's look at databases.

Increasingly, throughout the international database research community, the relational approach is beginning to be subsumed by the logic base approach, an approach which is very closely related to rule-based, logic-based programming. This can be illustrated again by our rule relating potential customers to products. The rule can be regarded as a query generator. Given a problem of relating customers to products, it generates the query:

"find some type of work z for the customer x
for which the product y is suitable".

In relational database terminology, the two conditions of the rule are joined together by the relational join operator. But this is not simply a query to a conventional relational database, where all knowledge is stored explicitly in the form of tables, but rather it is a query whose conditions are evaluated by other rules (or, equivalently, by other procedures). Thus rules behave as procedures which generate queries and those queries are answered by being targeted to other rules which generate queries in turn, etc.

Thus rule-based, logic-based software technology unifies executable systems analysis with data bases containing rules as well as conventional, explicitly stored data. But what does this new software technology have to do with expert systems?

Expert Systems

The following example, which begins very like the preceding example, shows how well suited this technology is to expert systems applications. This example comes from a logical reconstruction by Peter Hammond at Imperial College of an expert system originally implemented in the expert system shell EMYCIN by Peter Alvey at the Imperial Cancer Research Institute in London. The rule starts out in exactly the same way as the rule for selling products to customers, but this time we are concerned with treating patients. The rule expresses that a patient should take some treatment if the patient has a complaint which the treatment suppresses. But with a human patient we are more likely to worry that the cure is not worse than the disease. In its final form, therefore, the rule has one conclusion and three conditions:

x should take y **if** x has complaint z
and y suppresses z
and not y unsuitable for x.

Notice, in this example, another feature of declarative languages: how easy it is to modify knowledge when the knowledge is made explicit. Suppose our first formulation of the rule contained only the first two conditions and therefore made the mistake of treating humans the same as companies. If later we should decide the rule is too wide-sweeping, for whatever reason, we can restrict its application by adding extra conditions. Such changes can be understood and explained in declarative, human terms, without needing to consider their effect on the behaviour of a computer.

Notice too that we are only looking at a top-level rule,

targeted on the goal of relating patients with treatments. We could unravel the conditions of the rule: what does it mean to say that *y* suppresses *z*, what does it mean to say that *y* is not unsuitable for *x*. We could unravel the conditions top down in the same way that structured systems analysis unravels data flow diagrams – but time prevents us from doing more than simply listing some of the lower-level rules.

y is unsuitable for *x* **if** *y* aggravates *u* in *x*
and *x* has condition *u*

aspirin suppresses inflammation
aspirin suppresses pain
etc.

aspirin aggravates peptic ulcer in *x*
lomotil aggravates impaired liver function in *x*
alcohol aggravates high blood pressure in *x*
if *x* is over 40
and *x* is obese.

Thus we can use the new declarative language technology both to implement new expert systems applications as well as to assist the conventional software development life cycle. Or we can do better. We can change the nature of computing itself and the nature of the software methodology which services it. We can make computers understand knowledge expressed in human terms and make them use that knowledge flexibly in different ways for different purposes.

Such computers will solve problems in a manner which approximates human problem-solving and consequently will change the nature of *human-computer interaction*. If the computer needs to solve a problem, it can use its own knowledge to reduce the problem to subproblems or it can ask the user. Why should the computer know everything and do everything itself? It needs to do everything itself only if the human reasons so differently from the computer that the two cannot naturally interact. But if the two are working in harmony within the same problem-solving paradigm, then the human can play an intimate part in the computer-based problem solving process.

The system can *explain* a conclusion by quoting the rules it used to come to its conclusion. You can accept the explanation or reject it. You can use the explanation to reach a different conclusion entirely. It's a common feature of human decision making that we ask peope for advice. We don't simply want their recommendation, we want to hear their argument in support of their conclusion. Having heard that argument, we need to determine whether we agree with it or not, whether we accept the assumptions which justify the conclusion or not. This allows us to stay in control.

Legislation as a Critical Application

Legislation is a particularly critical application, an application which illustrates executable analysis, execution of which is often sufficiently efficient that none of the later stages of conventional software development are required. On the other hand legislation is by no means trivial. It requires complex knowledge representation and reasoning. It is more complex than such typical AI applications as understanding children's stories and expert system for fault diagnosis. So in one respect legislation is a harder domain to tackle. In another respect it is easier.

In AI we are inundated with problems of ambiguity. Researchers in natural language processing seem to welcome ambiguity. In the case of legislation there may be ambiguities; but it is not, or should not, be the intention of the legislator, to put them there. Flexibility, yes; ambiguity, no. For that reason we do not have as much difficulty with looking at practical applications as we do with looking at toy AI natural story understanding problems. It's an ideal domain therefore for tackling hard problems of knowledge representation and problem-solving without being sidetracked by potentially irrelevant issues.

The formalization of legislation also illustrates the incremental method of software development by trial and error. If we were writing programs when we represent the meaning of legislation by trial and error, then we would be bad programmers. Good programmers start with rigid, or at least formal, software specifications and then implement them correctly first time round – never get it wrong. So a PROLOG programmer who is always correcting errors in his programs is a bad programmer. But for a person who is using PROLOG not as a programming language, not even as a formal specification language but as a language for analysing the knowledge that lies behind the user requirement, trial and error is unavoidable. Even mathematicians prove theorems and develop axiomatic theories by trial and error.

But the formalization of rules and regulations is representative of a much wider class of applications. It is applicable whenever an organisation uses rules

to regulate its affairs, whether or not they have legal, binding authority.
What function do regulations serve? Having rules means not having to deal with each problem as it arises, as if no similar problem had arisen in the past. It means deciding what the general rules are, so that different customers are treated equally, applying the same criteria to one as we do to another.
Indeed, the whole concept of rule-based knowledge representation has important *human implications*. When we extract knowledge from experts in the form of rules we see, often for the first time, what the rules really are. The process of eliciting knowledge from experts can be a painful process. It is difficult to know what the expert thinks and what he believes. But this is just as true of normal people. It's hard for us to know what rules we use ourselves in solving day to day problems. If we could articulate them, then we could examine them. Even if our first attempts at articulation were incorrect, we could improve them by trial and error. We could see them for what they are; we could challenge them; and we could see if they are fair, if they apply to one customer as well as to another, to ourselves as well as to others.
The formalization of rules and regulations also illustrates the potential of another application for expert systems technology, different from simply applying known expertise, different from applying the law in individual cases. It illustrates how the trial and error process of formulating regulations can be used as an important tool in developing and improving human expertise – where there is no expert within a given company, for regulating pension schemes, for example. One way to start, is to hypothesize some rules. Instead of trying them out on people, try them out on the computer, in an interactive manner which is based on a common model of deductive problem-solving which is shared by the human and the machine.
Let's look at one or two examples from the *British Nationality Act* and see to what extent they confirm the theory. The very first subsection of the Act is concerned with *acquisition by birth:*

"A person born in the United Kingdom after commencement shall be a British citizen if at the time of birth his father or mother is:
(a) a British citizen; or.....

Notice how the word "if" in the English text occurs almost exactly where it would occur in a rule-based logical representation.
The conclusion of this very first clause of the British Nationality Act is that a person is a British citizen. There are some logical conditions, however, tucked away inside the syntax of the conclusion. One condition is that the person be born in the United Kingdom and the other that he be born after the commencement of the Act, that is to say after the date of which the Act takes effect. The other conditions are explicitly written after the "if". Obviously rule-based knowledge representation provides us with a very natural way of representing such knowledge.

x is a British citizen
if x was born in the U.K.
and x was born on date y
and y is after commencement
and z is a parent of x
and z was a British citizen on date y.

My colleagues, Therese Cory, Peter Hammond, Frank Kriwaczek, Fariba Sadri, Marek Sergot, and I have investigated the representation of the British Nationality Act in PROLOG. About 80% of its approximately 70 odd pages have been written in PROLOG. We found the structure of the Act very difficult to comprehend and so we tried using data flow diagrams to help. We soon came to the reluctant conclusion that data flow diagrams were inadequate for two reasons. First, they required directions on the flow of data, which as in the customer-products example unnecessarily restricted the different ways the rule might be used. Second, it is not easy to represent the logical connections between different processes in a diagram. In the end, we decided to use and-or graphs, a kind of data flow diagram in which logical connections between processes are made explicit, but data flow between processes is ignored. The and-or graph helped to give us an overall view of the structure of the Act, but it gave us little help in deciding detailed knowledge representation issues. Moreover, it soon became clear that there was little alternative to trial and error refinement of the rules. The inadequacy of our first attempt to formalize subsection 1.1.a, in particular, did not come to light until we came to section 2.1.a which is concerned with *acquisition by descent*:

"A person born outside the United Kingdom after commencement shall be a British citizen if at the time of birth his father or mother –
(A) is a British citizen otherwise than by descent; or
...

Notice the disconcerting condition "British citizen

otherwise than by descent". This shows that our earlier assumption that the conclusion of 1.1.a is that

"*x* is a British citizen"

was naive. Moreover, it also ignores the implicit assumption that *x* acquires citizenship at the time of birth. Taking both of these omissions into account, we can revise our original formalization, obtaining the next approximation:

X acquires British citizenship by 1.1.a on date *y*
if *x* was born in the U.K.
and *x* was born on date *y*
and *y* is after commencement
and *z* is a parent of *x*
and *z* is a British citizen
by *w* on date *y*.

Subsection 2.1.a can be represented similarly:
x acquires British citizenship by 2.1.a on date *y*
if *x* was born outside U.K.
and *x* was born on date *y*
and *y* is after commencement
and *z* is a parent of *x*
and *z* is a British citizen by *v* on date *y*
and *v* is **not** by descent.

However in both of these rules there is a mismatch between the form in which citizenship is expressed in the conclusion and the form in which it is expressed in the condition. We need an additional rule which is not explicitly stated in the Act, but which is taken for granted:

x is a British citizen by *w* on date *y*
if *x* acquires British citizenship by *w* on date *z*
and *y* is after *z*
and *x* is alive on date *y*
and *x* has not renounced British citizenship before date *y*
and *x* has not been deprived of British citizenship before date *y*.

In other words, a person is a British citizen of a particular kind on a particular date if he/she acquired that citizenship on an earlier date, is alive, has not renounced it and has not been deprived of it.
The less obvious situation where a person who has died might be regarded as a British citizen after death is dealt with explicitly in subsection 48: A parent who is no longer alive at the time of birth of his child is regarded as being a British citizen at the time of birth, if he was a British citizen when he died.
These rules illustrate some of the top level of the British Nationality Act. The conditions which occur in these and other rules can be satisfied in a variety of ways.
Conditions can be *defined by rules*. For example, the condition

"*z* is settled in the U.K. on date *y*"

which is a condition of 1.1.b is defined in subsections 50.2, 50.3 and 50.4; and its definition is naturally represented by means of rules.
Conditions can by *defined by data*. For example, the condition

"*z* is a British dependent territory"

is defined by a list of territories enumerated in schedule 6. Conceptually, for every territory there is an assertion, e.g.

"Gibraltar is a British dependent territory."

Each such assertion can be regarded as a trivial rule having one conclusion and no conditions.
Conditions can be *computed by programs*. For example,

"*y* is after commencement."

But any program is a procedure or collection of procedures which can be represented by rules which are used backwards to reduce problems to subproblems.
Conditions can be solved by *querying the user*. For example,

"*x* was born on date *y*"

In general, any condition can be solved either by the computer or by the human user. The computer can recognise that it is unable to solve a given problem and can therefore automatically request a solution from the user.
Conditions can be solved by *querying an expert*. For example, the condition

"*x* is ordinarily resident in the U.K. on date *y*"

is not defined in the Act, but is decided by the Secretary of State. In the absence of access to the Secretary of State, the system would need to consult

an expert, either a human expert or an expert system.
The rule-based formalization of the British Nationality Act by trial and error exemplifies the use of declarative language technology for an application which has both conventional and novel characteristics. On the one hand, if we restrict ourselves to problems of determining citizenship, it is not very different from a complicated data processing application.
On the other hand, given appropriate inference machinery, the same representation can, at least in theory, be used to generate and test arbitrary logical consequences of the Act. In both cases we have short-circuited the conventional software development life cycle, completing it without leaving the executable systems analysis stage.

Other Applications of Artificial Intelligence to Software Engineering

So far I have concentrated attention on those relationships between Artificial Intelligence and Software Engineering which appeal to me most – applications of AI technology which revolutionize the software life cycle, which in many cases altogether do away with program implementation, and even system specification. There are of course other applications of AI technology, and they are the ones the software engineer might prefer to draw to our attention: intelligent tools which help to preserve the conventional software engineering process; intelligent front-ends to otherwise inscrutible conventional computer programs; knowledge bases to support the conventional software process; expert systems which incorporate the conventional software engineering expertise. Don't worry about the way the software engineer ought to work; see how he does work and develop intelligent tools which help him to do what he already does better. In my opinion, cognitive psychology makes a similar mistake. Don't worry about developing better ways to do better things. Take people the way they are; and design computers to simulate them.
Such applications of Artificial Intelligence technology have their place, especially if they are the only way we can convince the Software Engineer to experiment with AI technology. But let's not devote all of our resources to helping the old software methodology live longer.
There are other applications of AI to SE, which I have not talked about, but which have great present value and future potential. The formal, computer-assisted derivation of programs from specifications, in particular, is an area which straddles the fields of Artificial Intelligence and Software Engineering. It is needed if an executable system analysis does not perform efficiently enough to meet the user's performance targets. This was not the case, for the most part, with our formalization of the British Nationality Act, although even there we used program transformation techniques, by hand, to eliminate certain loops.
In many other cases, such as sorting files for example, executing an analysis of the user's problem domain isn't sufficient. We need to improve efficiency by restricting the class of problems to be solved and restricting knowledge so that it is directed to that class. This changes the systems analysis into an executable system specification, still written in the same logic-based language. But now the specification has an appearance of formality and rigour, which is more apparent than real, because syntactically there is no difference between it and the empirically derived systems analysis.
If the executable logic-based specification is still not sufficiently efficient, it can be transformed further into a more efficient program. If necessary, the program can be written in a conventional programming language. But given adequate software and hardware resources, it can also be transformed into a program expressed in the same rule-based, logic-based language. Using the same language for all stages of the software development process greatly simplifies the problems of maintaining consistency between the different stages. Moreover, transformation and derivation techniques which are guaranteed to preserve correctness can be used to pass from one stage to the next. Such techniques have been developed within the community of declarative language researchers who live within the intersection of AI and SE.
I have talked about the applications of AI to SE. What are the applications of SE to AI? Certainly the Software Engineer has three major concerns which do not always attract sufficient attention in AI:

correctness,
scale and
complexity.

I have already argued that many AI applications are better thought of as executable analyses or executable specifications. To the extent that that is the case, such applications are as correct as any conventional

systems analysis or specification. However, many AI applications go beyond analysis and specification in the extent to which they are concerned with matters of efficiency. In such cases, the resulting programs are as much in need of validation and verification as any conventional program. The Software Engineer is right to criticize the AI programmer who uses AI techniques which do not have logical foundations, and are not amenable to proof.

This is an area in which logic-based approaches to knowledge representation and programming in AI have a distinct advantage over other approaches such as frames and object-oriented programming. Knowledge representations and programs expressed in logic are expressed in the same formalism as the software engineer uses for expressing formal specifications. Using the same logic-based language for both programs and specifications significantly simplifies the problems of proving correctness.

What about scale and complexity? I wonder whether there is very much more to be said other than to repeat deMarco's advice about not using more than a single sheet of paper for a single data flow diagram (or the equivalent collection of rules, whether they represent an analysis, specification or program). It may be, however, that frames and object-oriented programming have some useful contributions to make here. If so, then I believe they would need to be integrated with logic-based approaches, probably along the lines suggested by Pat Hayes in his paper on the Logic of Frames.

Human Implications

I would now like to address some of the human implications of new technology.

I don't believe that technology for technology's sake will always be good. I believe that the technology of knowledge-based software is going to make life better on the average. But, unless we are aware of some of the potential dangers and take suitable precautions, there may be some spectacular undesirable results.

It is all too easy to let computers take over. It's all too easy to let the computer decide. We've done it before. We do it with humans, with professional advisors. "Let the doctor tell me what to do." "Let the accountant decide how to run my financial affairs." The human expert can intimidate us by knowing more than we do and by having greater expertise. If humans can do that with humans, then computers will be able to do that with humans too; and they will do it, if we allow the enthusiastic technologist to have his way. The enthusiastic technologist will inevitably design computers to do more and more of our thinking and decision-making for us. They have done it with television already. We can't entertain ourselves without technology any more. We enjoy ourselves more sitting in front of the television than we do interacting with live people. The same will happen with computers unless we are determined to prevent it.

Computers are possibly the most useful of all technologies for aiding the disabled. They can help people who are handicapped and significantly improve their ability to deal with the world. But those same facilities which can assist the handicapped can also assist and potentially disable the able-bodied person, whether he needs assistance or not.

I see real dangers, but on the average I see great potential benefits. The new computing technology has some obvious uses for implementing intelligent front-ends, not just for conventional software, but for any kind of unfriendly machinery – my oven, for example. I hate my oven. I don't know how to use it properly and it doesn't know how to take advantage of my ability to cook food. The intermediary of something which is more machine-like than me and therefore more sympathetic to my oven than me, yet which understands the world more like I do than computers do today, can make the world of machinery more friendly and more understandable.

The new rule-based, logic-based languages allow us to get rid of the "take it or leave it" attitude of computers today. They make it possible for computers to explain their conclusions, and therefore easier for us to decide for ourselves whether to accept their conclusions. Only when computer programs are expressed in declarative, explicit form, can we identify what assumptions they use, can we decide whether to accept their assumptions and therefore whether to accept their conclusions.

Obviously such computers will increase human productivity. Every economic activity can be performed more productively.

Such computers can also increase human knowledge and expertise. Computerised encyclopaedias have already begun to give us ready access to everything that is already known. Through the technique of knowledge elicitation, things that are only known unconsciously can begin to be articulated and brought out into the open. In the same way that an expert system might give us a better understanding of a medical expert's previously unconscious knowledge and beliefs, knowledge elicitation can give us a

better understanding not only of experts, but of common people.

Not only knowledge but also human reasoning and human rationality can be enhanced. Once knowledge is made explicit, we can see more clearly what we believe. We can begin to see what others believe. We can begin to see the individual steps that explain and justify knowledge and belief. We can begin to think more rationally, because we can better understand ourselves and others. We can suspend our beliefs because we know what they are. We can temporarily assume another's beliefs because we can have a hypothesis about what they may be; and we can reason with those assumptions to see where they lead. I believe that, on the average, this will lead to a better world.

In conclusion then, let me summarise. I believe that the mechanisation of logic, the same dream that Leibniz had, logic machines, will make computing better, and is therefore the key to new generation computing. It is the link also between knowledge representation languages in Artificial Intelligence and systems analysis languages, program specification languages, and database languages in Software Engineering.

But in the end what matters is not computers, or Software Engineering, or Artificial Intelligence, but people. And, provided we take the right precautions, I believe the new technology will help us to be more human, to understand ourselves, and to understand others.