The Knowledge Engineering Review

Convergence of AI programming and software engineering

C. J. RAWLINGS
Imperial Cancer Research Fund, P.O. Box 123, Lincoln's Inn Fields, London WC2A 3PX

A Review of *ARTIFICIAL INTELLIGENCE PROGRAMMING ENVIRONMENTS* (Editor: Robert Hawley) and *SOFTWARE ENGINEERING ENVIRONMENTS* (Editor: Pearl Brereton)

Artificial Intelligence Programming Environments
Editor: Robert Hawley, Ellis Horwood Ltd, Chichester 1987. Pp. 214, £39.50.

It is often difficult to explain all the advantages of working in an extensible and interactive programming environment to people who have never used an AI programming language. *Artificial Intelligence Programming Environments* edited by Robert Hawley is therefore intended to reveal what makes AI programming environments important to AI practitioners. This is a valuable goal since the work style and programming practices encouraged by AI languages seems to be the cause of some of the more significant cultural differences between "traditional" and AI programmers. Hawley's book also provides an overview of different AI languages and their programming environments.

Distinguishing the point where the AI language ends and the programming environment begins is often very difficult. In fact, it is this very point—that AI programming environments have been built by extending the AI programming language itself—that makes them so powerful. Robert Hawley is to be commended for his attempts to draw common threads between the very different styles that are found in the AI programming community. He has combined a small number of contributions that have appeared elsewhere (e.g. the *AI Magazine* of the American Association of Artificial Intelligence) with commissioned chapters in order to give roughly equal coverage to logic (Prolog) programming, applicative (Lisp and POP11) languages and object oriented programming.

Artificial Intelligence Programming Environments contains 11 chapters in 214 pages divided into 3 parts and a 2-page index. The first part contains three chapters on programming in Lisp, POPLOG and the object-oriented language Smalltalk. Part two covers the development of what are called *AI tools* although more specifically these two chapters deal with issues concerning the design of very high level (essentially rule-based) languages for knowledge representation. Part three (AI development environments) is the largest section of the book and its six chapters describe a variety of issues more or less related to AI development environments. Only two chapters attempt to describe what many people might interpret to be the meaning of this section heading i.e. AI development toolkits such as IntelliCorp's KEE (chapter 10) and LOOPS (chapter 11).

1 AI programming languages

The overwhelming feeling I got from this book, particularly in the earlier chapters, was the enthusiasm that the authors had for their chosen language and for what they thought made it a productive and exciting environment in which to work. Unfortunately, this enthusiasm was occasionally tinged with prejudice and I was sad to see repeated some of the more ill-informed and largely irrelevant shots from the old war between the proceduralists and declarativists that a few years ago was rekindled in battles between Lisp and Prolog followers. It is therefore particularly

unfortunate that we find the worst examples of parochialism in the first chapter (Lisp Programming by Tony Hasemer). In a rather jaunty and only mildly useful introduction to Lisp it is suggested (pp. 27) that "No one, at least no-one in his or her right mind, who has ever used such a system as an interactive program-development tool, ever wants thereafter to use any other" and (pp. 29) that Lisp is "as some ... would argue, the only language usable by the sane". Prolog is begrudgingly suggested as useful in applications that require pattern matching.

A common theme in the three introductory chapters is the importance of interactive program debugging tools and screen editors coupled closely to the language interpreter. This is one of the few themes that carries across the divisions amongst the different implementation languages. Chapter 1 provides a brief overview of the stepping debugger in Lisp and a short description of the Prolog trace package appears in chapter 6 where Pain and Bundy discuss this and other methods of displaying the Prolog proof tree for teaching novices how to program. Du Boulay in chapter 2 also mentions the use of the Prolog tracer for teaching but he concentrates on the importance of the extendable screen editor VED in POPLOG for all aspects of program development and interactions with the POPLOG environment.

2 Object-oriented programming

Chapter 3 (Smalltalk: The only true object-oriented programming environment by Steven Holmes) is the first of three chapters spread throughout the book on object-oriented programming (OOP). Holmes provides a readable introduction to the history and general features of Smalltalk, including a description of how the environment can be modified by the programmer. The other OOP chapters are chapter 9 (Object-oriented programming in FLAVORS and CommonORBIT by Koenraad De Smedt) and chapter 11 (Loops: a multi-paradigm environment by Martin Gittins). In chapter 9, De Smedt provides a useful review of the difference and similarities between FLAVORS and CommonORBIT with occasional references to Smalltalk and CommonLOOPS. He provides detailed accounts of both the mapping between message passing and function application and the inheritance methods in the different OOP systems. In chapter 11, Gittins reviews the knowledge representation methods available in LOOPS.

Taken together, these three chapters provide a very good introduction and useful overview of OOP; although occasionally the terminological differences between different OOP languages becomes bewildering. My main criticism is that they do not address the issues of how OOP can be used for AI research or development. The examples given show applications such as window-management graphics or, as in chapter 3, the development of a spreadsheet calculator. To be fair, Gittins does mention that rule sets can be used as methods in Loops, although this is not elaborated upon and hints about how the object-oriented approach is used in the KEE system can be found in chapter 10 (Applications development using a hybrid AI developments system by John Kunz et al). Nevertheless I would have liked a more AI slant to the use of OOP methods.

Another complaint I have about the OOP chapters is that they contain an annoying level of repeated introductory material. The editor should, I think, have been a little more ruthless or perhaps should have structured the book in a different way. In fact, I have a general criticism of *Artificial Intelligence Programming Environments* that the grouping of chapters does not make it particularly easy to identify important themes. For example, in addition to the OOP chapters (chapters 3, 9 and 11) from parts one and three, there are another four chapters (chapters 2, 5, 6 and 8) distributed between parts one, two and three which cover the themes of accessibility and expressive power. That is, they discuss how AI programming environments contribute to making computers more accessible to novice users and programmers with particular reference to teaching computer programming and how more "natural" programming languages might be developed. Thus in chapter 2 (POPLOG for beginners: a powerful environment for learning programming) Benedict du Boulay describes the use of the POPLOG multi-language environment (particularly Prolog) for teaching. Chapter 5 (Towards more natural programming languages by Steven Hardy) takes a futuristic look at what programmers and users really want in a programming language and

then proposes a solution based on an extended logic language. Chapter 6 (What stories should we tell novice Prolog programmers by Helen Pain and Alan Bundy) reviews ways of presenting the computational model of Prolog to student programmers and chapter 8 (Towards an intelligent help file finder by Tom Khabaza) addresses the problems of finding the relevant on-line documentation in large evolving systems and reviews some of the techniques being currently used and their limitations.

3 Extensibility and very high level languages

Perhaps the most important theme that can be identified in *Artificial Intelligence Programming Environments* is the extensibility of AI programming languages. This is, how the architectures of AI languages enable enhancements and modifications to be incorporated into the language to provide a programming environment that more closely reflects the needs of the user. The most common way in which this is done is to devise a new language, a very high level language. In Allan Ramsay's excellent chapter 4 (Embedding very high level languages) the techniques for implementing very high level languages within the POPLOG environment are introduced in a lucid and practical way. This is followed by another excellent (but somewhat turgid) chapter by Steven Hardy (Towards more natural programming languages). The principal theme of Hardy's chapter is the design of a prototype extended logic language (ELF, for English Like Formalism) that reflects the requirement to describe the temporal ordering of events. In addition, he gives a useful introduction to the logical foundations of Prolog and goes to great lengths to describe how ELF statements might be parsed into Prolog clauses. In describing these ideas, Hardy provides a detailed introduction to why very high level languages are important in AI and how Prolog could be used to develop such a language.

4 Programming methods and AI

In chapter 7 (Partial evaluation as an example of the relationship between programming methodology and artificial intelligence) Kenneth Kahn makes explicit a theme underlying a number of other chapters; the reciprocity between the development of programming methods and AI. Using partial evaluation as an example he describes how having (possibly) partial knowledge about the inputs to a program can be used to optimize interpretation. Partial evaluation is the process of taking general programs and automatically generating more efficient but more specialized versions. In practical terms, partial evaluation can be used to provide efficient interpretation of programs written in very high level languages. As Kahn describes it, partial evaluation flattens the layers of abstraction (interpretation) frequently used by AI programs so that the overheads of multiply-embedded interpreters are removed.

Partial evaluators reason about the results of running a program even if the information available is incomplete. Partial evaluation therefore illustrates how AI and programming methods can be mutually dependent because the implementation of general and more powerful partial evaluators will require knowledge based techniques.

5 Purism versus pragmatism

The last two chapters in this book highlight another contentious area in the AI programming community and a theme that exists as an undercurrent in *Artificial Intelligence Programming Environments*. The contention lies between the methodological (programming language) purists and the pragmatists. Chapter 10 (Applications development using a hybrid AI developments system by John Kunz et al.) and chapter 11 (Loops: a multi-paradigm environment by Martin Gittins) espouse pragmatism and argue for the "horses for courses" view, i.e. that no single AI paradigm elegantly captures all the different types of knowledge and that therefore it makes sense to pick the best tool for each different job. Both KEE (in chapter 10) and LOOPS (in chapter 11) are described as having mechanisms for combining rule-based and object-oriented or frame-based paradigms with both

procedural and declarative knowledge representations. Chapter 10 provides a clear and interesting discussion of how the KEE environment supports the various aspects of knowledge engineering, in particular rapid prototyping; a theme that is curiously absent from many of the other chapters in the book. Chapter 10 also provides a better insight into the use of graphics and the user interface in the AI development environment than do most other articles. My only real criticism of this chapter is that Kunz *et al.* use a naive and unrealistic rule-based representation of a concept hierarchy to support a subjective and poorly enunciated argument that "pure rule-based" systems are inappropriate for this task. Whilst agreeing with the general point that some implementation methods are inelegant for some knowledge representation tasks, the particular example used was not well chosen. The hand of the purists can be seen in many of the chapters dealing with individual languages or approaches (e.g. Lisp chapter 1, Object-oriented programming chapters 3 and 9).

6 Summary

In general, this is a well-written and readable book and it serves a useful purpose in bringing together views on the tools of their trade from AI practitioners with a variety of backgrounds. The difficulties in extracting the themes of the book from individual chapters could have been reduced by more cross-referencing between chapters and more aggressive editing would have prevented some of the annoying idiosyncrasies of wayward contributors. I would recommend it to anyone who had enough energy after mastering one AI language to look around to see what other people might be doing and I expect that it would also be of value to students of AI seeking an introduction to the general issues underlying AI programming languages and their architectures.

Software Engineering Environments
Editor: Pearl Brereton, Ellis Horwood Ltd, Chichester 1988. Pp. 233, £35.00.

The premise upon which Robert Hawleys book *Artificial Intelligence Programming Environments* is built and upon which many of its contributors base their comments is that exploratory programming and an evolutionary style is the most appropriate one for AI. Outside the AI community, and within some of its neater subgroups, this is a highly controversial issue. In particular, the practices of software engineering, data analysis and data modelling are gaining strength both academically and commercially and their principal *raison d'être* is to eradicate unprincipled software development. It is therefore perhaps no surprise that despite the similarity in titles, *Software Engineering Environments* edited by Pearl Brereton tackles entirely different issues from those covered in Hawley's book on *Artificial Intelligence Programming Environments.*

The first thing that I noticed was that in comparison to *Artificial Intelligence Programming Environments*, *Software Engineering Environments* serves an entirely different purpose and the contributions come from a completely different source. In fact, *Software Engineering Environments* is the published proceedings of the third annual conference on Software Engineering Environments held at the University of Keele in April 1987. The sponsors of the conference were the UK Alvey Directorate and it is therefore no surprise to find that the majority of chapters are papers from projects in the Alvey Software Engineering programme. In this respect the book is a reflection of the progress that has been facilitated by the coordination of UK research in software engineering under the auspices of the Alvey programme.

Software Engineering Environments contains a total of 17 chapters taking 233 pages and grouped into 6 sections with a 2 page index. Chapter 1 discusses the broad issues of the principal tool of the software engineering trade, the Integrated Project Support Environment (IPSE) and the way in which they can be evaluated. Chapters 2 to 5 describe a selection of IPSE architectures and contain reports from 3 of the large IPSE and software engineering tools projects sponsored by the Alvey programme. Chapters 6 to 9 cover knowledge based software engineering environments with chapters 8, 9, 10 and 16 describing research into software re-use. Chapters 11 to 13 are reports of the

experience of commercial software houses using existing software engineering products and chapters 14 to 17 cover formal methods and modelling.

There is no introductory section to this book. The first chapter (The evaluation of project support environments for the STARTS user guide by Pulford *et al.*) provides some background since it outlines the present view of an ideal IPSE; otherwise each chapter provides its own introduction. Unfortunately, since each chapter is relatively short (on average, 13 pages) this can sometimes be too limited for the reader with little previous exposure to the aims and terminology of software engineering. According to Pulford *et al.* an ideal IPSE should:

> "... provide a context for software development over the whole of the project life cycle. It will be capable of supporting all sizes of project up to and including, large projects using large teams and developing large complex, high quality software. It will provide the controls and facilities to allow projects to be carried out with optimum productivity and quality. The ideal PSE will be applicable to all types of project and be capable of evolving with a project ... At the same time all this will be integrated to that information can be freely exchanged within the project and all users are given an appropriate and tailored set of facilities in which to carry out their task.... Flexibility and integration characterise the ideal PSE."

I came to *Software Engineering Environments* hoping that, like Hawley's book on AI Programming Environments, I would get an enthusiastic introduction to software engineering with a review of the tool used by software engineers and a comparison of their features and relative merits. In this respect I was a little disappointed, although the reasons are more to do with the style and purposes of the book than the content. The dominant style of contributors is a research report or position statement to fellow scientists, rather than an introduction and review of the methods and tools of software engineering to a lay readership. Where comparisons are made between features of different IPSEs they tend to be at a very technical level (e.g. the database models used in PCTE and ASPECT described in chapter 4). I was also aware that AI techniques had been creeping in to IPSE design and development and I was keen to find out where AI is making a contribution to software engineering. As a rough indication I found that out of 17 chapters, four mentioned the use of an AI programming language (Prolog) and two of an object-oriented language. However, this is only a superficial assessment and it is perhaps more interesting to look at some of the general themes that emerge.

7 AI in software engineering

Looking first to the chapters listed as using AI techniques (chapters 6–10) two contrasting approaches are distinctly visible. Chapters 6 and 7 follow a key theme of software engineering: developing abstract or formal representations of the structure and functions of software that enable the user to describe the software development process and permit the support environment to reason about it. In chapter 6 (ENCORES: an environment for constructing or reasoning with engineered software) D. McArthur describes an approach to automatic programming through algebraic specification in the HOPE language. ENCORES uses a variant of the blackboard architecture (Hayes-Roth, 1985) to represent and reason about the levels of abstraction in the representation of program transformations in HOPE and in the strategies used for transformation. Program transformations are described using a rule-based language (APPEAL) that is also used to represent levels of abstraction in the control of the transformation process. The three levels in ENCORES: problem, strategy and tactic correspond approximately with the levels called problem, tactic and paradigm in the PROGRAIS system described by Christian Gresse and Patrick Raffinat in chapter 3 (PROGRAIS: an experimental programming environment to support transformational development of software).

PROGRAIS is a software engineering approach based on representing the transformations from one description of the program to the next. The aim is to express the structure and history of the

development process rather than just the implementation. PROGRAIS is intended to document and justify the design process and the decisions taken during the project and is based around a structure called a derivation tree. The tree is rooted at the linguistic description of the user's requirements, nodes represent decision points and leaves are either solutions not yet tried, blind alleys or implemented code. The decisions taken by the software engineer at each node in the derivation tree have been classified into general decision types, called by the authors paradigms, that can be linked together to form branches of the tree (called tactics). The application of a tactic is intended "to guide the intuition of a programme" with the purpose of achieving a specified goal. Tactics and paradigms are selected under the influence of meta-level concepts called strategies which might be, for example, modularization or stepwise-refinement.

Although these authors have not done so, it seems to me that the PROGRAIS approach to describing software development would form a good basis for a knowledge based approach to software engineering. The issues of selecting paradigms and tactics and repairing previously erroneous designs are the bread and butter of AI planning and it would be fruitful to see if the categorizations of decisions developed by these authors could be incorporated into one of the contemporary planning systems. If these decision processes and justifications could be subjected to a reasoning system, the resulting IPSE could play an active part in the software design process, rather than, as PROGRAIS does, only providing a framework into which the software engineer must fit the description of project development.

Chapter 7 (Knowledge-based software development tools by Thomas Pressburger and Douglas Smith) describes some of the work in progress at the Kestrel Institute. Their aims are to produce a Knowledge Based Software Assistant that embodies a comprehensive, integrated software development environment with formalized knowledge about each aspect of the programming process. Their approach is an interesting one and proposes the use of three general design principles. Firstly, that software development should be about the use of very high level, yet formal, specifications that can be compiled into efficient code. Software maintenance should then concentrate on the evolution of the specifications, rather than making fine adjustments to the code. Secondly, everything to do with the programming and design processes should be represented in the knowledge base. Finally, there should be knowledge based support for all aspects of the programming process.

The Kestrel Institute is one of the leading centres for research in knowledge based software engineering and this chapter summarizes some of the topics they have tackled, including design tactics for search and optimization algorithms, the use of directed inference for deductive synthesis and program transformation, optimization using finite differencing and the selection of datastructures. These issues are illustrated in more detail in the second half of the chapter with a worked example.

As someone who knows a lot more about AI than software engineering I was a little disappointed by this chapter, particularly as the editor's introduction heralded it as a report on one of the highlights of the original conference. I was particularly looking for how they had treated areas of work that do not fit nicely into formal descriptions and I found remarkably few. The one I noticed and would have enjoyed reading more about was a heuristic measure of simplicity and strength that is used in the RAINBOW inference engine for selecting formulae for application. However, this comment is not an entirely fair criticism since the authors were not writing for an AI audience, but one composed of software engineers who would, presumably, be more interested in the selection of abstract datatypes, cost functions etc.

8 Philosophies of software engineering

Throughout this book it is evident that the predominant philosophy in software engineering is to emphasize the enforcement of software development methodology in order to control quality factors such as correctness and cost. This puts the responsibility for fitting the application to the representational framework upon the software engineer; the IPSE thus becomes the master and the software engineer the slave. In chapter 8 (The re-use of low-level programming knowledge in the

UNIVERSE programming environment) Jeff Parker and Bob Hendley present the alternative view. They see the major problem of software design to be matching the application to the programming or modelling language. Since this is what programmers do relatively successfully, they propose that the most important task is to understand how expert human programmers program and then build tools that support them. As they describe it: "UNIVERSE ... is a program development environment that has been based on the notion of providing support for what programmers *actually* do, rather than what is felt that a programmer *should* be doing". To achieve this, the authors propose a refreshingly different approach (for this book) that exploits recent developments in the area of ergonomic programming aids (e.g. structured editors, incremental compilers, and animated debuggers; which are the very stuff of AI programming environments) and the psychology of programming i.e. programming plans and notational design. Their approach is strongly influenced by the ideas of plan-based representations of programming that were used in the Programmers Apprentice (Waters, 1982) and they provide a useful mini-review of the evidence that has been gathered to support the idea that program plans are the cognitive representations of software engineers.

9 Software re-use

The software engineering issue addressed by the UNIVERSE system is the re-use of software components. Software re-use is currently a hot topic in software engineering and is considered essential for a practical and effective IPSE. The practical basis for the UNIVERSE system is a structured editor in which program plans can be developed at different levels of abstraction and stored in a plan library for subsequent re-use. Keyword indexing is used to permit previously constructed plans to be retrieved for re-use.

The difficulties of retrieving relevant software components for re-use links well with chapter 9 by Murray Wood and Ian Somerville on A knowledge-based software components catalogue. Their principal concern is finding a method of retrieving re-usable software components for the ECLIPSE project that is more effective than searches based on key words such as those used in UNIVERSE. Their approach is to adopt methods from research in natural language understanding to represent software components and their functions using a Concept Case Frame to represent conceptual dependencies (after Schank). By building classification schemes for software function (verbs) and the objects to which they are applied (nouns) together with possible synonyms, they make retrieval of the relevant software component a search for a case frame that most closely matches the query. It is interesting to note that the requirements of a knowledge based software components catalogue are very similar to those of an intelligent help file finder as described by Khabaza (chapter 8 of *Artificial Intelligence Programming Environments*).

Continuing the theme of re-use chapter 10 (Configuration management within an IPSE and its implications for software re-use by B. Dillistone) describes the role of configuration management in the ASPECT project. Configuration management is the process of keeping track of objects created and required by the development environments and how they are related. Much of this chapter is given to the formal definition of the allowed dependencies between nodes in a model of software configuration using the Z specification language. Each definition is however, augmented with clearly written prose so that it is not essential to follow the details of the mathematics.

10 A convergence of software engineering and AI

From an outsider's perspective, the different software engineering philsophies (enforcement and management *versus* support) seem more ideological than irreconcilable and a broader view of how software engineering and AI are developing reveals a considerable number of common concerns. Firstly, both AI and software engineering research are looking for the ideal high level representation language. In AI the emphasis is on closing the gap between the language used by users to describe their application, whereas in software engineering, the emphasis is on finding high level formal

specification languages that permit programs to be proven correct mechanically. In many respects these two goals are the opposite ends of the same spectrum and in both AI and software engineering there are moves towards convergence. So while in software engineering research there is an interest in mixing formal and informal (e.g. graphical) specification languages which explicitly capture the various levels of abstraction in the software design process, in AI there has been a general increased interest in formal characterization of knowledge representation languages.

11 Mixing informal and formal methods

The need to mix informal and formal specifications in software engineering is well illustrated in chapter 2 (Defining formal models of the software development process by Martyn Ould and Clive Roberts) which describes how models of the software development process can be used to facilitate integration. In particular they show how informal diagrammatic techniques and formal requirements modelling languages are used to describe the roles and activities of actors in software development (people and machines) and the processes in which they take a part.

The need for informal and exploratory programming in combination with formal methods is also discussed in chapter 14 (Structure of interactive environments by Alan Wills) which describes some of the architectural issues in the design of tools to support specification using formal methods. Wills is a member of the Alvey Ipse2.5 project and one of the principal aims of this project is to promote the use of formal methods in software design by integrating them with informal methods. His review of what is wrong with contemporary formal reasoning systems and how they might be improved echoes a number of arguments that contributors to Hawley's book make in support of the AI approach to programming. For example, Wills says that writing specifications in a formal language is difficult and rather than the restrictive batch or compile-test-debug models of current formal reasoning tools, what is needed are tools that support exploration such as incremental compilers, immediate-mode structure editors linked to the specification language, navigation aids for finding the status of other parts of the project and high quality output that supports the proper (typeset) notations for the mathematical and logical notations and symbols. Wills goes on to propose an architecture based on "transformers" that provide the mappings between the different methods of representation and presentation that he requires. Transformers are rather like a two-way version of views in database management systems and have a fairly close relationship with object-oriented programming methods. The trend to more formal approaches in AI can be seen in Touretzky's work on the mathematics of inheritance systems (Touretzky, 1986) and in the cross-fertilization of ideas between AI and database research for use in knowledge base management systems (Brodie and Mylopoulos, 1986; Kerschberg, 1986).

Another area where some convergence between AI and software engineering is apparent can be seen in the chapters that concentrate on data modelling in IPSEs. Here, the requirement to increase the semantic richness of existing modelling methods in order to represent adequately the software development process converges with the current interest in conceptual modelling as a basis for very large knowledge base management systems in AI (Brodie and Mylopoulos, 1986; Kerschberg, 1986; Brodie, Mylopoulos and Schmidt, 1984).

12 Increasing the semantic richness

Chapter 2 describes many of the reasons for extending standard data modelling methods and chapters 4 and 5 concentrate on the database aspects of two IPSEs being developed under the Alvey and ESPRIT initiatives. In chapter 4 (A database view of the PCTE and ASPECT) Peter Hitchcock compares the database design principles used in the Portable Common Tools Environment (PCTE) and ASPECT software engineering environments; in particular how the two projects address the maintenance of data independence and data modelling. Although both projects espouse an Entity-Relationship (E-R) data model, he concludes that in the PCTE project, data independence has been compromised and they have not correctly implemented a true E-R model. In chapter 5 (The Eclipse

Data Model—a functional account) John Cartmell and Albert Alderson describe the use of a functional data model for defining an interface that can unify the relational data model used by ECLIPSE based on PCTE methods for storing the data representing the stages in the project life cycle and a network model used for describing the detailed parts of the project such as the source code and documentation. From an outsiders' point of view, Hitchcocks chapter is more interesting, since he provides a clear general overview of using databases as a fundamental part of an IPSE and the benefits of the relational data model (in comparison to the network model in particular) and how the ASPECT and PCTE IPSEs make use of these ideas. In contrast, Cartmell and Alderson's chapter on ECLIPSE makes very few allowances for the inexperienced and gives a detailed exposition of the specification of the ECLIPSE data model in a functional language. This is not impenetrable, but is clearly intended as a technical presentation to the cognoscenti.

The use of AI techniques to increase the semantic richness of software representations is well illustrated in chapter 9 (A knowledge-based software components catalogue by Murray Wood and Ian Somerville) where there is a clear influence from research in natural language understanding. Chapter 15 (Rule based specification of information systems by F. Van Assche *et al.*) also argues for more rich representations of program development in software engineering environments. Although this chapter has been separated from earlier ones using AI techniques, the approach is clearly a knowledge-based one. What Van Assche *et al.* argue is that to bury the business policy of a company implicitly in its computer software leads to inflexible and unmanageable systems. What they propose is that business policy as well as the more traditional aspects of software design (code, specification etc.) are represented as explicit models in a knowledge-rich environment for software engineering business systems. Their rationale for using a rule-based approach is that a declarative representation does not obscure business policy, that rules can be subjected to formal reasoning for verification, that rules capture requirements and policy in perspicuous chunks, that an expert-system-like interpreter can be used to give advice, justifications etc., rules can be added and modified relatively easily and finally that through languages like Prolog, rules can be used throughout the system, from describing high-level policy right down to the implementation. The remainder of the chapter describes how rules are used to provide constraints on an entity-relationship model used for the structural knowledge and to effect changes in the knowledge base in response to events.

13 Toolkits

The theme of toolkits for knowledge engineering and software engineering can be seen as another area of common interest. Until recently, most of the development in AI toolkits has been concentrated on the programming languages and environments needed by knowledge engineers to implement their systems. However, implementation is only one of the tasks of a knowledge engineer and until recently, the development of more integrated support tools has been relatively neglected and what progress there has been is not mentioned at all in Hawley's book. Nevertheless, there have recently been a number of developments in integrated knowledge acquisition and engineering tools that are designed to support the knowledge engineer in the earlier stages of a project; particularly the processing of interview transcripts into conceptual models. These new knowledge engineering tools include KADS (Anjewierden, 1987; Breuker and Wielinga, 1987), Keats (Motta *et al.*) and DETEKTR (Frelling *et al.*). However to my knowledge, there have as yet been no attempts to develop the type of knowledge engineering tools that could assist at the level of project support provided by IPSEs where the issues of team working, software re-use and resource management become important.

The remaining chapters of *Software Engineering Environments* report the experiences of users of first generation IPSEs and two implementation architectures for future generation IPSEs. Chapter 11 (AIMS/38- A working first generation IPSE for the IBM System38 by Jan Lohmeijer and Brian Smith), chapter 12 (Common tool environment—a case study by Cliff Leach) and chapter 13 (BIS/IPSE—automated support in practice by D. Richardson) describe the use of first generation IPSEs in commercial environments. The overall view was that were sufficient resources and commitment

existed to use the IPSE to the full there was an increase in the quality of the software being produced and a reduction in the length of the development life cycle.

The final two chapters in the book, chapter 16 (An environment for prototyping reconfigurable software systems in Prolog by Paola Mello and Antonio Natali) and chapter 17 (Ten15: a portable abstract machine by I. Currie *et al.*) describe new languages for writing executable specifications. Mello and Natali describe CPU (Communicating Prolog Units), a model that extends Prolog with object-oriented features. CPU makes a clear distinction between objects and meta-objects and provides specific mechanisms for linking objects to their meta-level. These features are used to develop a very modular style of software development. The emphasis in the design of Ten15 is the use of strong typing to facilitate integration and program correctness within a high-level language. Ten15 is defined algebraically as an abstraction over general programming (rather than hardware) concepts. It uses a common addressing mechanism and a common system of types through which programs written in different languages and on different machines may interface.

14 Summary

To summarize, *Software engineering Environments* provides a useful overview of recent progress in software engineering tools and methods; particularly from a UK perspective. The majority of the contributions are well-written but the general paucity of introductory material and the level of technical detail in some chapters might prove daunting to readers unfamiliar with software engineering. It would have been helpful if the editor had made more effort to draw out the common themes and important issues; either by linking sections of the book with short discussion (as Hawley did for *Artificial Intelligence Programming Environments*) which helps to smooth over the changes in prose style or by cross-referencing points made by different authors. As it stands the result is rather a peculiar hybrid between a conference proceedings and a multi-author text book.

From the point of view of someone outside software engineering I found more of interest than I expected and I was left with the feeling that AI and software engineering have more in common than perhaps many would suspect. Most of the chapters covering the use of AI techniques in software engineering tended (as might be expected in a software engineering book) to concentrate on the software engineering rather than AI issues. Furthermore, there were not a sufficient number of contributions to give a representative view of the contemporary uses of AI in software engineering. In particular, projects on intelligent assistants/tutors for software engineering were under-represented; but perhaps this is more a reflection of a bias in the UK Alvey software engineering programme. Nevertheless, I think that for readers who are interested in a taste of how AI and software engineering are working together this book is a good starting point and it makes a good companion to the collection of papers on artificial intelligence software engineering edited by Charles Rich and Richard Waters (Rich and Waters, 1986).

References

Hayes-Roth, B (1985). "A blackboard model of control" *Artificial Intelligence* **26** pp. 251–321.

Waters, R C (1982). "The programmer's apprentice" *IEEE Transactions on Software Engineering* **SE-8** (1).

Touretzky, D S (1986). "The Mathematics of Inheritance Systems" *Research Notes in Artificial Intelligence*, Pitman, London.

Brodie, M L and Mylopoulos, J (Eds.)(1986) "On Knowledge Base Management Systems. Integrating Artificial Intelligence and Database Technologies" *Topics in Information Systems*, Springer-Verlag.

Kerschberg, L (Ed.) (1986). "Expert Database Systems" *Database Systems and Applications*. Benjamin/Cummings.

Brodie, M L, Mylopoulos, J and Schmidt, J W (Eds.) (1984). "On Conceptual Modelling. Perspectives from Artificial Intelligence, Databases and Programming Languages" *Topics in Information Systems*, Springer-Verlag.

Anjewierden, A (1987). "Knowledge Acquisition Tools" *AI Communications* **0** (1) pp. 29–38, European Coordinating Committee for Artificial Intelligence.

Breuker, J and Wielinga, B (1987). "Knowledge Acquisition as modeling expertise: the KADS methodology"

In: *Based systems. Proceedings of the first International Workshop on Knowledge Acquisition for Knowledge*.Addis, T (Ed.) Reading University.

Motta, E, Eisenstadt, M, West, M, Pitman, K and Evertsz, R (1986). *KEATS: The Knowledge Engineers Assistant*. Human Cognition Research Lab., Open University, UK.

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

Rich, C and Waters, R C (Eds.) (1986). *Readings in Artificial Intelligence and Software Engineering*. Morgan Kaufmann Publishers, Inc., California.