Engineering Knowledge-Based Software for Large Embedded Systems*

M. Bouteldja
Helsinki University of Technology
Faculty of Information Technology
Department of Computer Science

Abstract

The main aim of the work described in this thesis was to investigate how knowledge-based techniques could be used for developing embedded software in large systems We identify and discuss the various constraints that characterize such systems and their implication on knowledge-based software development. We particularly examine the role of the programming tools in meeting these constraints. A programming language integrating abstract data types and the production rule paradigm, together with a supporting programming environment, is presented as today's realistic approach for introducing knowledge-based software techniques in large embedded systems. The findings are substantiated both by research work in the area and practical experience gained in knowledge-based software development for a digital switch, which is a typical example of a large embedded system.

*Thesis submitted to the Department of Computer Science, Helsinki University of Technology, in partial fullfillment of the requirements for the degree of Licenciate in Technology.

Contents

1 Preface

The work described here has been carried out during the years 1986 to 1989 at Nokia Telecommunications and Helsinki University of Technology. It is part of the ExBed project (Expert System Framework for Embedded Applications) which falls under the Finnish Programme for Research in Information Technology initiated and supported by the Technology Development Centre of Finland together with private industry. The main objective of the ExBed project was to determine to what extent and what type of knowledge-based techniques could be used in embedded systems software. The project adopted a pragmatic attitude in the sense that results were to be based on concrete experimentation with the proposed techniques. The main emphasis of this thesis is to show why the proposed techniques are suitable for knowledge-based software development in large embedded systems, with the arguments based on practical application development at Nokia Telecommunications.

The thesis is based on an evolutionary series of papers published during the course of the project, each one emphasizing some results used in the subsequent stages of the work. The purpose of this report is therefore to introduce the reader to the project and summarize the main results presented in the following papers:

1. Nuutila, E., Kuusela, J., Tamminen, M., Veilahti, J., Arkko, J. and Bouteldja, M. XC - a Language for Embedded Rule Based Systems, *SIGPLAN Notices* 22, 9 (Sep. 1987),23-32.

2. Bouteldja, M. Dxpert: An Expert System for Complex Alarm Interpretation in the DX 200 Digital Switching System. In *Proceedings of the International Expert Systems Conference* (London, June 2-4), Learned Information, Oxford, 1987.

3. Bouteldja, M., and Lepistö, P. On Some Problems in Applying Artificial Intelligence in Large Industrial Systems. In *Scandinavian Conference on Artificial Intelligence* , T. Danielsen, Ed., IOS, Amsterdam, 1988.

4. Bouteldja, M. Engineering Embedded Knowledge-Based Software for a Digital Switching System, In *Proceedings of SETSS'89* , IEE, London, 1989.

5. Bouteldja, M., Arkko, J., Kuusela, J., Nuutila, E. and Tamminen, M. Building an Embedded Knowledge-Based Application using Abstractions, In *Proceedings of the 2nd Scandinavian Conference on Artificial Intelligence* ,Tampere, Finland, 1989.

The first paper presents XC, an extension of the C++ programming language that combines abstract data types with the production rule paradigm. It is shown why it is particularly suitable for embedded knowledge-based applications. In particular, it is compared to OPS83. My share of the work has been to benchmark OPS83 and interpret the results.

The second paper is about a knowledge-based alarm interpretation system that I have built for Nokia Telecommunications. The structure and operation of the system is presented together with a discussion of the suitability of OPS83 as an implementation language.

The third paper reports on the results of a study I led at Nokia Telecommunications to determine what were the major problems we faced when developing knowledge-based applications for a digital switching system. This was primarily all my work except the part on knowledge acquisition where Lepistö provided some help.

The fourth paper discusses the requirements of embedded knowledge-based software development in a digital switching system. It is argued how a programming environment based on abstract data types combined with rule-based programming meets the requirements.

The fifth paper presents the XE programming environment and how it has been used to build an embedded diagnostics application. This is also primarily my work. The co-authors provided the parts on the description of the XE programming environment.

Acknowledgments

I wish to thank several people for the help they have provided during this long project:

- Nokia Telecommunications for sponsoring this work, in particular R&D managers Timo Vesterinen and Tuomo Huhtanen who constantly supported the initiative despite negative forces in the department.
- Timo Lehtimäki for building with me the XEDA embedded application.
- the ExBed research group at the Department of Computer Science (HUT), including Markku Tamminen, Juha Kuusela, Esko Nuutila, Vesa Hirvisalo, Jari Arkko and Jussi Rintanen, who built the XE programming environment and provided scientific and technical cooperation.
- Professor Markku Syrjänen for being my thesis supervisor.
- my family, for supporting me through the long dark working winter days.

2 Introduction

Knowledge-based technology promises to provide the means with which knowledge intensive problem solving tasks can be computerized. It is therefore no wonder that we become interested in understanding and using this technology in the context of large embedded systems like a digital switch, which are generally knowledge rich domains. Knowledge-based technology has had some success in the implementation of consulting expert systems in areas where knowledge is readily available from known sources and the resulting system is self-contained, operating independently of any other existing larger computerized entity. Interesting and potentially more beneficial applications of the technology in large embedded systems, are found in a different context. For instance, applications will be embedded within other software in the product and will generally have no user interface. Furthermore, we are interested in the application of knowledge-based techniques and methodology for engineering software in general, not just for specific applications like expert systems. There are at least two important justifications for this interest:

- knowledge-based techniques allow the software to take a step away from the computer and toward the problem domain. This will help to reduce the distance between the specification of a problem solution and its implementation. Development time and errors will be cut down because the complexity of the mapping process is reduced.
- Domain knowledge is a major factor in software development productivity, quality and costs, as exemplified in a recently carried out empirical study [Curtis et al. 1988]. Knowledge engineering puts domain knowledge in an important position and therefore brings to traditional software engineering a useful attitude and set of methodologies, like knowledge acquisition.

In this section, we will define what we mean by large embedded systems and delineate their characteristics. We will then discuss how these characteristics have an effect on knowledge-based software development. The following sections will then present the arguments that have led us to adopt the methodology presented to implement knowledge-based software for these systems. The beneficial inputs to traditional software engineering are also discussed. We conclude with a summary of the main ideas resulting from this work.

2.1 What are embedded systems?

An embedded system is a computer-based entity which provides a service to the end-user in the form of a specific product that hides internally the computer (hardware and software) part. The user is not necessarily aware of the existence of the computer whithin the product. The french term "système embarqué", which literally translated stands for 'a system that is on board of something', also stresses the inclusion of the computer inside a larger entity. Embedded systems range from digital switches, through airborne navigation controllers, down to photocopying machines. A digital switch will most likely be based on a powerful mini-computer or a distributed architecture of microcomputers, and would be considered a large embedded system. A photocopying machine on the other hand, will most likely contain a microprocessor-based computer, and would be considered a small embedded system. The distinction is primarily a factor of the amount of software involved. If the software development can be characterized as a programming-in-the-large project, then we are dealing with a large embedded system.

2.2 Characteristics of large embedded systems

It is of utmost importance to consider the constraints within which large embedded systems exist in order to introduce successfully knowledge-based technology in these systems. The following are a list of features that are directly relevant in the development of knowledge-based software.

- *tight memory and execution time constraints*
  It should come as no surprise that memory is scarce in embedded systems. This is not due, as one might believe, to the fact that currently used microprocessor architectures do not offer a large addressing space. The reason is that memory usage is minimized for reasons of cost-effectiveness. In embedded systems, the cost of adding extra memory is high because it is not a simple matter of adding memory chips. It may involve extra pcb cards, extra connections and changes to the software (e.g., memory diagnostics software). Furthermore, mass storage is usually not available in these systems. Even in cases where it is available, data is copied into RAM to satisfy fast access requirements.

  Embedded systems are often involved in real-time control tasks and have therefore speed requirements dictated by the real-time process they are interacting with. Therefore execution efficiency and more fundamentally predictability are important issues.

- *high reliability requirements*
  Reliability requirements are usually very high in embedded systems. For instance, digital switching systems are only permited a total maximum of 3 minutes down time per year. The requirements are not exagerated: in a society increasingly dependent on telecommunications, the consequences of a telephone exchange failure can be catastrophic. Embedded systems are often involved in safety-critical tasks, so reliability is a major issue. In large embedded systems, the reliability of

the hardware has been satisfactorily under control. Unfortunately, the same cannot be said about software reliability. Since software constitutes a major part of these systems, its reliability becomes a major preoccupation.

- *poor testing environments*
  Embedded systems always have their software developed in a hardware environment different from the target machine. Although the sophistication of the development environment can be improved, it remains that most embedded systems are tested in the target system itself which does not provide good debugging facilities. Furthermore, in large embedded systems, testing in the target configuration is a very costly activity. That is not only due to poor testing facilities but also to the overheads involved in setting up a testing session.

- *programming-in-the-large aspects*

  - disciplined software engineering
    Large embedded systems software development always follows a disciplined software engineering approach in order to satisfy large project management requirements. Prototyping approaches are not currently favoured.
  - modular decomposition of software
    Large embedded systems fall under the category of programming-in-the-large and programming-in-the-many environments. Typically, over 1 million lines of source code and over 20 software designers are involved. In these environments, the possibility to split the software into smaller manageable entities is vital. This modular decomposition is needed to facilitate concurrent software development and improve understandability and maintainability of the software. It also enables the software to be configured easily for different releases of the system, which may be dictated either by different markets or different levels of functionality.
  - single procedural language environments
    Large embedded systems tend to be programmed in one single procedural high-level language, with critical parts sometimes coded in assembler. Multi-language software development is not favoured because of the overheads and management fears associated with such approaches. It becomes more difficult to shift around project members and programming standards have to specified for all the languages used.

- *thin spread of domain knowledge*
  It is a fact, in large embedded systems, that domain knowledge is thinly

spread across a large population of sytem, hardware and software designers. That is, because of the vastness of knowledge involved and the hierarchical structure of the organization, each designer is at best expert in the area for which he has designeda solution.

2.3 Effect on knowledge-based software development

Below is a list of the main effects of the previously listed characteristics on the development of knowledge-based software for those systems. These points will be elaborated upon in the following sections either as part of the results of this work or as suggestions for further research.

- *Efficient languages and knowledge-based techniques*
  Because of execution time and memory constraints in embedded systems, the implementation language used and the knowledge representation techniques it supports, must be optimized for efficiency. At least for this reason, frame-based languages like KEE [IntelliCorp 1985] and logic-based languages like Prolog [Bratko 1986] cannot be considered. Production rule-based representation and inference techniques can on the other hand be implemented efficiently enough to be used in embedded systems. Furthermore, since procedural programming will need to be mixed with knowledge-based techniques, it is important that the interaction between the two is also achieved efficiently.

- *static strong type checking and verification*
  Because of the high reliability requirements of embedded systems and the poor testing environments usually associated with them, it pays to have as much verification as possible performed in the development environment. We believe strongly that a language with strong type checking performed at compile-time helps considerably in improving reliability in a cost-effective manner. Domain knowledge should be verified as much as possible in the knowledge acquisition phase both by having it checked with the source and using consistency checking tools.

- *Modular facilities*
  The necessity to decompose software in small manageable entities and memory constraints found in large embedded systems, lead to the important requirement that modular decomposition of knowledge-based software should also be possible and encouraged. When using production rules, for instance, it should be possible to split the knowledge into different rule sets, each one, perhaps, with its own associated working memory.

- *Interface to existing software*
  Unless we are in the very rare situation where a large embedded sys-

tem's software is being developed using a hybrid language (procedural and knowledge-based), the question of how to interface with existing software has to be tackled. Knowledge-based programs must be able to interface to already existing software. This is because existing software will offer possibilities for using operating system services, accessing data bases, and in general communicate with other software. This communication also needs to be from the existing software to the knowledge-based software in cases where it is offering a service. This vital two-way communication capability can be partly solved by the programming language used to develop knowledge-based software and partly by the environment where this software will be running. The former will have the difficult task to reconciliate different programming paradigms as well as possible.

- *Integration of knowledge acquisition to disciplined software engineering* Knowledge acquisition plays a central role in knowledge-based software development. Its introduction and acceptance in a disciplined software engineering environment as an activity in the software life-cycle coming under project management is a very significant goal to aim for. Not only will it strengthen the position of knowledge-based software but will undoubtedly raise the value of domain knowledge in conventional software.

3 Rule-based systems

3.1 Suitability

The rule-based paradigm practically remains the most widely used implementation technique in knowledge-based software development. Its techniques and generally associated benefits are described in [Hayes 1985]. In the embedded systems area, most of the knowledge-based applications developed have used production rules as the main knowledge representation paradigm [Fagan 1980, Karonis et al. 1987, Sachs et al., Theuretzbacher 1986, Wright et al. 1986, Årzen 1986]. Nevertheless, it is not clear from this litterature how large and integrated these systems were. Specifically, there is no report in the scientific litterature on knowledge-based software development in programming-in-the-large environments, except for some experiment on multi-paradigm software development [Däcker et al. 1985]. We consider the production rules paradigm suitable for large embedded systems because of the following points:

- The data-driven inference strategy is simple enough to comprehend and corresponds to a general model of reacting to an appropriate external stimuli, commonly used in large embedded systems software development. An example of that is the SDL specification language [CCITT 1984] used in the telecommunications sector and which is based on the finite state automaton model of computation.
- Useful applications in large embedded systems depend on very wide and complex domain knowledge which is difficult to model. Therefore deep models are extremely difficult to construct and a realistic alternative is shallow knowledge, best represented with rules.
- Efficient run-time implementations of the paradigm are possible.
- Efficient development tools, like compilers, not necessitating heavy investment in special-purpose machines, are possible.

Any software development project in embedded systems will also involve parts that are best coded using a procedural language. It is therefore important to be able to mix, as freely as possible, both paradigms in the same application. We believe that supporting these two pradigms in the same language is more suitable for large embedded systems. This is due to the following facts:

- Technically, the interface between the two paradigms is likely to be implemented in a smoother and more efficient manner. This will mean that the user can mix the two paradigms more easily than if the two paradigms were supported by two different languages with the interface built in an ad-hoc fashion.

- There is only one compiler to maintain and one can be sure that there will not be incompatibilities of the interface in new versions. This cannot be guaranteed when the knowledge-based and procedural components are in different languages.
- In practice, it will be easier for the user to implement conventional and knowledge-based software using the same language. There will be no need to memorize different lexical, syntactical forms and possibly different development environments.

3.2 OPS83 versus XC

The material covered in this section is based on paper 1 to which the reader is refered for more details.

OPS83 is the latest and most powerful member of the OPS family of rule-based languages [Forgy 1984]. It is a hybrid language offering the possibilities to mix the procedural and rule-based paradigms in the same program. It has efficient implementations in C, which also means an interface to C, that can be found in minicomputers, workstations and standard microcomputers. It is a typed language in the style of Pascal. OPS83 uses the rete pattern matching algorithm [Forgy 1982] which is widely believed to be the fastest algorithm for finding the set of instantiable rules in a production system. However, there has been unsubstantiated claims that the state saving mechanism used in the Rete pattern match may not be worthwhile [Mc Dermot et al. 1978b].

XC is a language specifically designed for embedded knowledge-based software development [Nuutila et al. 1987]. It is an extension of the C++ programming language [Stroustroup 1986] which combines efficiently the production rule paradigm with abstract data types and object-oriented programming. XC is significantly more suitable for developing embedded knowledge-based software for the following reasons:

- OPS83 imposes restrictions on the type of objects that can be manipulated in working memory. Only the record type called element is allowed. In XC, a working memory can contain objects of any abstract data type and the same objects can be manipulated by procedural code.
- Although OPS83 supports the module concept for decomposing programs into manageable parts, all rules belong to the single linear rule set and communicate through one global working memory. XC allows many working memories and many rule sets in the same program. This increases the modularity and execution efficiency of the application.
- The Rete pattern matching is unavoidable in OPS83. Yet, it can be problematic when used in applications where the assumptions made by the algorithm do not hold. For instance, in an application monitoring

a fast real-time process, the temporal redundancy assumption does not hold; namely data in working memory will be changing very fast. In these circumstances, the Rete algorithm will slow considerably the execution of the system. Furthermore, memory consumption is very high in OPS83 since the state of the partial matches is saved in the Rete network. XC provides a simple minded algorithm that iterates over all rules looking for instantiations at each match cycle. Thus memory consumption is lower because no partial matching state is saved; execution efficiency is also improved if, as expected in embedded systems, the knowledge is decomposed into small manageable rule sets.

We tested the McDermot conjecture by benchmarking XC against OPS83 using a simple minded algorithm on a few small application examples [Sigplan]. We found that XC ran 1.5 to 9 times faster than OPS83 depending on the application. Similarly, XC consumed 64% up to 800% less memory. Recently, a new pattern match algorithm has been designed and found to be more efficient than the Rete algorithm [Miranker 1987]. This latter design was prompted by work on parallel processor for expert systems production where the Rete pattern matching was found unsuitable for parallel execution [Stolfo et al. 1984].

The main drawback we found with XC was due to the C++ language on which it is based. C++ contains several type insecurities some of which are summarized in the work of Sakkinen [Sakkinen 1988]. This, we felt, reduced support for reliability that we considered of paramount importance in embedded systems. Furthermore, we felt that a programming environment supporting the language would be necessary for paractical knowledge-based software development.

4 Applications in large industrial context

4.1 DXpert

This sub-section summarizes the work described in paper 2 to which the reader is refered for more details.

DXpert is an expert system for the interpretation of alarm messages originating from the DX 200 digital switching system. It can analyze an alarm dump and provide a fault diagnostic together with recommendations on remedial action. It is implemented using OPS83 on an MS-DOS PC loosely connected to the switch through a serial communications link. The system contained approximately 100 OPS83 rules with an average of 5 condition patterns per rule. Our conclusions resulting from work with this application can be summarized as follows:

- The importance of the availability of both the procedural and rule-based paradigm in the same language was confirmed. We also discovered that there are often situations in embedded systems applications where we wish the procedural part to call the rule-based part, and not vice-versa. This kind of interface, from procedural to rule-based, is much more likely to be built when both paradigms are part of the same language. Furthermore, a large embedded system, is likely to contain hundreds of litteral definitions pertaining to different parameters of the system. When another language is used for knowledge-based software development, all these parameters have to be redefined with problematic consequences on maintenance.

- Tailorability of the inference engine to the application, as afforded by OPS83, is a must. The standard conflict resolution strategies like MEA found in OPS5 [Brownston et al. 1985] have an important flaw: they are unaware of the characteristics of the application environment. We have found that in our application, for instance, only specifity was strictly necessary for conflict resolution. Refraction was included as an extra safety precaution but was strictly speaking redundant since a fired rule always deleted its instantiated working memory elements in the action part.

- The absence of a modularity mechanism for rules was a drawback. Rules in DXpert were found to fall neatly into three categories: hypothesis-generating rules, testing rules and recommendation rules. Furthermore, the operation of the system was so structured that only one category of rules was relevant at any particular stage of the problem solving. For instance, recommendation rules were only needed at the end when the fault hypotheses were confirmed. Yet, in OPS83 all rules must belong to the one and only active linear set of rules.

- The strict type checking of OPS83 was very helpful for preventing certain kinds of mistakes. Nevertheless, the absence of an integrated programming environment providing support for recompilation, debugging and version control was felt.

4.2 Problematic issues

The points made in this sub-section are drawn from paper 3 to which the reader is refered for more details.

We identified some major problem areas to be faced when building knowledge-based software for large embedded systems [Bouteldja et al. 1988]. Although some of these problems are also found in other environments, their importance and the way they are handled may differ. There are three major aspects to watch for when building knowledge-based software for these systems:

1. **Large and Distributed domain knowledge**
   Knowledge acquisition in large industrial systems presents the extra problem of having to handle large and distributed knowledge. The distribution of domain knowledge is inevitable in these environments since no one person can master the complexity of a large system. Using traditional interviewing techniques for knowledge acquisition becomes prohibitively time-consuming. Furthermore, it is much more difficult to motivate a large number of experts to give some of their time to knowledge acquisition. To minimize the risks of failure because of these problems, the approach we recommend is to select only applications whose domain knowledge is well-documented or can be extracted from at most a couple of experts.

2. **Integration to main stream software engineering**
   In order for a knowledge-based application to be successfull, i.e., become a product, its development must be integrated to the main software development effort. This is because software engineering in large embedded systems follows a disciplined methodology involving preset milestones, documentation standards, quality reviews, software configuration management and more generally control over the development process. Knowledge-based software that has not been through such a development process will not be accepted in the final product.

3. **Reliability**
   In industrial environments, without assurance of reliability, knowledge-based applications are doomed to failure. Although conventional software engineering has not completely solved the problem of reliability, certain aspects of knowledge-based techniques do not facilitate this task. Certain tools and techniques used still lack a sound theoretical foundation.

Expert system shells, for example, support several kinds of knowledge representation formalisms without a formal basis for their integration [Inder 1988].

4.3 Benefits of knowledge acquisition

It is very interesting to note that conventional software projects do not include a knowledge acquisition phase as such. Nevertheless, we all know how important it is to understand the problem domain area and associated knowledge, before being able to write any software for it. In conventional software development, the application domain knowledge is buried between other types of information in requirements, specification and design documents. Because no special attention is given to it, it will often be incomplete and erroneous. Knowledge acquisition gathers the domain knowledge in one place, puts it into semi-formal form (decision trees, rules, etc...) thus reducing ambiguity, and verifies as much as possible the correctness of the knowledge gathered. This is a major point given the painful fact, realized today through empirical studies in software engineering [Curtis et al. 1988], that incomplete and erroneous knowledge of the application domain is responsible for the most costly mistakes in software development. Hence we believe it is high time knowledge acquisition formed an integral part of the software development life-cycle. In this respect, software engineering has something important to learn from knowledge engineering.

5 Programming abstractions approach

This section covers both papers 4 and 5, where more details can be found.

The main benefit of knowledge-based software is to make application domain knowledge more visible in the software. Abstractions mechanisms provided in modern programming languages like [Shaw 1984], offer a good foundation for bringing knowledge visibility in programs. Of particular importance are abstract data types [Liskov et al. 1974] whose development was spurred by the realization that there was a significant gulf between programs and problems. Even if they are not knowledge abstractions, as rightly pointed out in [Abott 1987], they provide a foundation for bridging the gulf between program-level and knowledge-level abstractions. This view is supported by current work at Tecknowledge where abstract data types are an essential component of ABE, an environment for engineering intelligent systems [Erman et al. 1988]. In this environment, abstract data types are for example used to describe the behaviour of structured objects.

In addition, abstract data types offer good support for reliability and programming in the large. Reliability is supported through strong type checking at compile-time time and exception handling facilities at run-time. Programming-in-the-large is supported through the modularity and encapsulation of abstract data types (ADT). Object-oriented design [Meyer 1988] can be used to decompose the problem into manageable parts to be implemented by different people and isolation of the interface from the implementation insures ease of maintainability. Abstraction-based programming can also support rapid prototyping, which is both a methodology currently well accepted in knowledge engineering and a very promising area of development in traditional software engineering [Luqi et al. 1988]. Reusability is supported at the language level through type parameterization.

5.1 The XE programming language

This sub-section is very much a short description of the XE programming language with an emphasis on its rule abstraction. For further details the reader is refered to the *XE Reference manual* (Report TKO-A26) and *Programming in XE* (Report TKO-C33). The discussion is based on paper 4 where more details can be found.

XE is a general purpose programming language and environment [Arkko et al. 1989] design owes very much to CLU [Liskov et al. 1977, 1986]. XE supports the use of abstractions in program construction. Beside the procedure abstraction available in all programming languages and the iterator and data abstractions supported by CLU, XE provides the programmer an integrated rule abstraction.

A data abstraction or abstract data type (ADT), defines a set of objects and a set of primitive operations to create, examine and manipulate them. The data types of XE are called abstract because the concrete representation of an object can be seen only by routines inside the definition of the data type. In this approach the capabilities of an object and the applicability of abstractions is defined, not by membership in a type hierarchy, but by the external interface of the object and the interface requirements of the abstractions. These data types can be parameterized (genericity) increasing the expressive power of the language and its support for reuse.

Iterators are an XE abstraction that allows performing a block of code for each element in an abstract collection of elements, without disclosing the representation of the collection. One can think of the iterator as a control abstraction generalizing the *for* construct found in current programming languages, which can iterate over ranges of integers. The XE *for* construct can iterate over collections of any type of object. The selection of the next object in the collection is done by the user-defined *iterator*. The iterator produces the objects in the collection one at a time and each object is consumed by the *for* construct in turn. Iterators are central to the rule concept, as embodied in XE.

5.1.1 The rule abstraction

Rules implement a condition-action abstraction derived from the production system model. In XE, a rule is a special kind of iterator that computes a sequence of instantiations. An instantiation usually contains data that satisfies the condition of a rule and an action that should be applied to the data. It may also contain data that is used when comparing instantiations. Rules have a fixed number of arguments and can be parameterized just like a procedure or an iterator. They can also signal an exception. Rules can be invoked in *for* and *iterate* statements just like iterators. An important feature of rules in XE is that they can access any other XE data, either through actual arguments passed to them or through variables within their scope. Variables can be made visible to rules by using the *binds* mechanism. An XE rule is of the form:

rule ::= **rule** arguments [yields] [signals] [binds]
    { equate or own variable declarations}
    when
        conditions
    :
        [statements]
end

The rule abstraction of XE does not contain concepts such as rule set, working memory, conflict resolution, rule firing, and inference engine, which are typically used in rule-based programming. All these concepts can easily be built in XE itself, using the other abstractions, typically as follows:

- Rule set
  An abstract data type whose concrete representation includes the rules belonging to the set. Amongst the operations provided by the data type, one or more iterator abstractions that yield the rules in the desired order would be included.

- Working memory
  An abstract data type providing operations to put and retract XE objects. If a working memory is to be used in a fashion similar to the pattern matching supported by OPS5 and similar inference engines, it would normally include one or more iterator abstractions to access relevant elements. The working memory could be made generic by using type parameterization. The type of XE objects to be manipulated would then be supplied as a parameter when the working memory is first created.

- Conflict resolution
  A conventional procedure abstraction that compares two rule instantiations and chooses the best one according to the strategy coded.

- Rule firing
  An invocation of an action yielded by a rule instantiation.

- Inference engine
  A procedure abstraction that iterates over rules in a rule set and their instantiations.

In addition to the rule abstraction, the XE language has several features that are not found in CLU. We will present only those that have an effect on embedded knowledge-based software development.

- **complete type checking at compile-time**
  Because XE does not have the type *any*, it can perform complete type checking at compile-time (that is not the case in CLU). This feature is important in embedded systems where testing environments are poor and debugging at run-time is costly. We believe compile-time type checking generally improves the correctness of programs developed for embedded systems in a cost-effective manner. Knowledge-based software is no exception.

- **user-defined type parameters**
  In XE, it is possible to have any constant type as a type parameter, whereas in CLU only simple built-in types are allowed. This feature, for

instance, allows us to write a generic working memory where the type of the object to be manipulated can be any user-defined constant type. Embedded systems software are usually programming-in-the-large environments and such a feature is extremely important for providing a reusable library of knowledge-based software.

- **static memory allocation**
  One of the problems that has hampered the use of knowledge-based techniques in real-time embedded systems is the prohibitive time spent by the software on garbage collection. Using the *static* construct in XE, it is possible to define static objects that will not be subject to garbage collection.

- **external language interface** XE provides a basic interface to a procedural programming language like PL/M or C. External procedures can be called from XE using this interface. The interface can be improved through the use of abstract data types.

- **efficient code generation**
  Many code generation and optimization techniques have been developed in order to make the code produced by the XE compiler more efficient. Efficient code generation from a program based on abstract data types is difficult and still a subject of research [Ganapathi et al. 1989]. This is due to the many levels of routine calls resulting from abstraction. To remove this effect a good compiler must be able to expand calls to small routines inline. Other features to be carefully considered in code generation are iterators and exceptions. In the case of XE and its rule abstraction, iterator optimization is especially important. Inline coding, tail recursion elimination, jump propagation and redundant code elimination greatly enhance the quality of the code produced from XE rules. The XE optimization phases produce good results: if array bounds checks are left away the code produced by the XE compiler has been comparable to that produced by the Microsoft C compiler.

5.2 The XE programming environment

This sub-section is very much a short description of the XE programming environment. For further details the reader is refered to the *XE Programming Environment User's Guide* (Report TKO-C35), *Notes on XE Run Time Characteristics and Code Generation* (Report TKO-C37) and *XE Programming Environment, Implementation Notes* (Report TKO-C40).

The XE programming environment has been especially designed to support the development of XE programs in a programming-in-the-large context. Thus program integrity, version management and access synchronization are well supported. Furthermore, it has several features that help improve the productivity of the programmer. For instance smart recompilation minimizes the time it takes to recompile a system after a change, and dynamic linking improves the debuging process by allowing a program to be modified while it is running.

The basic entities manipulated in the XE programming environment are *constants*, *procedures*, *iterators*, *rules* and (abstract) *data types*. These entities, all refered to as *modules*, are the only entities that can be operated on in the environment. The concept of files is taken care of by the storage system and is invisible to the programmer. To facilitate the development and control of large systems,modules can be combined to form larger structuring entities. A set of modules performing a task can form a *system*, the topmost hierarchical entity in the environment. There can exist many *generations* of the same system which correspond to the historical development of the system. A generation has attributes that describe its status.

The XE programming environment operates on a Lisp workstation and takes advantage of the Genera operating system [Walker et al. 1987]. Programs can be developed and unit tested on the workstation before generating code for the target system where further tests will be carried out. Because compiler tools like ACK [Tanenbaum et al. 1983] have been used, code generation for different target processors is facilitated. The current version generates code for the Intel 8086.

Thus, the XE programming environment is attractive in large embedded systems knowledge-based software development primarily because:

- it offers support for programming-in-the-many and programming-in-the- large.
- it offers facilities for generating code for different target processors.

5.3 An example application: XEDA

This sub-section is based on paper 5 to which the reader is refered for more details.

5.3.1 The application environment

The DX 200 is a fully digital switching system with a distributed control architecture. It is based on a network of Intel 16 bit microcomputers communicating through a fast message bus. Tasks are performed by communicating processes running concurrently under the DMX real-time operating system. All call control microcomputer units are duplicated. The operation and maintenance computer (OMC) serves as an interface between the operating personnel and the exchange. The DX 200 software development project involves over 200 software designers with the total of source code lines exceeding 3 million. As such, it is undoubtedly a programming-in-the-large effort.

Reliability requirements are severe; the maximum down-time per year is 3 minutes. In order to provide uninterrupted service, the exchange must be capable of automatically recovering from faults. Thus, the most important task of the maintenance system is to guarantee continuous operation of the exchange even in fault conditions. The maintenance task is carried out by several cooperating processes. Faults can be detected by the supervisory process or any other process running in the unit. For each fault detected, a message is sent to the alarm handling process in the OMC. The alarm process prints the alarm in a readable format for the operator. On the basis of the amount and type of alarms received for a particular unit, the alarm handling process decides whether a unit is faulty and activates the recovery process. The recovery process takes the faulty unit out of operation and replaces it with the stand-by duplicate. The diagnostics process is then activated to try and locate the faulty plug-in unit.

5.3.2 Functional description

It is the diagnotics process function mentioned above that was selected as a candidate for improvement through knowledge-based techniques. When the diagnostics process is invoked, it performs a suite of tests on the different hardware components making up the unit. These tests were performed always in the same fixed order. XEDA, which stands for XE Diagnostic Adviser, tries to make the process of test selection more intelligent. Its task is to guide the diagnostics process to first try out tests that are most likely to localize the fault. The advice is given after a knowledge-based analysis of the current alarm situation. In the event that our test suite adviser is not able to infer anything from the current alarm situation because of its limited knowledge, the diagnostics process continues testing with the default settings.

XEDA runs as a process in the Operation and Maintenance Computer and communicates with other processes developed with conventional programming techniques through the DX operating system message passing mechanism. This approach is similar to [Theuretzbacher 1986] where an OPS83 program is put in a CHILL process envelope.

The main logic driving XEDA is the following:

1. Receive task message
2. Start up inference engine which will
    - fire rules that obtain relevant alarms

- fire rules that infer a better test suite order from those alarms

3. write new test suite order, if applicable, to the diagnostics process file
4. send task completed message
5. go back to step 1

Step 3 involves checking that the new test suite order arrived at, is safe and profitable.

A test suite order is safe if it respects certain order constraints imposed by the architecture of the DX 200. For instance, the subscriber switching card should be tested before the group switching card, otherwise there is a risk of getting erroneous results. The approach taken here is not to reject an unsafe test order result but rather to correct it to satisfy the safety constraints. This is because a test suite can be long and comprise several rearrangements to the original order. If the order is unsafe only at some point in the sequence, the rest of the sequence might still be in a profitable order.

A test suite order is profitable if the time taken to locate the fault using this order is significantly shorter than the time it would take using the original test order. Because XEDA's execution time is in the order of a few minutes, we consider significant any time gain greater than 5 minutes. Main memory tests, for instance, can take up to 45 minutes. This makes any new test order relegating these tests to the end highly profitable.

5.3.3 Knowledge representation

XEDA's structure reflects a methodology supported by XE where problem decomposition is based on the recognition of abstractions. The abstractions built in XEDA correspond to the objects involved in its domain area (dx files, alarms, diagnostics tests, etc...), together with control logic that uses the capabilities of these objects to solve the problem at hand.

XEDA uses rule sets which are implemented as XE abstract data types. These rule-set data types contain XE rules augmented with other useful information for inference (like rule weight) and provide operations to manipulate these rules. There are two types of rules in XEDA. The first type has the knowledge used to decide which alarms are to be examined in a given situation (depending on, for instance, which computer unit is being diagnosed). The second type is the set of rules containing the knowledge for deriving a good test ordering. We decided to have both these two types of rules in the same rule set initially because we thought it would be easier for maintenance purposes not to have to know where to add an extra rule. This does not have any significant effect on efficiency since XE offers language-level optimization features that permit us to examine only a particular type rules.

The header of the definition of the Xeda rule set looks as follows:

dx-dia-rule-set = **datatype is** *create, reset, rules, retract*
...

The concrete representation used is an array of records having a rule type as one of the fields:

rule-rec = **record** { *used*: **bool**,
dx-rule: *dx-dia-rule-rec* }
rule-array = **array**[*rule-rec*]

rep = **record** { *contents*: *rule-array* ,
count: **int** }

The *create* operation initializes and sets up the rules. We delete a rule after it is fired once using the retract operation. This way we don't have to watch for cyclicity in the inference engine.

The *reset* operation is used to recover deleted rules when restarting XEDA for a new inference task. Since the deleting only involves setting a boolean flag, this operation is trivial.

A sample *dx-dia-rule-rec* looks like this:

```
{ id: "cac1007",
  rule-def: rule( a-wm: dx-wm[alarm],
                  r-weights: rule-weights,
                  weights: weights-list,
                  o-res: ordering-results,
            binds( unit-type, unit-index,phase, heaviest-alarm)


            when similar(:phase, 'ordering)
                 and (unit-type = cac)
                 and (heaviest-alarm = 1007)
                 and [ some a: alarm in elements(:a-wm) ]
                          ( (a.handbook-nr = 1007)
                            and (a.accused-unit-type = unit-type
                            and (a.extra-info[1]  = 83) )

            : recommend(r-weights, weights, o-res, "cac1007", 1007, prom-d)
            end }
```

In clear English, the rule says that:

- *if we are in the ordering phase and the unit under diagnosis is cac and the heaviest alarm in the unit is 1007 and there is an alarm 1007 which points to a problem in the current unit being diagnosed with extra information = 83 then try to recommend a test sequence where prom-d would be tested first.*

The *id* field indicates the name of the rule and is there primarily for debugging purposes. For instance, the inference engine can be made to print the id of each rule it fires and it can be done very simply with one statement. An OPS like programming language would require a statement to be inserted in the action part of each rule.

The *rule-def* field contains the rule definition itself. As can be seen, the rule definition starts with a rule header containing formal arguments defining the type of objects that the rule can be related to. Any XE object is a valid argument. The *binds* construct establishes which variables are visible within the rules. This is very useful since, as is the case in this application, we may need a rule to consider several different types of objects of which there is only one instance. In a standard production system, they would have to be stored in the common working memory and therefore increase the search time in pattern matching. In the above rule, the current *unit-type, unit-index, phase* and *heaviest-alarm* are all single instance objects that are matched efficiently. The variables *r-weights, weights* and *o-res* are all used by the *recommend* procedure in the action-part of the rule to ascertain the validity of the conclusion and form an *ordering-results* object.

The rule uses a working memory *dx-wm* which is also passed as a parameter.

Working memory is implemented here as a parameterized abstract data type. The parameter is the type of the element that needs to be manipulated in the working memory. The definition of *dx-wm* in XE looks as follows:

dx-wm = **datatype** *[T:* **type** *]* **is** *create, insert, elements*

The procedures *create* and *insert* respectively create a working memory where objects of type *T* can be manipulated and insert new elements in it. The type *T* is supplied at instantiation time and our rule header for instance it is the type *alarm.* When Xeda is started up, it requests all potentially interesting alarm data and stores it in this working memory.

5.3.4 Inference engine

The inference engine is an XE procedure that takes into account in its match-execute cycle, the fact that it may need to be interrupted during its operation.

All DX 200 processes are monitored by the supervision process. This monitoring consists of sending a check message and awaiting a reply to it. If the reply message is received within a specified period of time, the process monitored is considered sound. Because the inferencing could last more than this specified response time, it is necessary to have the incoming message checking performed after each match-execute cycle. Furthermore, the DX operator may wish to stop the diagnostics process which consequently means that Xeda needs to be interrupted. State-of-the-art production system languages, like OPS83, do not offer this facility and could therefore not be used in an environment like ours. With the exception handling mechanism of XE, this problem can be handled neatly. The inference engine procedure checks for incoming messages after each cycle and signals an interrupt exception to the caller when an appropriate message is received. It is up to the caller then to handle the exception in the manner wished.

So the header of the inference engine procedure looks as follows:

dx-dia-inference = **proc** *(alarm-total-weights: weight-list,*
a-wm: dx-wm[alarm],
o-res: ordering-results,
rule-w: rule-weights,
rule-s: rule-set)
signals*(interrupt(mesg-type))*

Note that there has been no need for any conflict resolution. It suffices to fire any elligible rule.

There was another type of knowledge used in XEDA which was not suitable to represent in rules, but rather as an abstract data type. This knowledge concerned the diagnostics tests, the time they took to be carried out, and the order relations imposed by the architecture.

Other abstractions used in XEDA include iterators, procedures and data types. Abstract data types are used extensively to hide the representation of objects and permit easy maintenance of the program. For instance, the *dx-alarm* data type defines how an alarm object looks for XEDA and how it can be manipulated. Other parts of the XEDA program do not have any knowledge of how the alarm data is acquired (e.g., through message communication or reading a file). Should the structure of alarms change in the DX 200, the place where to carry out the changes can be quicly located in the XEDA software and modifications will be limited to this data type.

5.3.5 External interface

The interface from XE to our conventional programming language, PL/M, is supported partly by the XE language and partly by user-defined abstract data types. This feature of XE is indispensable because of the need to use operating system calls and gain access to the DX 200 data base through PL/M procedure calls. We have found that abstract data types provide a convenient mechanism to improve this foreign language interface. For instance, a *vector* abstract data type was developed to pass array structures between PL/M and XE. This was needed because of the different ways the two languages handle arrays. Arrays have no fixed size in XE; and PL/M manipulates structures only with pointers. The header of the *vector* datatype looks as follows:

vector = **datatype** *[T:* **type** *, N: int]* **is** *create, dump, undump*
inherits *fetch, store, elements*

The *vector* abstract data type has a fixed size *N*.
The *create* operation creates a vector of size *N*.
The *dump* operation linearizes a vector object for PL/M use and *undump* transforms a PL/M byte back to the PL/M object. The *inherits* construct allows the *vector* datatype to inherit from the representation type (in this case array) the *fetch, store* and *elements* operations. A similar approach has been adopted in [Erman et al. 1988] where abstract data types are used to describe data tokens passed between heterogeneous modules.

Communication from other processes to XEDA, is achieved through the message-based inter-process communication mechanism of the DX 200 operating system. Actually, this means calling the *send-message* and *receive-message* PL/M routines.

5.3.6 Time and space efficiency

The code generated by XE is efficient. Xeda totals 6000 lines of XE source code, of which 40% are rules (100 rules). The Intel 8086 executable code generated was 85K bytes. We found XE to generate 350 bytes on average for each additional rule. This figure is 150 bytes less than what OPS83 would generate. In terms of speed, the execution is sufficiently fast for XEDA to be a useful application. The requirement is that the inference time should not get near the time saved by restructuring the order of the test suite. For instance, assume a given situation where the fault could be located in 13 minutes using the initial test suite and after XEDA's restructuring, in 3 minutes. If XEDA's inferencing time is in the order of 10 minutes, then it is just a waste of processor time to utilize the services of this application. We have found that the openess of the rule mechanism in XE offers substantial room for optimization.

Furthermore, the XE programming environment was helpful. A simple simulator for DX 200 system calls was built very quickly in XE itself on the Lisp machine. The strong compile-time checking minimized error debugging on the target machine.

6 Conclusions

In summary, we believe that knowledge-based techniques and methodologies should and can be used in engineering software for embedded systems. Knowledge-based programming has not yet had an impact on mainstream programming. It is appropriate and important at this point to integrate knowledge-based programming into the mainstream of software development. There are at least two important justifications for this claim:

- knowledge-based techniques allow the software to take a step away from the computer and toward the problem domain. This will help to reduce the distance between the specification of a problem solution and its implementation. Development time and errors will be cut down because the complexity of the mapping process is reduced. Knowledge-based techniques bring visibility to the resulting software with a major positive effect on maintenance.

- Domain knowledge is a major factor in software development productivity, quality and costs. Knowledge engineering puts domain knowledge in an important position and therefore brings to traditional software engineering a useful attitude and set of methodologies, like knowledge acquisition.

The development language and environment play an important role in carrying through the application of these techniques to product level. Based on experience using the XE programming language and environment, we find the integration of the rule-based paradigm with abstract data types and abstraction-based programmingin general a viable and workingapproach for large embedded systems. The main arguments are:

- Generally speaking, it is a very promising approach for reconciling procedural and knowledge programming which is a recurrent major problem in today's software systems.

- Efficient implementation of the rule-based paradigm is possible.

- The inference engine can be fully tuned and optimized for each application.

- Efficient cohabitation and interfacing with abstract data types is possible.

- An identical data representation formalism can be used both by the procedural and rule-based parts of a program.

- The same language can be used to develop both conventional and knowledge-based software.

- Abstract data types support modularity and reusability.
- Because of the high reliability requirements of embedded systems and the poor testing environments usually associated with them, it pays to have as much verification as possible performed in the development environment. An ADT-based language with strong type checking performed at compile-time helps considerably in improving reliability in a cost-effective manner.
- By providing support for programming-in-the-large in an integrated programming environment, major software product development is made possible.

There are two areas where we suggest some further research:

- It should be investigated how easily the facilities in the XE language permit the construction of domain oriented knowledge-level abstractions. For instance how can a generic task [Chandrasekaran 1986] be implemented in XE?
- More practical experiments are needed to establish how well XE measures up in hard real-time tasks. Furthermore, a knowledge-based system developed wit XE should be monitored for a long period of practical and the ease of maintainability of the system studied from the language point of view.

7 References

1. Abott, R., J. Knowledge Abstraction. *Commun. ACM* 30, 8 (Aug. 1987), 664-671.

2. Arkko, J., Kuusela, J., Nuutila E., Tamminen, M., and Hirvisalo, V. The ExBed project - some experiences. In *Proceedings of the Finnish AI Symposium* (Helsinki,Aug. 15-18), Helsinki, 1988.

3. Arkko, J., Hirvisalo, V., Kuusela, J., Nuutila, E. and Tamminen, M. Some Experiences with Rules in Procedural Programming Languages. In *Proceedings of International Computer Science Conference'88*, Hong Kong, 1988.

4. Arkko, J., Hirvisalo, V., Kuusela, J., Nuutila, E. and Tamminen, M. *XE Reference Manual*. Report HTKK-TKO-A26, Helsinki University of Technology, Dept. of Computer Science, 1989.

5. Arkko, J., Hirvisalo, V., Kuusela, J., Nuutila, E. and Tamminen, M. *Programming in XE*. Report HTKK-TKO-C33, Helsinki University of Technology, Dept. of Computer Science, 1989.

6. Arkko, J., Hirvisalo, V., Kuusela, J., Nuutila, E. and Tamminen, M. *Notes on XE Run Time Characteristics and Code Generation*. Report HTKK-TKO-C37, Helsinki University of Technology, Dept. of Computer Science, 1989.

7. Bratko, I. *Prolog Programming for Artificial Intelligence.* Addison-Wesley, 1986.

8. Bouteldja, M. Dxpert, An Expert System for Complex Alarm Interpretation in the DX 200 Digital Switching System. In *Proceedings of the International Expert Systems Conference*, (London, June 2-4), Learned Information, Oxford, 1987.

9. Bouteldja, M. Engineering Embedded Knowledge-Based Software for a Digital Switching System, In *Proceedings of SETSS'89*, IEE, London, 1989.

10. Bouteldja, M., Arkko, J., Kuusela, J., Nuutila, E. and Tamminen, M. Building an Embedded Knowledge-Based Application using Abstractions, In *Proceedings of the 2nd Scandinavian Conference on Artificial Intelligence*,Tampere, Finland, 1989.

11. Bouteldja, M., and Lepistö P. On Some Problems in Applying Artificial Intelligence in Large Industrial Systems. In *Scandinavian Conference on Artificial Intelligence*, T. Danielsen, Ed., IOS, Amsterdam, 1988.

12. Brownston, L., Farrell, R., Kant, L. and Martin N. *Programming Expert Systems in OPS5.* Addison Wesley, 1985.

13. *CCITT, Functional Specification and Description Language SDL* . CCITT red book, Vol. VI, Recommendations Z.100-Z.104, Geneve, 1984.

14. Chandrasekaran, B. Generic tasks in knowledge-based reasoning: high-level building blocks for expert system design. *IEEE Expert* 1, 3, 1986, 23-30.

15. Curtis, B., Krasner, H., and Iscoe N. A field study of the software design process for large systems. *Commun. ACM* 31, 11 (Nov. 1988), 1268-1287.

16. Däcker, B., Elshiewy, N., Hedeland, P., Welin, C-W., and Williams, M. Experiments with programming languages and techniques for telecommunications applications. In *Proceedings of SETSS'85* , IEE, London, 1985.

17. Erman, L., D., Lark, Jxx ., S., and Hayes-Roth, F. ABE: An Environment for Engineering Intelligent Systems *IEEE Trans. Soft. Eng.* 14, 12 (Dec. 1988), 1758-1770.

18. Fagan, L. *Ventilator manager: a program to provide on line consultative advice in the intensive care unit.* PhD. thesis, Computer Science Department, Stanford University, 1980.

19. Fickas, S., and Novick, D. Control in Rule-Based Systems: Relaxing Restrictive Assumptions. In *Proceedings of the 5th International Conference on Expert Systems and Their Applications* , Avignon, 1985.

20. Fickas, S., Downing, K., Novick, D. and Robinson, B. The Specification, Design and Implementation of Large Knowledge-Based Systems. In *Northcon85, Conference Record* , Portland, OR, USA, Oct. 1985.

21. Forgy C.L. Rete: a fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence* 19, 1, 1982, 17-37.

22. Forgy C.L. *The OPS83 Report* . Report CMU-CS-84-133, Carnegie-Mellon University, Dept. of Computer Science, 1984.

23. Ganapathi, M., and Mendal, G. Issues in Ada Compiler Technology. *IEEE Computer* 22, 2, (Feb. 1989), 52-62.

24. Hayes-Roth, F. Rule-based systems. *Commun. ACM* 28, 9, (Sept. 1985).

25. Inder R. The state of the ART. *AIRING* , AIAI, University of Edinburgh, (Feb. 1988).

26. IntelliCorp, *KEE Software Development System User's Manual.* Version 3.0, IntelliCorp, 1985.

27. Karonis, F., C., and King, R., E. Multi-Level Expert Systems for Industrial Control. In *Proceedings of the International Expert Systems Conference* , (London, June 2-4), Learned Information, Oxford, 1987.

28. Kuusela, J., Arkko, J., Hirvisalo, V., Nuutila, E., and Tamminen, M. *XE Programming Environment User's Guide.* Report-HTKK-TKO-C35, Helsinki University of Technology, Dept. of Computer Science, 1989.

29. Laffey, T., J., Cox, P., A., Schmidt, J., L., Kao, S., M., and Read, J., Y. Real-Time Knowledge-Based Systems. *AI magazine* 9, 1, 1988, 27-45.

30. Liskov, B, and Guttag J. *Abstraction and Specification in Program Development* . MIT press, Mass., 1986.

31. Liskov, B., Snyder, A., Atkinson, R., and Schaffert, C. Abstraction Mechanisms in CLU. *Commun. ACM* 20, 8 (Aug. 1977), 564-576.

32. Liskov, B., and Zilles, S. Programming with abstract data types. *SIGPLAN Not.* , (Apr. 1974), 50-59.

33. Luqi, Berzins, V., and Yeh, R.T. A Prototyping Language for Real-Time Software. *IEEE Trans. Software Eng.* 14, 10 (Oct. 1988), 1409-1423.

34. McDermot, J., and Forgy, C. Production System Conflict Resolution Strategies. In *Pattern-directed Inference Systems* , Academic Press, 1978.

35. McDermot, J., Newell, A., and Moore, J. The Efficiency of Certain Production System Implementations. In *Pattern-directed Inference Systems* , Academic Press, 1978.

36. Meyer, B. *Object-oriented Software Construction* . Prentice Hall International, London, 1988.

37. Milliken, K.R., et al. *YES/L1: A Language for Implementing Real Time Expert Systems* . IBM Thomas J. Watson Research Center, Yorktown Heights, New York, 1985.

38. Miranker, P. TREAT: A Better Match Algorithm for AI Production Systems. In *Proceedings of AAAI* ,Seattle, Washington,July 1987.

39. Nuutila, E., Kuusela, J., Tamminen, M., Veilahti, J., Arkko, J. and Bouteldja, M. XC - a Language for Embedded Rule Based Systems. *SIGPLAN Not.* 22, 9 (Sep. 1987), 23-32.

40. Odette, L., L., and Dress, W., B. Engineering intelligence into real-time applications. *Expert Systems* 4, 4, (Nov. 1987), 228-239.

41. Rayment, P., J., and Thomas, S. Advanced Embedded Expert System Techniques. In *Proceedings of the International Expert Systems Conference*, (London, June 2-4), Learned Information, Oxford, 1987.

42. Riley, G., D. Timing Tests of Expert System Building Tools. *Report FM7-86-51*, L. B. Johnson Space Center, NASA, 1986.

43. Robertson, J. Stimulus - a Base Language for Real Time Expert Systems. In *Proceedings of the Conference on AI and Advanced Computer Technology*, Wiesbaden, Sept. 1985.

44. Sachs, P., A., Paterson, A., M., and Turner, M., H., M. Escort - an expert system for complex operations in real-time. *Expert Systems* 3, 1, (1986), 22-29.

45. Sakkinen, M. On the darker side of C++. In *Proceedings of the European Conference on Object-Oriented Programming*, Oslo, 1988.

46. Sauers, R., and Walsh, R. On the Requirements of Future Expert Systems. In *Proceedings of IJCAI*, Karlsruhe, West Germany, Aug. 1983.

47. Shaw, M. Abstraction techniques in modern programming languages. *IEEE Softw.*, (Oct. 1984), 10-26.

48. Stolfo, S., J., and Miranker, D., P. DADO: A Parallel Processor for Expert Systems. In *Proceedings of International Conference on Parallel Processing.* IEEE, Computer Society Press, 1984.

49. Stroustrup, B. *The C++ Programming Language*. Addison-Wesley, 1986.

50. Tanenbaum, A., S., Van Staveren, H., Keizer, E., G., and Stevenson, J., W. A Practical Toolkit for Making Portable Compilers. *Commun. ACM* 26, 9, (Sep. 1983), 654-660.

51. Theuretzbacher, N. Expert Systems Technology for Safety-Critical Real-Time Systems. *Electrical Communication* 60, 2, ITT, 1986.

52. Vesonder, G., Stolfo, S., Zielinski, J., Miller, F., and Copp, D. ACE: An Expert System for Telephone Cable Maintenance. In *Proceedings of IJCAI*, Karlsruhe, West-Germany, Aug. 1983.

53. Walker, J.H., Moon, D.A., Weinreb, D.L., and McMahon, M. The Symbolics Genera Programming Environment. *IEEE Software*, (Nov. 1987), 363-45.

54. Wright, M., L., Green, M. W., Fiegl, G., and Cross, P., F. An expert system for real-time control. *IEEE Software* 3, 2, (March 1986), 16-24.

55. Ȧrzen, K-E., Expert systems for process control. In *Proceedings of the 1st International Conference on Applications of AI in Engineering Practice* Southampton, UK, April 1986.