Life cycles in Software and Knowledge Engineering : a comparative review.

Michael Wilson, David Duce

Informatics Dept., Rutherford Appleton Laboratory

Dan Simpson

Dept. of Computer Science, Brighton Polytechnic.

ABSTRACT

Developments in software engineering have led to models of the system life cycle incorporating the use of prototyping and formal methods of program verification. These are becoming supported by integrated project support environments and permit the planning and monitoring of software development projects.

In contrast, Knowledge Based Systems (KBS) are developed using informal views of the system life cycle. Tools have been developed to support some stages of the life cycle in an undisciplined manner. The commercial use of KBS needs development projects to be planned and monitored. This requires methods and tools based on systematic life cycle models to be established for KBS.

This paper reviews the current state of life cycle approaches to software engineering and KBS development projects in order to provide a direction for the development of methodical KBS life cycle models.

Introduction

Over the past 20 years it has become accepted in the software engineering community that software development should be undertaken using a model of the software life cycle. The benefits of such an approach include:

the ability to plan the project;

the ability to estimate resource requirements for the development;

the ability to size the likely software product;

the ability to estimate hardware requirements;

the ability to update estimates on the basis of real figures during monitoring;

the availability of documents for monitoring and control;

the ability to fit the development process into a Quality Management System;

a development structure which may be audited for quality.

The result of using life cycle approaches is that the development process is made visible to the project management, project controller, quality controller, the project sponsor and users of the system. The major benefit is that potential problems can be caught early within the development with substantial savings in effort and resources used in rework and correction.

The traditional life cycle model used in software engineering has provided some of the above advantages, although it has been shown not to apply to all styles of product development; particularly where the program requirements are initially ill specified.

This paper outlines the state of life cycle models in software engineering and Knowledge Based Systems (KBS) as a foundation for the development of KBS life cycle models and associated development techniques which will allow the KBS builder to develop software in a way which will accrue the benefits listed above.

Current Commercial Practice and KBS life cycles.

Expert systems have now penetrated the commercial market not only in that tools are being bought to enable industry to explore their potential but also to develop practical systems which are now being used. Some individual organisations have now accumulated considerable experience in the development of KBS, however, there is no consensus in industrial practice on a development method for them.

Most KBS currently developed are stand-alone systems that ask users questions and reason through a hierarchical classification in order to explain solutions. The problems addressed involve the diagnosis of symptoms or fault finding. A majority of these systems are small and are based as much on the regulations and theory that describe how decisions should be reached as on human expertise. The development of these systems in industrial practice is led by a colloquial life cycle model based on early experience with KBS development which has been outlined, but not formalised (e.g. Hayes-Roth, Waterman and Lenet, 1983; Frieling et al, 1985). This generally follows the stages described below.

1) Problem description

This usually divides into three sub parts:

A) The identification of problems where the use of KBS within an organisation will provide the leverage to overcome them cost effectively.

B) Initial problem analysis

A description of the problem will be assessed as to whether it really is amenable to a KBS solution (e.g. after Prerau, 1985).

C) Description of problem requirements

The intended role the system and the requirements it should meet are identified and a user model is established to guide future development.

For larger problems there may be an embedded version of the whole life cycle here applied to some sub-set of the problem as a feasibility study. In practice, prototypes are

often produced at this early stage to present clients with concrete evidence of the capabilities of KBS. Consequently, a sequence of successive prototypes is often developed.

2) Knowledge acquisition

Knowledge acquisition is not a clear stage since it is involved in the problem requirement description and continues through later stages. A commonly used definition of knowledge acquisition is (cited by Anjewierden, 1987:29):

> "Knowledge acquisition consists of the elicitation and interpretation of data on the functioning of expertise in some domain, in order to design, build, extend, adapt or modify a knowledge based (expert) system (KBS). In this view, knowledge acquisition is a permanent and crucial activity throughout all stages of designing, implementing and maintaining an expert system."

The knowledge acquisition usually takes the form of interviews between a knowledge engineer and one or more domain experts (see McGraw and Seale, 1988 for a review of the issues motivating the choice of how many experts to use), followed by the analysis of interview transcripts. For problems where there is reason to believe that the knowledge used to perform a task cannot be verbalised easily, several other methods may be used. Firstly, tools can be used to induce knowledge from sets of examples. However, these tools have well documented limitations to their application (see Hart, 1987 for a review). For example, a *complete* set of examples of the problems and solutions is required, although it is known that completeness is very difficult to assess without a thorough knowledge of the domain which must be absent to motivate the use of induction techniques. Secondly, performance at the task may be observed and recorded, followed by the analysis of performance protocols (see Bainbridge, 1986, for a review). Thirdly, psychological distance measures may be used to structure domain concepts although these are not applied as often as they are appropriate (see Gammack, 1987 for a review). Fourthly, questionnaires and other focussed indirect measures of expert knowledge may be used, especially to test the completeness of the developing knowledge base.

The knowledge acquired must be fed back to the domain expert so that it can be validated. For this to happen, it must be represented in some form. The type of representation chosen will interact with the structure of the life cycle. Firstly, it may be presented to the expert in the form of a prototype system. This will allow the expert to check the inference processes working with the knowledge quite easily, and both to correct errors and to identify and correct omissions. However, a knowledge engineer will be limited to using representations available in the prototyping tool, none of which may be compatible with the natural representation of the domain knowledge. Such tools usually employ general knowledge representation techniques such as frames, production rules, logic or semantic nets (see Cercone and McCalla, 1987 for a review) which the domain expert is required to understand if the prototype is to be modified successfully. This may involve considerable learning effort.

Alternatively, the knowledge acquired may be presented in the form of a paper knowledge base intermediate between the expert's representation and that in the implementation vehicle. This offers the possible advantage of using a representation familiar to the domain expert and appropriate to the domain knowledge structure, thus saving the expert effort in learning a new representation, for example, a standard software, design or fault representation may be selected (see Peters, 1980, for a review). If no domain specific representation is available, a specifically designed mediating representation (see Young, 1987 for a review) could be used which is easily learned. The use of a paper knowledge base also prevents the knowledge engineer from making inappropriate implementational trade-offs at this stage of the development. Although paper knowledge bases aid in the validation of the static knowledge, only very simple inference strategies can be followed on them, whereas prototypes allow these to be followed with ease.

If a prototype is used, there will be an iteration between this stage and the next with the result that the representation chosen will probably follow through to implementation. If a paper knowledge base is used, several different representations may be used through the

life cycle where each has advantages.

There is no formalised method used to select techniques appropriate to particular problem and knowledge types. The usual motivation for the choice of technique is familiarity to a member of the project team, or the constraints of the software tools available. Unless the representation chosen is solely used for mediating, it will result from a tradeoff between expressive power and computational tractability. The representation chosen should have the power to express the problem as understood, however, as expressive power increases, computational tractability reduces (see Williams and Lambert, 1988 for a review of this tradeoff).

3) Prototype design and development

There are two reasons proposed for the development of prototypes in the KBS development cycle:

i) the user requirements cannot be specified adequately at the outset;

ii) the technical aspects of system design or implementation cannot be specified adequately.

These arguments apply more to KBS development than to conventional software development since KBS requirements specifications are often non-existent, imprecise, or rapidly changing. For a summary of the issues concerning whether prototypes should be discarded after being transformed into paper specifications, incrementally developed into the deliverable implementation or iteratively developed into the deliverable implementation see (Springer-Verlag, 1984).

Whichever style of mediating representation was used for knowledge elicitation, a final decision will have to be made at this stage about both the representation and the inference strategy to be used in this and later stages of development.

4) Knowledge base implementation and refinement

When the knowledge acquired and the specification have reached an acceptable level of validity and completeness, the full system is developed. The decision must be made here as to what the delivery vehicle for the system will be. This may be a progressive development of the prototype or a completely new design and implementation. Often the delivery vehicle for this implementation will be different from that used for the prototype. If prototypes are developed on workstations using AI toolkits, they will frequently be redesigned and re-coded in conventional languages to be used on cheaper and more generally available delivery vehicles.

There is usually iteration between this stage and the next as there was between stages 2 and 3.

5) Testing

A partially implemented system will often be tested in the target environment or an environment simulating it in order to assess how it meets the requirements. A problem often found at this stage is to establish a database of cases against which to test the developing system. This may cause the developers to run the KBS in tandem with a present system involving human expertise in order to compare the results and verify the KBS.

The testing procedure is usually performed informally, often by the expert who acted as a knowledge source using the system and either noting down disagreements with the system or making changes to it directly. In those cases where more thorough testing is performed (e.g. Wyatt, 1987) changes are often required to the delivery vehicle to meet the user's requirements.

6) Installation and System Integration

Many large KBS will use pre-existing software to provide them with data. They may be linked to large mainframe systems from which much of the information they require can be retrieved. This can require considerable integration into existing systems particularly if the whole resulting system runs under time constraints. All KBS will have to be

integrated into a pre-existing organisational environment. This will include not only educating users in the use of KBS and the modification of existing organisational systems to incorporate them, but also the establishment of procedures to maintain the KBS.

7) Use and Maintenance

A KBS can be maintained either by the customer/user or by the developer. Since the knowledge can change frequently and quickly, the KBS must change quickly too. For example, a KBS in the financial domain may have to change within hours of a national budget. Many conventional databases are put under the same requirement, though the problem here is comparatively trivial. For the KBS as for the conventional databases the solution is not for the customer to have a lifetime maintenance contract which would make them dependent on a developing company that may stop trading. The customers must be able to maintain the KBS themselves. Therefore, they must understand the knowledge base structure and the inference procedures applied. The problem has been identified with many AI toolkits that users find it difficult to develop a consistent model of them. This is likely to be particularly so for domain experts involved only in maintaining systems.

This brief description of a life cycle does not describe how all systems have been developed, or specifically how many research tools could be used, but it is consistent with commercial practice. Commercial expert system developers are aware of the need for documentation in order to monitor the progress of projects, but since there is no accepted set of documenting points, none have been included here.

Tools are used where available, but tool support has been provided on a stand alone basis motivated by local difficulties with the production of KBS. Existing tools (e.g. shells and AI toolkits) only help the Knowledge Engineer in actually coding a prototype or implementation of a KBS, there are very few tools to support other parts of the cycle. Further, these tools only aid the actual coding task itself and do not offer control structures for

code version management or documentation control of the type used in conventional software development projects. Thus it is difficult to select appropriate tools for different phases in the development process because of compatibility problems between the output provided by one tool and the input required by the next. Equally, since the tools have not been designed to be used as alternatives there are few grounds on which to choose those appropriate to a particular project.

The life cycle illustrates where research effort has been devoted in the past (e.g. development tools, knowledge acquisition techniques). However, there still remain many sections of the life cycle which are presently addressed in unsystematic ways based on the undocumented experience of the individuals involved. For example, those with experience in developing expert systems do make planning decisions about cost, effort and time, but no large scale studies have been performed to provide general formulae for these as they have in conventional programming. To overcome this lack of data for estimating, one managerial technique which has been used is that of staged contracting, whereby a customer only agrees to the next stage in the development rather than committing himself to the full development cycle at any point.

A modified form of this informal stereotypical life cycle has been used for systems which are larger than the usual, based on pre-existing knowledge bases, incorporated into existing software, or used to develop embedded or real time systems, but there are only examples of the use of such modified life cycles and no formal description of them.

There are three major sources of information which can be drawn on in order to change present KBS life cycle management. These are, firstly, the careful appraisal of both successful and unsuccessful large scale KBS projects. Although there are many accounts of successful small KBS and some of large KBS, few accounts of unsuccessful KBS projects are available. Secondly, there are methods used in conventional software engineering which can be modified for the KBS development cycle. Thirdly, the output of research projects on the KBS life cycle or parts of it, which, if they scale up to large

systems, can be incorporated into a life cycle oriented view.

Software Engineering Research and life cycles.

The classical waterfall model of the software development life cycle is shown in Figure 1. It was recognised in the waterfall model that as a system progresses through the life cycle it becomes more complete and complex. Consequently, changes at later stages are more costly than those at early stages with changes in the maintenance phase being extremely expensive compared to those in the requirements specification stage. Each stage in the development process is meant to produce a deliverable which can be verified against previous stages and validated against the initial requirements. This structuring has permitted the development of methods and tools to support them which aid both management and developers in estimating time, cost and manpower requirements through a project's life.

Figure 1 about here

Fig 1. The classical waterfall software life cycle (after Macro and Buxton, 1987).

In the waterfall model the development process is seen as proceeding sequentially through the identified phases, with each being completed before the next is started, thereby providing no possibility for the modification of work done at an earlier stage. This model has been recently supplanted in software engineering by life cycle models (for example that illustrated in Figure 2) in which the process is seen as a dynamic one in which exploratory or prototyping work into later stages in the development process is done to provide essential feedback to the current stage.

Figure 2 about here

Fig 2. A recent software life cycle (after STARTS 1987).

The functional role of the specification in these life cycles should be noted. It is to provide both the customer/client and the developer with a contractually binding view of the product that should be produced. Both the customer and the developer will "sign off" the specification agreeing respectively that it is a description of what is required and what will be delivered. Within this view, the two questions which arise are, whether the customer can specify what is required, and whether a specification presented by the developer can be understood sufficiently that it can be seen to meet those requirements. The specification can be presented to the customer as a prototype which can be tested and the functionality of which can be agreed upon. This prototype can then be recoded either automatically or manually into a paper specification which can be contractually binding. Alternatively, the specification can be presented only on paper where the functionality (especially of the interface) may be less clear. In either case, when a paper specification is produced it can be tested to show that it fulfills the more formal demands on the system which may not be visible in a prototype; for example, that the thirteenth decimal place in an accountancy program is handled as desired. Where either the prototype itself, or the specification can be shown not to meet desires before it becomes contractually binding, it can be altered. Where the agreed specification does not describe the customer's desires because the mismatch was not noticed, the development of an unsatisfactory product will become contractually binding. After the specification is agreed the developer should be able to estimate, plan and monitor the development project with a clear view of both the end product and the process that will deliver it.

It is useful to identify and define the components of development approaches. Current development approaches should have five components: notations, development rules, development guidelines, development processes and development methods. A notation comprises a syntax, a semantics and the relations between them which will allow a system to be described. A development rule determines which descriptions can be verified

with respect to other descriptions. Development guidelines indicate how to use notations and development rules to best effect. A development process lays down the nature of, relations between, and order of development for, the descriptions to be constructed and verified. It thereby identifies, among other things, what test strategies are used and how quality assurance is performed. A development method lays down the notations, development rules and development guidelines to be adopted in a particular development process.

The IEEE have published a set of standards for various phases in the software development life cycle (ref ??) which are being submitted to the ISO by ANSI. These are expected to be brought together under international standard ISO 9000 by 1989 in order to ensure that the quality and productivity of a project can be planned and monitored.

The software engineering community already has a set of methods and tools which can be used to instantiate these standards. Development methods such as JSD (Jackson, 1982), SSADM (Ashworth, 1988), Yourdon (Yourdon, 1972) and MASCOT (1979) are gaining reasonably wide acceptance in certain application domains. The acceptance stems largely from the existence of experience from which guidelines have been developed. The notations tend to be graphical and the introduction of tooling for these methods is an active development area. Few development rules are available for these methods.

In addition to these accepted methods, both more rigorous and formal methods have been developed at the research level. A rigorous development method is one in which the guidelines override the rules only when they provide sufficient confidence that the rules could be used to verify the properties assumed for a description. A formal development method is a development method in which the guidelines do not override the rules, in other words, the rules must always be applied to verify that a description has the properties assumed of it.

Many formal notations for system specification have emerged in recent years, for example VDM, Z, OBJ, CSP, Larch, Special, CIP (see Cohen, Harwood and Jackson, 1986, for a review). Some of these notations have associated formal rules (i.e. VDM) and are starting to find application in industry (at least for critical points in systems) as well as working towards international standardisation. Prototyping tools to support the notations are becoming available, and case study work is beginning to build up in some application domains from which development guidelines can be extracted.

In the context of formal methods, work on development rules and guidelines is directed at the proof obligations which are imposed by formal methods. Much research remains to be done at the level of both theory and tools before industrial users will be able to apply such rules widely.

There are various techniques for verifying that new descriptions have the desired properties of earlier descriptions. Especially important techniques are:

Value testing, which can demonstrate that a new description satisfies test cases in which the input of particular values leads to the output of particular values;

Conventional testing, which differs from value testing in that the inputs and outputs may be symbolic expressions instead of particular values;

Proof which within some theoretical framework can demonstrate with certainty that the new description is a correct refinement or implementation of the old description;

Transformation which produces the new description by modifying descriptions only in ways that are guaranteed to preserve the desired properties.

The current state in the verification of systems in software engineering is that it is only feasible to formally verify small modules of code (200 lines of Pascal or less). Verification therefore tends to be applied only to parts of a design which are critical in some sense. The remainder of the code is verified using a more informal technique.

There is much current work on the provision of tooling for the development process. Classically, tool support has been provided on a stand alone basis resulting in incompatibilities similar to those found in KBS development. Under ESPRIT funding, a common tool interface (PCTE; Cambell, 1986) has been defined which includes user interface and object storage components.

Tooling for formal notations and methods is slowly emerging from a number of projects. A VDM tool-set providing structure editors and syntax checkers is available (Crispin, 1987) and generic proof tools are the subject of a part of the Alvey IPSE 2.5 project (Jones, 1987). Other notations and methods are supported by various degrees of tooling.

In addition to the constructive aspects of the development process, life cycle approaches also address the managerial issues: how to manage the process, to ensure that all the stages which should have been followed have been followed, to ascertain the state of the development at any point in time, to provide assistance for planning development projects and in estimating costs, resource effort, and time-scales.

There are several current methods available for estimating the cost of conventional software development, the time to complete projects, and the manpower and effort required by them. Most of these use measures derived from large numbers of case studies of system life cycles. If these methods are sufficiently accurate, they can be used to guide decisions about which path to take within the life cycle. When the estimators have a large body of experience of programming in that domain, or make estimates at the later stages during the development cycle, current software cost estimation models can estimate conventional software development costs within 20% of actual costs 70% of the time (see Boehm, 1981 for a review).

Methods and tooling for these managerial issues are addressed to varying degrees of sophistication by Integrated Project Support Environments (IPSE's). The key point about an IPSE is that it provides support for all aspects of a development project in an integrated manner, not just for the limited task of "writing the code". The more advanced

research projects in this area (for example IPSE 2.5) are concerned with how to provide such support in a generic way; the idea is that an IPSE provides a language and framework within which particular development methods can be described and tooling provided. The development process is then controlled by effectively executing this process description, so that, for example, a user of the system will be told what activities can currently be performed.

For an IPSE to be driven for some project requires decisions to have been made as to which life cycle model to use. Depending on the power and generality of the IPSE in use, this decision can be more or less tentative.

This section has concentrated on describing work in software engineering which may act as a basis for improvements in the development of KBS. But, it should be noted that the software engineering work is not complete. For example, there is still much research taking place on requirements capture, prototyping and the validation of the design against the customer's desires at points through the life cycle. Advances in these areas may lead to future developments in other parts of software engineering based life cycles.

KBS Research Advances and Life Cycles.

Following software engineering development methodologies such as Yourdon, Keller (1987) has presented a structured analysis methodology for KBS development. The life cycle supporting this methodology (illustrated in Figure 3) follows that of conventional structured systems analysis closely. A survey or feasibility study is used to decide if a project is worth doing. The output of this should not only recommend specific areas that can benefit from KBS, but also how KBS should be integrated into the customer's business, and the evolution of training policies. This is followed by "Structured Analysis" in which the user's needs are specified in terms of functions to be performed and the data relationships between them to produce a "Structured Specification". A "Logical Knowledge Base Description" is also produced to describe the logical information needs of the project. This description is used in the "Physical KB Design" to specify the

implementation details of the knowledge base. The "Structured Specification" is used in the "Design" phase to specify how the user's needs are to be implemented, and the "Implementation" phase consists of the production of the system specified in the "Packaged Design". In parallel with these phases "Knowledge Acquisition" takes place providing information which is available to all the other phases.

Figure 3 about here

Fig 3. Structured Systems KBS Life cycle (after Keller, 1987).

A second KBS life cycle developed from a conventional model is presented in the KADS methodology for KBS development resulting from ESPRIT project 1098 (Hayward, 1987). This describes a life cycle based on the waterfall model, the sequential development steps for a system, and the documents that are required to monitor the completion of these steps.

Figure 4 about here

Fig 4. Esprit P1098 KBS Life cycle (after Hayward, 1987).

This life cycle has been developed in detail for the initial analysis phase and will be developed for the design phase. It is unlikely that it will extend to the other stages. In the analysis phase a thorough description has been developed for types of task which provides inference strategies and meta-classes of knowledge compatible with them categorised by problem type. These are provided as a knowledge acquisition method and a specification of the documentation required for a requirements analysis and specification. The analysis stage is supported by a computer based documentation system, a detailed handbook and tools to aid developers in its use (Anjewierden, 1987). One limitation within the analysis phase is that the knowledge acquisition method only applies to the

capture and structuring of information obtained verbally and has not been developed to account for other knowledge sources.

The overall life cycle has many of the same limitations as the life cycle in current commercial practice in that:

a) it is described only from the point of view of the software engineer, and so, does not include estimating methods;

b) it only accounts for the "normal" KBS system, it does not describe the life cycles for KBS which interact with conventional software, either to access databases, as real time systems, or as other forms of embedded system;

c) it does not incorporate software engineering methods for verification and validation across the whole, or stages of, the life cycle.

d) it does not incorporate prototyping, and follows a strictly incremental path.

The methodology has not yet become (either formally or de facto) an industrial standard, but it has been very influential in European research centres. Systems have been developed using the methodology for the parts of the life cycle where it applies and have shown it to be useful in those areas.

As well as the development of these overall life cycle approaches, there have been attempts to provide verification rules for KBS development following those used in software engineering (e.g. Green and Keyes, 1987; Martinez et al, 1987; Nguyen, 1987). However, most of these have failed because expert systems are typically developed from informal specifications through prototyping. This method has not provided adequate specifications for requirements tracing since the desired outputs are often emergent properties of the interaction between the knowledge base and the inference engine. Consequently, several engineering analyses have been suggested as possible evaluative measures on KBS. These include: criticality (the cost of failure of the system); sensitivity (the systems response to variations in the input); efficiency (can the system respond within time and resource limits); Maintainability; interaction analysis; truth analysis;

uncertainty analysis.

In contrast to these advances based on software engineering, Partridge and Wilks (1987) have argued that because the areas of application of AI programs do not permit specifications to be exact, no matter how formal the verification of descriptions produced during development against an initial specification, they will always be inaccurate since the specification itself was. Consequently, they argue that the natural form of AI methodology is a "Run-Understand-Debug-Edit" (RUDE) cycle where the requirements can be continually altered. In this view specifications can be seen as the products of AI developments.

Although this cycle may appear to be very different from those suggested above it is not, since it still requires the components of a life cycle. When the start of the cycle is altered to "Debug-Edit-Run-Understand" it is clearly the conventional life cycle of: "Debug" a system by investigating its problems and producing a set of requirements and the specification of a new system; "Edit" that specification until it is a finished product by designing and implementing it; "Run" the product by supplying it to users in the system; "Understand" it by observing its performance, and then maintain it by starting the cycle again. RUDE feels as though it is a faster loop around the cycle than a software engineering life cycle because it does not describe all the stages in detail. Software engineering has described the stages and methods for accomplishing them and is now developing IPSE’s to support the automatic control of the cycle in order to increase the speed and ease at which it can be performed. These will offer the ability to change program requirements more frequently than when greater manual effort is required during the cycle. They should perform the functions which the AI toolkits designed to support RUDE do while encouraging planning and monitoring. In this paper the developments in software engineering and KBS life cycles have been described so that further developments may be able to offer the planning and monitoring facilities to KBS development which are becoming available in software engineering.

References

Anjewierden, A. (1987) Knowledge Acquisition Tools. *AICOM 0(1).* 29-38.

Ashworth, C.M. (1988) Structured systems analysis and design method (SSADM). *Information and Software Technology*, **30** (3), 153-163.

Bainbridge, L. (1986) Asking Questions and Accessing Knowledge *Future Computing Systems*, **1** (2), 143-149.

Boehm, B.W. (1981) *Software Engineering Economics.* Prentice Hall: Englewood Cliffs, NJ, USA.

Cambell, I. (1986) PCTE proposal for a common tool interface. In I. Sommerville (ed.) *Software Engineering Environments* Peter Peregrinus on behalf of the Institution of Electrical Engineers: London.

Cercone, N. and McCalla, G. (1987) What is Knowledge Representation. In N. Cercone and G. McCalla (eds.) The Knowledge Frontier: Essays in the Representation of Knowledge. Springer-Verlag: New York.

Cohen, B., Harwood, W.T. and Jackson, M.I. (1986) *The specification of Complex Systems.* Addison-Wesley: Wokingham, UK.

Crispin, R.J. (1987) Experience Using VDM in STC. In *VDM '87: A Formal Method at Work, VDM-Europe Symposium* Springer Verlag: Berlin.

Ford, L. (1987) Artificial Intelligence and Software Engineering: A Tutorial Introduction to There Relationship. *Art. Int. Review*, **1** (4), 255-273

Frieling, M., Alexander, J., Messick, S., Rehfuss, S. and Shulman, S. (1985) Starting a Knowledge Engineering Project: A Step-by-step Approach. *AI Magazine* **6** (3), 150-165.

Gammack, J. (1987) Different techniques and different aspects on declarative knowledge. In A.L. Kidd (ed.) *Knowledge Acquisition for Expert Systems: a Practical Handbook.* Plenum Press: New York.

Green, C.J.R. and Keyes, M.M. (1987) Verification and Validation of Expert Systems. In *WESTEX 87 - Proceedings of the Western Conference on Expert Systems,* 38-43. IEEE Comput. Soc. Press: Washington, D.C.

Hart, A. (1987) Role of Induction in Knowledge Elicitation. In A.L. Kidd (ed.) *Knowledge Acquisition for Expert Systems: a Practical Handbook.* Plenum Press: New York.

Hayes-Roth, F., Waterman, D.A., and Lenet, D.B. (1983) *Building Expert Systems,* Addison-Wesley: Reading, Mass.

Hayward (1987) How to build knowledge based systems: techniques, tools, and case studies. In *ESPRIT '87: proceedings of the Esprit Conference,* 665-687. CEC: Brussels.

Jackson, M.A. (1982) *System Development* Prentice-Hall International: Englewood Cliffs, NJ.

Jones, K. D. (1987) Support Environments for VDM. In *VDM '87: A Formal Method at Work, VDM-Europe Symposium* Springer Verlag: Berlin.

Keller, R. (1987) *Expert System Technology: Development and Application.* Yourdon Press: Englewood Cliffs, NJ, USA.

Macro, A. and Buxton, J. (1987) *The Craft of Software Engineering* Addison-Wesley: Reading, Mass.

Martinez, J., Muro, P., Silva, M. (1987) Modelling, validation and software implementation of production systems using high level Petri Nets. In *Proceedings of the 1987 IEEE International Conference on Robotics and Automation,* 1180-1185. IEEE Comput. Soc. Press: Washington, D.C.

Mascot (1979) *The Official Handbook of MASCOT.* MASCOT Suppliers Association

McGraw, K.L. and Seale, M.R. (1988) Knowledge Elicitation with Multiple Experts: Considerations and Techniques. *Art. Int. Review,* **2** (1), 31-44.

Nguyen, T.A. (1987) Verifying consistency of production systems. In *Proceedings of the Third Conference on Artificial Intelligence Applications*, 4-8. IEEE Comput. Soc. Press: Washington, D.C.

Partridge, D. and Wilks, Y. (1987) Does AI have a Methodology which is Different from Software Engineering? *Art. Int. Review* **1** (2), 111-121

Peters, L.J., (1980) Software Representation and Composition Techniques. *Proceedings of the IEEE,* **68** (9), 1085-93.

Prerau, D.S. (1985) Selecting an Appropriate Domain for an Expert System. *AI Magazine,* **6** (2), Summer, 26-30.

Springer-Verlag (Pub.) (1984) *Approaches to Prototyping*, Springer-Verlag: Berlin.

STARTS Purchasers Group (1987) *Supplement to the STARTS Purchasers Handbook* NCC Publications: Manchester.

Williams, A. and Lambert, S. (1988) Expressive Power and Computability. In G.A. Ringland and D.A. Duce (Eds.) *Approaches to Knowledge Representation: An Introduction.* Research Studies Press: Letchworth, England.

Wyatt, J. (1987) The evaluation of clinical decision support systems: a discussion of the methodology in the ACORN project. In J. Fox, M. Fieschi and R. Engelbrecht (eds.) *Proceedings of AIME '87.* Springer-Verlag: Berlin.

Young, R. (1987) Intermediate Representations. In M.A. Bramer (Ed.) *Research and Development in Expert Systems IV,* Cambridge University Press: Cambridge.

Yourdon, E. (1972) *Techniques of program structure and design.* Yourdon Press: Englewood Cliffs, NJ.