Artificial Intelligence Review (1987) 1, 255–273

Artificial intelligence and software engineering: a tutorial introduction to their relationship

L. Ford
Department of Computer Science, University of Exeter, Exeter EX4 4PT, UK

Abstract This article is a tutorial introduction to artificial intelligence for software engineers, and a similar introduction to software engineering for artificial intelligence workers. Software engineering and artificial intelligence are compared and contrasted in terms of the problems they attempt to solve, the methods they employ, and the tools and techniques that are used. It is argued that a fusion of the two disciplines will be needed for many new software demands. The evidence for this is examined briefly and some of the steps that are needed for an alliance of the two disciplines are mentioned.

Introduction

This tutorial aims to provide an introduction to software engineering (SE) for those readers with primarily an artificial intelligence (AI) background, and a similar introduction to AI for professional software engineers. Both AI and SE are disciplines of Computer Science and as such have some common themes. These are explored, as are some of their essential differences. The two respective communities of software developers often appear to follow divergent paths, perhaps through a misunderstanding of what each is trying to achieve or through a professional jealousy and mistrust, but in either case it is not a healthy situation for Computer Science as a whole which will, I believe, come to rely on a fusion of the two disciplines to realize its full potential in a satisfactory way.

AI has developed models, notations, techniques, and a methodology that could be useful to SE, while the latter has provided both formal and informal methods that could contribute to the development of AI software. (It should be mentioned that some formal methods of SE, in particular program proving, do not yet lend themselves to the construction of large-scale practical software, whereas SE's informal methods that embody sound methodological principles, such as the collection known as Structured Programming, are in general use in SE and could arguably be used in the development of some systems labelled 'expert'.)

Each, therefore, potentially has something to offer the other that I hope the enquiring reader will not be too biased to ignore.

Computer Science is the study of computing machinery as well as the general process of computation, and the nature of the underlying machinery — the hardware — has played an important role in SE and could play an even more crucial part in the realization of 'intelligent' machine behaviour. The topic is not, however, covered here. The interested reader is referred to Bishop (1986) who provides a comprehensive overview of new architectures and machines.

The next section compares the problems addressed by AI and SE. Following on from this the nature of AI and SE solutions are examined to bring out their critical differences. The development methodologies of the two disciplines are then looked at, in some detail. The tools, techniques, and languages of AI and SE are also briefly surveyed. Finally some speculations on future trends, that presage an alliance of the two disciplines, are given.

Problems addressed by AI and SE

There are no hard and fast definitions of AI or SE problems such that a given problem can be categorized as belonging exclusively to the province of one or other discipline. Both disciplines are concerned with methodologies, tools, and techniques that can serve to provide solutions to many of the same or similar problems.

A common claim for AI is that it addresses problems which, hitherto, have only been solvable by intelligent humans. There are a number of difficulties with this definition, not least being that it suggests a constant movement of 'solved' AI problems to some other discipline, perhaps SE. Thus problems solved before the advent of AI, such as the first invoicing program are, on this reckoning, prehistoric ancestors of AI. And AI's current favourite offspring, the expert system, for which a methodology, tools, and techniques have been laid down, presumably falls within the province of software engineering. (Campbell (1984) suggests other problems with this definition and explores the general problem of what is and is not AI.)

Although there are no accepted definitions of AI most would agree, without invalidating the earlier observation, that the two disciplines can address the same problems, that an SE treatment of some problems is more appropriate than an AI one, and vice versa. This raises the question of how problems can be characterized to enable the appropriate development discipline to be determined.

Data in the real world are rarely reliable — that is they are rarely complete, consistent, without error, and unmasked by extraneous data. For some problems it is feasible and advantageous to remove extraneous data prior to input to a computer program. Thus a clerk receiving an order for goods from a client could, when completing a computer order input form, ignore the client's request for urgent delivery (since a condition of the system may be that all orders are to be processed in the same way irrespective of delivery requirements). The clerk could also attempt to ensure that the client's data were complete, by entering information in all essential fields of the input form. This would allow a computer input program to validate the data in a relatively simple and systematic way such that programs subsequently processing validated data can assume them to be reliable. Another feature of the data received by the clerk is that they vary over time; the data received

by the input program, however, vary by place. Thus the clerk must be capable of dealing with time-varying situations, e.g. each utterance by the client presents a new situation that has to be dealt with, whereas a program validating input data prepared by a clerk needs only to examine the pre-defined places on an order transaction to find the data it requires. Data in the latter form are said to be static; time-varying data are dynamic.

Let us now imagine a computer system replacing the function of the clerk. It must cope with unreliable and dynamic data. Such a system is bounded and can be pre-specified only to the extent to which these requirements can be pre-defined. As the sensitivity to a client increases, however, the complexity of defining all the possibilities of unreliability and event sequences also increases — and not linearly. This gives rise to the combinatorial explosion of possible situations and increasingly reduces the possibility of pre-specifying them. The space of possible solutions to these situations can be of such size that it becomes more convenient to consider solution methods rather than particular solutions for each situation. That is, the task of providing a solution to such a problem is more manageable when emphasis is placed on the way a solution can be found rather than what a solution comprises.

The methods and tools of software engineering are suited to problems that can be characterized as having reliable, static data for which the problem solution space is small. Problems with unreliable and dynamic data usually imply a much larger solution space; AI techniques have been developed to deal with this situation. To repeat, however, there is no reason why either discipline should not tackle the problems that, on the above account, seem more suited for solution by the other. SE can, for example, deal with large solution spaces which can be factored, and AI is able to provide solutions to problems with reliable, static data and small solution spaces. It will become evident in the following sections, however, that SE has fashioned its methods and tools to suit the problems that have been characterized for it above, while AI, in the process of undertaking a similar exercise, has largely ignored the possibility of refining its tools to meet characteristic SE problems.

The problems addressed by software engineers cover those applications associated with data processing (DP). These include: order processing, invoicing, inventory control, management accounting, etc. A feature of these applications is the high volume of relatively few types of input data, and the static non time-varying nature of the data. It is true that most existing applications are the result of a software engineering process. It is also fair to say that few applications of AI can properly be regarded as practical software. Most AI work, to date, is of an experimental nature, often addressing small, paradigmatic problems in order to increase the stock of knowledge about intelligence. The most promising fruits yielded by AI work, with respect to practical applicability, are expert systems. These systems attempt to exhibit expert performance in difficult problem domains, e.g. diagnosis, planning, and design. There are systems in regular use which help to configure computers, aid in chip design, and diagnose circuit faults. Although production of systems like these is increasing there are many problems associated with their development and acceptance which will become apparent later in this article.

The nature of solutions

The nature of AI solutions to problems is usually distinctive from SE solutions. Consider two types of problem—interpretation and diagnosis—that have been tackled in one form or another by software engineers and AI practitioners.

Interpretation problems involve data analysis to support some subsequent processing. This type of problem is prevalent in DP systems where analysis of order transactions may lead to production of invoices or sales reports. A number of features of such transactions provide the key to distinguishing an AI interpretation problem from those found in software engineered DP systems. In the latter, transactions are well-defined, assumed to be correct after validation checks, and are not contradictory in the sense that one transaction is inconsistent with another. As such, a program processing them can have a completely specified transaction record definition that will cater for all input. AI interpretation problems, on the other hand, e.g. *DENDRAL* (Buchanan and Feigenbaum, 1978), need to find consistent, correct, and rigorously complete interpretations of data that themselves are not complete, perhaps contradictory, and noisy, i.e. containing extraneous data. Whereas the SE solution to an interpretation problem can be regarded as correct (since the details of a transaction map exactly onto its record definition) an AI solution can usually only be regarded as adequate or inadequate because assumptions will have to be made by the program about the meaning of the data.

Input validation components of SE systems can loosely be regarded as examples of solutions to diagnosis problems—they find fault with data. The fault-finding, however, is strictly limited to detecting faults without providing a reason for them. The definition of a fault is also well-defined, perhaps expressed in terms of ranges of valid values. The most celebrated AI diagnosis program, *MYCIN* (Shortliffe, 1976), diagnoses the infectious disease bacteraemia and selects an appropriate antibiotic therapy for it. There are a number of difficulties with this problem that a solution must address. First, there are the problems associated with interpretation of data, all of which may not be available. In addition some symptoms may be masked by others. A major point is that the causes of the 'fault', and hence diagnosis of the disease, are not completely understood from a medical point of view. This means that a program solution has to cope with an incomplete specification of the problem. Thus the solution contains statements of the form:

A suggests B;
C and D tend to rule out E;

which clearly reflect incompleteness. The solution could not, therefore, be regarded as static, in the way many SE solutions are, with respect to their specification, but rather dynamic and susceptible to improvement as medical knowledge increases. The AI solution is thus not amenable to definition in abstract terms. Rather it needs its solution statements to be of the types above which are heuristic. As a consequence of this it is unlikely that a solution to one AI problem can be carried across to form the basis of the solution to another and similar problem. The context-sensitivity of its statements, as shown above, prohibit this. However, numerous

software packages for accounting, invoicing and so on, demonstrate the possibility of having context-free solutions to SE problems.

The fact that solutions to SE problems are invariably cast in abstract terms has important consequences for their implementation. It permits a problem to be regarded as consisting of a number of sub-problems, each of which can largely be considered in isolation from the others. It also facilitates the use of repetitive constructs, or repeat statements, in a solution that are so desirable for many DP problems. An SE solution can thus have a structure that closely resembles the structure of the problem it is to solve.

AI problems are rarely conducive to the 'divide and conquer' treatment mentioned above; the solution space is often not factorable because of the interaction among 'conceptual' sub-problems. AI solutions, therefore, have to prune the search space of solutions 'on the fly', rather than have it done for them in advance by the programmer as with conventional programs. In general, AI programs prune the search space by using factual and heuristic knowledge about the domain, e.g. the rule forms above, and these require an inferential processing mechanism rather than the repetitive one of SE solutions.

It has already been suggested that many SE solutions take large volumes of data as input and manipulate them in subsequent processing. The effective manipulation of data is evident in many SE solutions which have well-defined interfaces to databases. AI programs are usually more concerned with smaller input volumes with data couched in a symbolic form, e.g. natural language. Symbol manipulation is thus a feature of the input components of AI solutions, and indeed of other components, since knowledge is itself primarily expressed in symbolic form. It is, however, interesting to note the comment of Doyle (1985) that "the gut feeling of experienced applications developers that *LISP, PROLOG*, and fancy AI representational languages are irrelevant is right—strictly speaking, as far as the end result is concerned" since it suggests that an AI solution could be reformulated as a conventional algorithm recognizable, as such, by a software engineer. It is during the formulation stage of a solution that the AI programmer needs to express information symbolically since at this stage he is unable to understand the problem sufficiently to enable him to think of it in abstract terms.

Methods

Before any comparison of the methods of SE and AI are made distinctions should be drawn between the different methods that are used within each discipline.

In SE there is a difference between the formal methods that are taught to students of Computer Science and those that are used to build practical large-scale systems. In addition, students who develop their own experimental software will not usually need to apply all the stages of development associated with practical software because some stages, e.g. those that envisage interaction with a client, may not be appropriate.

In AI a distinction is drawn between the methods used to develop experimental software and those used to develop practical AI software.

The main SE method for developing practical software is first described. (This section can be omitted by the software engineer without any undue loss of continuity.) This is followed by a description of the experimental method for AI software with some additional notes on the production of practical AI software. (Even AI practitioners may wish to read this since it provides some background to the present debate on AI methodology and the way in which it differs from the formal and informal methods of SE (see Partridge and Wilks (1987) for a recent contribution to the debate).

Software development lifecycle

Nowadays conventional software that has practical application is mainly the result of a software development life cycle (SDLC) that consists of a number of sequential stages, each with its own specific end product. Although many different versions of the SDLC are used by the SE community, some as products, others as an informally agreed set of methodologies or philosophies, they nevertheless broadly subscribe to the idea of sequential stages of development, each of which provides input to the next stage. So although the stages presented in the SDLC that follows may differ from one organization to another, or from a large project to a small one, they are, nevertheless, to be found in one form or another in the majority of, if not all, system development projects. The names of the stages may be different, and the boundaries between them may alter, but the set of tasks contained within them should remain the same.

The particular SDLC suggested here was formulated by King (1984). It consists of seven stages:

1 feasibility study,
2 requirements definition,
3 system specification,
4 system design,
5 program design and development,
6 system test,
7 implementation and production.

A glance at this list suggests an orderly progression from one stage to another, perhaps involving many people with different roles and skills, and the question arises whether the overhead involved in managing the SDLC is worthwhile. Most DP projects needing more than six man-months effort, however, are probably in need of it. For smaller projects though, it is often possible, with the right mix of people and skills, to condense the SDLC.

Whatever SDLC scheme is used, the people involved, whether managers, end-users, or programmers, are all engaged in the activity of SE.

The objective of the first stage of the full SDLC is to demonstrate that the proposed system, in its as yet ill-defined form, is practical. Because the feasibility study is usually, through necessity, carried out before full financial approval has been gained, and there is the possibility that it may indicate the impracticality of

the proposed system, it is often the shortest and least expensive stage of the SDLC. Its importance, however, is crucial. The history of SE is littered with systems that have failed at the final production stage through inadequate attention to it. Initiation of the feasibility study usually comes from the user department, or organization, which provides a brief description of the issue to be resolved. Output from it should contain: a brief description of the proposed system; a characterization of the system type, e.g. batch or on-line, file or database; possible hardware and software types; a cost-benefit analysis; and a tentative project plan.

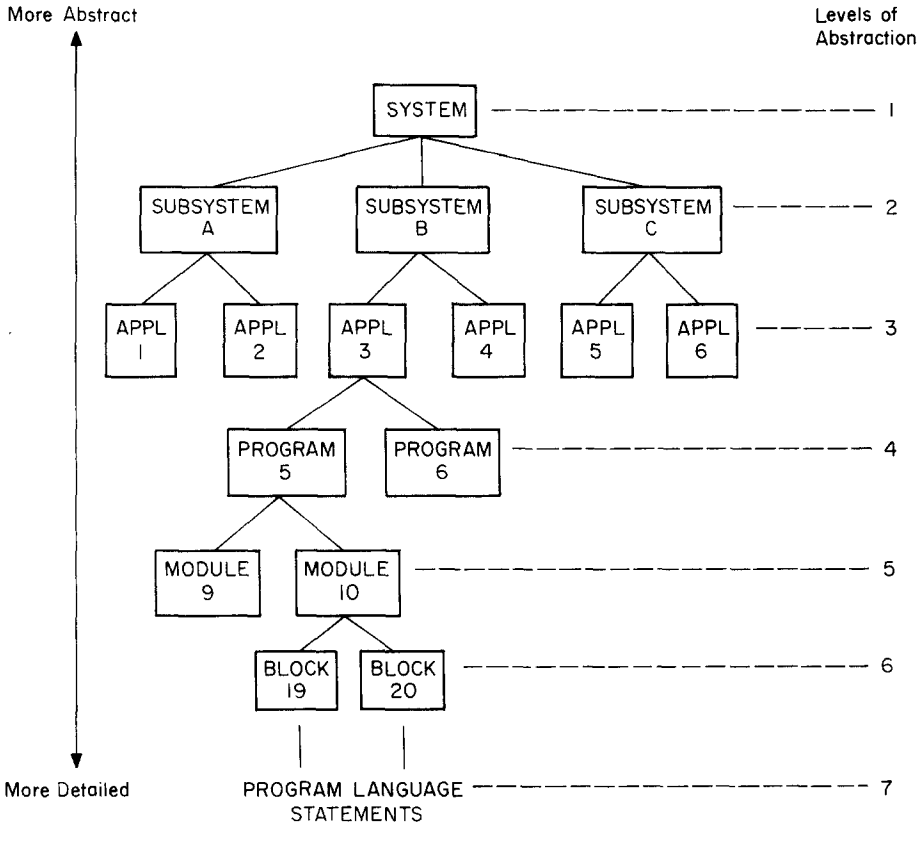
The requirements definition stage elaborates the proposed system, from the feasibility study to an accurate and complete set of user requirements. Where appropriate a detailed analysis of the existing equivalent system, manual or automated, is necessary for this. Output from this stage, usually in document form, would provide:

—an analysis of the current system,
—a set of user requirements,
—a summary of the proposed system,
—cost, resource, and time estimates.

The aim of the system specification stage is to define a system that meets the formulated user requirements. Translation of user requirements, to a specification, involves defining system data requirements in detail. This is done by refining the data information from the previous stage and by using structure charts and data flow diagrams. A system data dictionary containing all data elements and their relationships provides the formal expression of this information. Where databases are already being used by the client any necessary interfaces to them will need consideration. The specification document will also provide a description of the system in technical terms, network and telecommunication requirements, a description of system controls, e.g. for data and process security and recovery, a revised cost-benefit analysis (which would of course highlight any significant deviation from the one produced at the feasibility stage), and a further refinement to the resource plan. At the end of this stage it is customary to have a full review by interested parties, e.g. users, controllers, auditors, and operations staff.

King (1984) states: "the objective of the system design stage is to produce a detailed, technical, logical definition of the final system, so that the final set of programs can be produced from that definition" and uses the term 'logical definition' to indicate what the system does rather than how the system does it. A useful framework with which to append this detail is a hierarchical structure of the system of the type shown in Fig. 1 below.

The design stage, in practice, will only take the structure down to level 4: the next stage refines level 4 to level 7. The process of producing such a structure is variously called; top-down design, functional decomposition, and structured design. These terms were borrowed from programming methodology in recognition of the fact that system design and program design activities each have a need to manage the complexity of a problem through abstraction. Indeed two notations for program refinement—Jackson diagrams and Warnier–Orr diagrams—have each



been used at the system design as well as the program design and development stage. As well as the convenience of separating various concerns, the intention of top-down design is to enable the designer to satisfy himself that each new level of abstraction is a solution to the level above it. Clearly this averts the possibility of the results from the labour-intensive activity at level 7 failing to meet the requirements envisaged at level 1. Apart from providing a detailed system description based on the hierarchic framework, the system design stage will also refine the system controls and revise the cost-benefit analysis from the previous stage. In addition, it may recommend various design alternatives for review, and program design techniques and standards to be used at the next stage. Finally it should provide a preliminary system test plan.

The program design and development stage provides the flesh for levels 5–7 shown in Fig. 1. It involves not only programming, i.e. designing, developing, and testing program solutions, but also, in collaboration with users and other departments, preparing user and operator manuals, and providing documentation. Various techniques are available to the programmer which enable him to derive a solution, the two most common are Jackson Design Methodology (JDM) and Warnier–Orr Design Methodology. Each is based on the premise that a program structure can be derived directly from the structure of the data that a program will

process. A feature of JDM is the clear procedure—a programmer must follow the same five steps for each program design. This helps to produce consistent design structures, understandable to others as well as the programmers who create them.

Individual program testing and link testing (sometimes called integration testing) are completed prior to the next stage—system testing. Whereas program and link testing may be designed and undertaken by program authors, hence by people who are intimate with a program's working, system testing is undertaken firstly, on behalf of senior development staff who should have no knowledge of individual programs (usually referred to as black-box testing), and secondly, on behalf of users, i.e. the user acceptance test. This latter test is not to determine whether the user's requirements have changed since the inception of the project, but to find out whether the system meets the requirements that were originally agreed with the user.

The final stage, implementation and production, involves preparing users and operators for the installation of the system. Once installed it would be comforting to believe that the system would settle into a stable period of production. Nothing could be further from the truth. It is most likely that errors will be found in the system—either errors of logic or errors resulting from misunderstanding—and that user requirements, sensitive as they are to external and uncontrollable situations, will not change. Maintenance of computer systems (including enhancement) is reckoned to be the major part of the total system life cycle cost. The SDLC has evolved over the years in recognition of this, and much of the documentation produced during the first six stages is designed to meet post-implementation requirements, as much as the initial development activity.

AI methodology

AI has yet to evolve a methodology suited to the production of practical software, largely because most AI software has been of an experimental nature, and thus not developed in a disciplined way to satisfy the needs of users, managers, and others one stage downstream in the lifecycle, but also because it has been unable to draw from the methodological foundations provided by SE. These latter have been fashioned to accord with the nature of SE problems that were presented earlier—namely, that problems are bounded and can be completely specified—and are not easily adapted to meet the messier problems of AI.

The first three stages of the SDLC present particular problems to AI. How can a system be specified in such a way that it leads to the systematic development of a solution when the nature of AI problems is a lack of specificity? Recall that AI problems were characterized as having to deal with a very large number of different situations, not all of which could be predicted. The requirements definition for problems of this sort could not be provided with the desired precision. Consequently it becomes impossible to undertake a feasibility study with any confidence since the study itself relies largely on envisaging the practicality of the requirements definition and system specification stages.

How then is AI software developed? First, a distinction must be drawn between experimental software, developed primarily to investigate the problems of implementing machine intelligence or to test some thesis of human intelligence, and the software developed with practical day-to-day usage in mind. Partridge (1986) has described a run-understand-debug-edit (RUDE) paradigm to describe the former (although he suggests that there is a need to develop some disciplined version of it to support the possibility of practical AI software). RUDE recognizes the impossibility of providing a complete specification before programming commences and has, as a first stage, the running of a machine-executable specification that is a first approximation to the solution of a poorly understood problem. From the run, major inadequacies are uncovered in the understand and debug stages which are followed by an edit stage to provide a closer approximation. The cycle is recommenced and iterated until an adequate solution is found. Such a scheme may remind the software engineer of the halcyon days of twenty years ago when programs were designed very informally with no user or manager in sight. There is, however, one crucial difference. The AI worker is forced into this solution, or something like it, because of the nature of the problem he is tackling, whereas the software engineer, or programmer as he was then called, first created, not an approximation to the solution but, a complete solution which, with some debugging, met his original design concept.

A scheme rather like the RUDE cycle is witnessed in the early stages of SDLC. Rapid prototyping, as it is called, is a process of speedily providing a performance-mode description of user requirements and goes some way to forming a system specification. Its main use, however, is restricted to one-off applications which need not be integrated with current systems. Most expert systems, some examples of which can be regarded as practical software, are also rarely integrated with other systems. Development of these systems provides the basis of an AI methodology with some similarities to SDLC.

An expert system must have the following properties (Waterman, 1986): expertise, symbolic reasoning, depth, and self-knowledge. Expertise means the system should exhibit expert performance, have a high level of skill, and be reasonably robust. Symbolic reasoning is required to deal with knowledge, expressed as symbols, so the large solution spaces which characterize AI problems can be pruned. An expert system has depth in the sense that it operates effectively in a narrow domain containing difficult problems. The knowledge itself, however, is frequently not deep and contains 'compiled' or surface forms of deeper knowledge. This lack of depth is a common criticism of current expert systems since it effectively precludes them from justifying their decisions and indeed justifying the particular knowledge contained within them. (Software engineers may be interested to learn that pioneering AI work tackled the breadth problem by implementing general problem solving programs. This work was largely unsuccessful although it had the merit of emphasizing the enormous difficulty in producing common-sense behaviour that you or I would take for granted.) Self-knowledge is important to an expert system because it enables a system to reason with its own operation in order to check the plausibility of its conclusions. Furthermore, this

self-knowledge can be used to provide the end-user with an explanation of what a system is doing and why which, since an expert system is prone to making mistakes like a human expert, is crucial to their acceptance. Expert advice, whether emanating from a human or computer system, need not be taken after all.

Given the desire of a user organization to employ an expert system in some role, a feasibility study can be undertaken to examine the practicality of producing such a system provided there is some historical evidence for developing an expert system in a similar domain. Without this, a feasibility study would have little credibility because the practicality of developing the system relies heavily on the possibility of (a) being able to acquire specific domain knowledge, and (b) that the knowledge, when acquired, can be formulated so it is amenable to symbolic reasoning. Prior success with the domain would count as a strong indicator of feasibility. No previous success would label the project as experimental.

The first main task of building an expert system, once the domain has been identified and concerns about the feasibility of the project have been satisfied, is knowledge engineering. Initially this involves selection of hardware and tools to support a cyclic process of knowledge acquisition about the domain, its representation, and verification. Knowledge acquisition is the process of building the knowledge content of an expert system. Knowledge can be acquired from a variety of sources, e.g. books, films, current computer applications and their data, and from human experts. Eliciting knowledge from a human expert can be invaluable since he is a user of the heuristic type of knowledge which these systems typically need to employ. There are, however, a number of problems associated with the elicitation process. An expert's time may be at a premium or he may be unwilling to pass on his expertise. And even a co-operating expert may have difficulty introspecting his own mental processes to be in a position to articulate his problem-solving strategies. Elicitation is, thus, fraught with problems not least of which is the absence of an accepted methodology for it. Knowledge engineers, who are in some senses AI's equivalent to system analysts/designers of software engineering, help the expert to make his knowledge explicit, by observing and asking about his problem-solving protocols for case studies that have arisen in the past. Following this, the knowledge engineer may represent what has been gleaned to formulate an incomplete base of knowledge in a prototype system. Whereas the software engineer has well-defined and understood data and control structures with which he represents information about data and processes, the knowledge engineer has a variety of representation schemes and control possibilities at his disposal, many of which have a short history and may not be well understood. He must nevertheless choose from them to develop a first prototype. Old and new case data are then presented to the prototype to see how it performs, with the expert providing criticism that can be used to refine the knowledge base. It has been argued that the process described above misuses the expert's capabilities because what it requires of him— articulation of the rules, definitions, and hypotheses in the domain—is not what he is accustomed to doing. What he should be used for is what he is good at, namely generating or scrutinizing examples. Given a database of case histories, a technique has been derived for inducing rules from them. These automatically

induced rules can form the basis of a knowledge base although this recent technique is not widely used either because construction of the database poses problems or because of prior commitment by the knowledge engineeer to a particular tool/product which does not have this capability.

It was stated earlier that an expert system contains much more than a knowledge base. The knowledge engineer will have to choose a method of inference which is appropriate to the domain knowledge. Building an inference mechanism is necessary if a suitable one does not exist with the tools and environment being used. The software engineer may need to be reminded at this point that knowledge is rarely represented in the same way that he represents his data and processes, and hence amenable to interpretation or execution routinely, but rather it is expressed in a symbolic form, e.g. if A and B are true then the probability of C being true is 0.3, that requires a specifically tailored mechanism to interpret it. An explanation capability is another component of the expert system which can rarely be taken from the shelf, this too needs to be designed and implemented to accord with the knowledge representation scheme to be used.

Validation of a conventional system can be achieved through a series of tests, as previously described. Results from these can be labelled right or wrong with respect to specifications and requirements definitions. But results from expert systems and indeed all AI systems are examples of behaviour based on judgemental factors. It could be argued that the only way to assess the correctness of judgments is exercise of the same faculty. Thus a particular behaviour is judged to be correct or incorrect. But who is to make the judgement? This is a question whose answer depends on the use to be made of the system: if the system is to replace human experts then it would be appropriate for those same experts to judge it. If on the other hand the system is to be used only in an advisory capacity then it is for the user to determine the extent to which he is prepared to surrender his own judgement in deference to it. Many systems may well fall between these two extremes and it then becomes a difficult matter to assess what is appropriate.

Some form of objective verification of the system, prior to operational use, should in any case be undertaken. Sell (1985) has suggested five basic requirements for validation:

—consistency,
—completeness,
—soundness,
—precision,
—usability.

A consistent expert system should produce similar results for similar problems. Results, therefore, should not vary unduly from one problem to another where the major features of the problems are the same. In particular, two problems which are alike apart from superfluous data should have the same result. A system can be complete in two senses although it can be difficult to check either. In the semantic sense a system can be regarded as complete if it covers all problems in the domain and can derive everything that is derivable from the given data—it is thus

concerned with the bounds of a domain and its meaning. Formal completeness, on the other hand, is concerned with the capability of the system to identify and successfully discriminate all factors that have a bearing on all problems in the domain. Informally, it is concerned with the syntactic structure of the knowledge base rather than its meaning. Soundness complements semantic completeness. Whereas the latter requires that everything true is derivable, soundness requires that everything derivable is true. Precision is a quality that can be ascribed to systems that make judgements to a certain level of confidence. The greater degree of accuracy in the level of confidence, the greater precision a system has. Usability is of prime concern to end-users and is not to be confused with user-friendliness. Its concern is that interactions between system and user should proceed as intended by the system developer. For example, users should not be confronted by requests for input that are ambiguous or cannot be answered (because the system has made an invalid assumption).

A set of integrated tools to aid in the validation process of expert systems is not yet available—*ad hoc* methods are the order of the day. It must be appreciated, however, that the process of verification, using the validation checks above, represents a discipline that can only inform on what must always be a subjective matter.

Tools, techniques, languages

This section provides a brief description of some of the tools and techniques that support SDLC and AI practice, and mentions some of the common languages used. It should not be assumed that the SDLC tools are in wide use: although they are increasingly being embraced, many organizations still rely on *ad hoc* methods.

Most good SDLC packages describe procedures for estimating the project costs and schedules that are needed for the feasibility stage. Some are based on historical evidence of relative sizes of the various SDLC stages, and require as input some indication of the scale of the project (e.g. from small and straightforward to large and complex); others are based on user-supplied estimates on the number of source programming statements to be developed and provide expected man-months of effort. Each organization will, of course, interpret results in the light of its past experience.

The tools and techniques needed at the requirements definition stage are mainly for documentation purposes and include aids such as PSL/PSA developed at Michigan University for creating an initial system dictionary. Many DP systems will need to integrate with current commercial practices and techniques of business systems analysis, e.g. IBM's BSP, which can be used to help define the main functions of an organization. Two well-known products/techniques which can be used at this stage to functionally decompose the system are Yourdon's structured analysis and design (Yourdon, 1982) and Softech's Structured Analysis and Design Techniques (SADT).

Structure charts, such as *HIPO*, can be used at the next stage to provide a framework for the system specification. Further elaboration of the data dictionary is

needed at this stage, for which the same tool that served in the requirements definition stage, e.g. PSL/PSA, can be used.

Jackson and Warnier–Orr diagrams are useful for providing the top-down hierarchical structure needed at the system design stage. Alternatives for equipment and proprietary software such as DBMSs may need to be evaluated at this stage, and King (1984) suggests a Kepner–Tregoe Decision Analysis Matrix to help bring objectivity to the process.

There are many products available to help in the program design and development stage. Most of them support structured design along the lines of JDM, briefly described below. There are five steps to JDM:

1 draw the data structure;
2 identify correspondences between the data structures;
3 form the program structure;
4 list and allocate the executable operations;
5 write schematic logic.

In the first step, input/output data are depicted as a series of hierarchical structures. Their processing relationships are expressed in the next step which forms the basis for the program structure. This provides two things. Firstly, processes are decomposed to smaller ones; and secondly, their executional dependence is defined. In the fourth step each of the small processes is allocated fairly primitive data handling descriptions, e.g. read customer record. Finally, this is elaborated in JDM's own version of structured English which can be preprocessed to generate *COBOL* code or used manually to provide statements in other programming languages.

Test harnesses exist that provide an environment for the testing stage. These facilitate the testing of individual programs or the system as a whole with preestablished test data. One difficulty at this stage, however, is to ensure that all possible solution paths are tried. Some test data generators are now available which, when given a detailed program-level logical design, can generate the minimum set of test data to ensure that every path and condition is tried. Doing this does not, of course, guarantee that a program is correct since not all permutations of data values are tested. As Dijkstra astutely observed, testing can demonstrate the presence of errors but never their absence. In addition, such tests are not automatically checked against the requirements definition. Users therefore need to satisfy themselves manually that results are in accord.

At the implementation and production stage most organizations have developed their own standards for installation and production procedures.

There are several hundred programming languages used by software engineers. The more important ones have been summarized by Barron (1984) and are mentioned below to give the AI worker a feel for them. *COBOL* which dates back to the 1950s is the most common and successful of the commercial languages. It has good file handling facilities and a clear separation of data from procedures. The language was introduced before the advent of the structured programming movement and it lacks structuring facilities. The enormous commitment to the language, however,

has resulted in amendments to the language over the years to improve it. *FORTRAN* and *ALGOL* 60 are the pre-eminent languages for scientific work. Each emphasizes numerical calculation and the use of arrays. Of the general purpose languages *PL/I* and *Pascal* are the most popular. *Pascal* has gained widespread approval for its ease of portability (*Pascal* compilers are implemented in *Pascal*, thus easing its portability) and good structuring facilities. *BASIC* and *APL* have been dominant as examples of interactive languages although *LOGO* and *FORTH* are comparatively recent languages that are beginning to challenge them. *CORAL* 66 and *RTL/2* are real-time languages suited to responding to external signals arriving in an unpredictable manner. These languages are not dissimilar to the ones that support concurrent programming such as *Modula* and *Concurrent Pascal*. *Ada*, however, is the language that promises to be most widely used for such work, supported, as it is, by the US Department of Defense. Finally, of the system programming languages, C is at the forefront. It combines features associated with high-level languages such as *Pascal* with efficient hardware access facilities that were previously the province of assembly languages.

Some mention must be made of fourth generation environments (4GE) that are now becoming popular and which represent the possibility of a bridge between SE and AI.

There are five generations of computer languages:

1 machine code,
2 assembler,
3 high level, e.g. *COBOL*, *Pascal*,
4 4GL, e.g. *Oracle*,
5 5GL, e.g. declarative and object-oriented languages.

The fifth is used mainly in AI, and the second and third primarily in SE. It is the fourth level which has been introduced. A 4GE, which has as a component a 4GL, recognizes the need to separate four levels of concern, namely users, screens, applications, and databases. In the past these have often been bundled together. For example, a user requirement has resulted in a single application with data and screen layouts embedded, and thus inextricably tied to it. The economics of software development and enhancement suggest that a decoupling of these will reduce programmer effort and hence cost. A 4GE may consist of a data dictionary; a screen painter that allows interactive screen design; a report generator; a dialogue specifier (really no more than a control flow between non-procedural information within a 4GE); an administrative aid for cataloguing programs; a program development guidance system; and a query facility. Many of these features have been on the SE scene for some time but they are now being fully integrated into an application development environment. It is, perhaps, the emergence of relational databases, which allow disparate data to be linked relatively simply, which has been the cornerstone for these new developments.

The idea of an integrated environment, often linked to particular hardware, is a feature of AI applications development. These environments usually have good windowing facilities and pointing devices such as the mouse. There are now two

dozen or more products which aid the process of knowledge engineering, with most emphasis placed on representing and manipulating knowledge.

An environment, for example Knowledge Craft from Carnegie Group Inc., usually offers programmers a choice of knowledge representation techniques and control strategies. It also has database management, knowledge base editors, three programming regimes (logic, rule-based, and object-oriented), and a programmers' workbench that features an icon oriented interface with windows to view internal processes and graphically displayed knowledge (usually in tree notation). AI workers are now accustomed to having such sophisticated tools, but they are expensive and for bit-mapped display are only for the single user.

Harmon and King (1985) have categorized a number of tools. The categorization serves to give the software engineer some idea of the flexible approach which AI workers can bring to software development. First, are a variety of formalisms in which knowledge can be represented, including frames (which allow contextual information to be incrementally added to declarative knowledge), IF-THEN rules which specify the conditions under which a consequent is true or should be actioned, and objects (which correspond to entities in the real world). Knowledge can be further refined with certainty factors. For example, an IF-THEN rule could have probabilities associated with one or more of its conditions being true (since the truth of them may not be determined absolutely) and the rule itself may have a certainty factor associated with it because much knowledge is judgemental. Another feature of these tools concerns the way in which new facts are generated by inference. AI uses two techniques from formal logic in this respect substantially. *Modus ponens*, which states that if A is true, and if the implication $A \rightarrow B$ is true, then B is true; and resolution, which underpins implementation of logic as a programming language in the form of *PROLOG*. A variety of inference control strategies are available to the AI worker. These include backward chaining (which considers a conclusion and then attempts to establish the truth of its premises, which in turn usually necessitates considering other conclusions), forward chaining (which draws conclusions based on the premises which are currently considered to be true), and depth-first and breadth-first search. Products can be distinguished further by the extent to which they help the knowledge engineer build the knowledge base; their explanation capabilities; display options; and their ability to take data from sources, e.g. sensors, instruments, databases, and indeed other programs implemented in various languages.

The two most popular AI programming languages, *LISP* and *PROLOG*, have been integrated into some of these environments. *LISP* (*LIS*t Processing language) dates back to the late 1950s and has been prominent in AI work since then. It is a functional language, which evaluates functions rather than performing a sequence of steps, and assigns values to variables. Data structures of *LISP* are, as the name suggest, lists. Elements of lists may be symbols or lists; the language thus relies heavily on recursion. Whereas *LISP* is based on the lambda calculus, *PROLOG* its nearest rival in terms of popularity, is based on the predicate calculus. A *PROLOG* program consists of facts and rules of inference expressed in a predicate (or relationship) formalism. (*PROLOG* is in fact a relational calculus and a program can

be regarded as a database of extensional (factual) and intensional (rule) information that can be queried in a way not dissimilar to a conventional database.) Although *PROLOG* is regarded as a declarative language, as opposed to a procedural language, e.g. *Pascal*, on some occasions it is necessary to provide control information for the desired interpretation or for reasons of efficiency.

Future developments

Although there are many software problems which could be labelled as AI or SE, and for which the methods and tools of each discipline are appropriate, there is nevertheless an increasing demand for AI systems to have elements of SE and vice versa.

'User-friendliness' has been coined to denote the sensitivity a system has to the needs of its user community. To provide this sensitivity a system may need to allow natural language input and output. If it is to sustain an acceptable dialogue it will need to have a model of the user (which indicates, for example, a user's knowledge of the system he is communicating with) and knowledge of the system—the back end. Natural language understanding, user modelling, and system knowledge modelling, are topics in AI which are receiving substantial attention in the research and development of Intelligent Front Ends to interactive AI software such as expert systems, and also to conventional software systems such as statistical packages, control systems, and DBMSs. This work heralds the emergence of conventional software 'enveloped' by a layer of AI software. There is evidence for other SE systems having an 'injection' of AI to improve performance and efficiency. Speech recognition has traditionally been the preserve of algorithmic processing of syntactic data, but semantic information about the meaning rather than the form of utterances is being explored for an advance in overall performance. AI representation and inference techniques are being applied to improve it.

Conversely, AI is in need of good data modelling and data processing capabilities if it is to address the major computational problems of the commercial and scientific world. These capabilities are, of course, already evident in conventional systems and it seems clear that AI systems need to incorporate them in order to address other than stand-alone problems with small data processing requirements.

There is, therefore, some evidence and a case for the emergence of 'hybrid' systems, i.e. systems composed of AI and SE elements. This poses two problems. Firstly, how can the software be integrated, and secondly, what are the methodological implications for the development of new systems?

It is already possible to integrate software developed in *Pascal*, with an AI language such as *PROLOG*: some implementations of *PROLOG* allow calls to be made to compiled routines of *Pascal*, *C*, assembly code, etc. But the data passing mechanisms are, as yet, crude and inhibiting. This channel of communication needs to be wider and more flexible in terms of the data types that are allowed. We expect more investigation of the uses of DBMSs in these hybrid systems and for data abstraction techniques to be used to permit greater freedom of common data definition.

Once this first problem has been overcome the methodological implications for developing hybrid systems need to be addressed for substantial systems. The move to rapid prototyping by software engineers and its consequent involvement of users at an early stage of development is not dissimilar in character to the method employed in AI. It thus becomes possible to imagine an incremental development of hybrid systems with each increment representing an elaboration of a requirements definition (in SE terms) and a closer approximation to a solution in AI terms. The details of such a liaison of the two disciplines will obviously need to be carefully considered by both communities, but it does appear to be a pre-requisite of orderly development of hybrid systems.

Extreme problems of AI and SE for which no hybridization is necessary can also benefit from a methodological point of view with the application of techniques from the other discipline. Research projects are examining the possibility of using knowledge-based systems to help in the control and elaboration of various stages of the SDLC. In addition the verification techniques of formal SE are being examined for their usefulness in the validation process of expert systems.

These activities and needs of the user community presage a closer relationship between the two disciplines. In order for this to take place AI workers need a better understanding of SE, and software engineers a better understanding of AI. It is hoped that this tutorial has helped in some way to provide that understanding and will encourage practitioners of both disciplines to explore further for themselves.

Acknowledgments

A number of issues addressed in this tutorial were explored during a course on the relationship of AI and SE. I thank students on the course for their contribution to my understanding.

In addition I wish to thank Derek Partridge for many useful comments on an earlier draft.

References

Barron, D.W. (1984) Programming Languages: coherent design for simplicity, In: *The Computer Users Yearbook, Volume 1* (ed. R. Labbett) VNU Business Publications BV, London.

Bishop, P. (1986) *Fifth Generation Computers—Concepts, Implementations and Uses.* Ellis-Horwood Ltd., Chichester.

Buchanan, B.G. & Feigenbaum, E.A. (1978) *DENDRAL* and *meta–DENDRAL*: their applications dimension, *Artificial Intelligence*, **11**, 5–24.

Campbell, J.A. (1984) Three uncertainties of AI. In: *Artificial Intelligence: Human Effects* (eds M. Yazdani & A. Narayanan) Ellis-Horwood Ltd, Chichester.

Doyle, J. (1985) Expert systems and the myth of symbolic reasoning. In: *IEEE Transactions on Software Engineering, Vol SE–11*, **11**, 1386–1390.

King, D. (1984) *Current Practices in Software Development.* Yourdon Press, New York.

Partridge, D. (1986) Engineering artificial intelligence software, *Artificial Intelligence Review*, **1**, 27–41.

Partridge, D. & Wilks, Y. (1987) Does AI have a methodology which is different from software engineering?. *Artificial Intelligence Review*, **1**, 111–120.

Sell, P.S. (1985) *Expert Systems—A Practical Introduction*. Macmillan, Basingstoke.
Shortliffe, E.H. (1976) *Computer-Based Medical Consultations: MYCIN*. Elsevier, Amsterdam.
Waterman, D.A. (1986) A Guide to Expert Systems. Addison–Wesley, New York.
Yourdon, E. (1982) *Managing the System Life Cycle: a Software Development Methodology Overview*. Yourdon Press, New York.

Further Reading

This tutorial does little more than sketch some similarities, differences, and points of contact between AI and SE. The reader is recommended to consult some of the texts below for a more comprehensive understanding of the relationship between the two disciplines.

Sommerville, I. (1985) *Software Engineering*. Addison–Wesley, New York. (A fairly formal but readable introduction to Software Engineering.)
King, D. (1984) *Current Practices in Software Development*. Yourdon Press, New York. (A good guide to the tools and techniques of SDLC used.)
Charniak, E. & McDermott, D. (1985) *An Introduction to Artificial Intelligence*. Addison-Wesley, New York. (A formal and comprehensive introduction to AI.)
Harmon, P. & King, D. (1984) *AI in Business: Expert Systems*. John Wiley & Co, Chichester. (A useful guide to the tools and techniques used in the development of practical AI software.)
Partridge, D. (1986) *Artificial Intelligence: Applications in the Future of Software Engineering*, Ellis Horwood Ltd, Chichester. (A thought-provoking treatment of the problems and potential for generating practical AI software.)
Rich, C. & Waters, R.C. (eds) (1986) *Readings in Artificial Intelligence and Software Engineering*. Morgan Kaufmann, Los Altos, California. (A collection of papers which represent the somewhat limited view that 'the ultimate goal of AI applied to SE is automatic programming'. Nevertheless, some interesting ideas on the possibility of AI tools and techniques easing the task of the Software Engineer.)