1 Introduction

This report is an overview of both the ExBed project and its main result, the XE Language Environment.

The ExBed project was established to determine to which extent it is feasible to apply “AI techniques” within embedded systems, and what such techniques should look like. The resulting environment was to consist of two distinct parts: the development environment, where embedded applications are programmed and tested, and the actual embedded target environment. At runtime the resources of a “standard” embedded microcomputer provide a limiting factor for functionality, but there are no similar limitations on the development environment. The expert system paradigms supported by ExBed were to be integrated with a procedural host language.

The ExBed project has developed a succession of three programming languages, XC, XD, and XE, each of which integrate production rules with a procedural programming language. For XE the project has also developed an integrated programming environment. This report is an overview that helps the reader approach the documentation related to these languages, the XE programming environment, and the applications developed in the project.

2 Problem Formulation

2.1 Embedded systems and Artificial Intelligence

The ExBed project was established to study expert system techniques applicable to embedded systems.

The term “embedded system” relates to a programmable machine (computer) contained within a larger piece of equipment (a product), which is not itself a computer. See (Finster 1986) and (Foulger 1982) for a discussion of the special requirements of embedded systems.

The motivation for applying AI to an embedded system comes from a desire to add to the product features that increase its value. For instance, one may want to make it recover better from failures - or just make it more showy. However, when features are added a system grows. At the same time it is important to speed up the development cycle of products to remain competitive. It is hoped that knowledge engineering will shorten the development cycle and also provide models for dealing with embedded programs that are radically larger than present ones. Bouteldja (1989) gives one justification to the above hope by showing how knowledge-based techniques help bridge the gap between the specification of a problem, in terms of the application domain, and the implementation of a solution for it.

There is a concensus that expert systems have been most successful when applied to specialized narrow domains. Embedded systems have narrow domains, almost by definition. Thus embedded systems form a potentially very important application area for knowledge engineering. At the same time there are exceptionally many technical problems in such applications, and when the ExBed project was started only few embedded expert systems were known to exist worldwide (Fagan, 1980, Griesmer et al. 1984, Sachs et al. 1986, Wright 1986). See also (Laffey et al. 1988).

Figure 1: Block diagram of the Telenokia DX 200 digital switch.

2.2 ExBed Applications

The ExBed project was to build general purpose software to support the development of embedded (expert) systems. At the same time, in order to direct the effort and evaluate its results, the project was to develop prototype applications related to the products of the supporting companies (KONE Corporation and Nokia Corporation, specifically Telenokia). KONE chose the control of a group of elevators as its prototype application, while Nokia chose the maintenance of Telenokia's digital switching system. In both cases the platform for the expert system was to be a control computer based on ordinary 16 bit processors. Figure 1 is the block diagram of the DX 220 digital switch, which is based on a decentralized architecture implemented with several Intel 8086 processors.

2.3 Characteristics of Embedded Expert Systems

Embedded expert systems differ from traditional, consulting, ones in many respects. For instance, the requirement for reliability is much more stringent; there is no user whose knowledge could be relied on or who would reinitiate an unsuccessful reasoning process. Also, systems to be embedded into products must nowadays operate on "standard" microprocessors, often under tight resource constraints. It is therefore very important to be able to effectively control the use of resources. ¿From a knowledge engineering viewpoint embedded systems are difficult due to the fact that they must operate continuously. Thus they should be able to take into consideration in real time the continuous changes in the state of the surrounding environment.

Traditionally expert systems have been built either by using "AI programming languages" or closed expert system shells. However, for embedded expert systems it is very important to be able to establish a direct contact with other system parts implemented in "ordinary" procedural programming languages.

2.4 The Software Engineering Approach

Embedded systems are notoriously laborious to build and test. The difficulties are often due to the fact that there is no acceptable utility software available for the target computer. The ExBed project started out on the premise that the development and target environments should be separate (Fig. 2). The former could be provided with wide functionality irrespective of the resource limitations of the actual embedded application. Similarly, in the development phase one is allowed to use much machine resources to verify the correctness of an application and to optimize its runtime use of resources.

The installation of an expert system from the development environment into the target environment is very similar to the task of compiling a source code module into an object module. Thus it became a central objective of the ExBed project to develop such programming constructs and related compiling techniques that facilitate the verification of the correctness of an expert system and the optimization of its performance. As a matter of fact, the project came to follow more closely the research paradigms of software engineering than those of artificial intelligence.

2.5 Project Objectives and Outline

Rule based programming is a well established expert system technique and also a rather natural way of defining the operational behaviour of an embedded control system. The forward chaining inference mechanism has been found to be the most

Figure 2: Development environment and target environment.

appropriate for embedded systems (Fagan 1980, Griesmer et al. 1984) The support of rule based programming was taken as the minimal objective for the software to be developed in the project. The testing and maintenance of rule based programs is known to be difficult. Therefore, it was decided right at the start of the project to use abstract data types, or data abstraction (Liskov and Guttag 1986) as the basis of the programming methodology to be developed.

The ExBed project can be split into two phases. During the first year prototypes were developed for both applications mentioned above. At the same time the rule based programming language XC was implemented by using precompiler techniques to extend an existing procedural language, C++ (Stroustrup 1986). In this approach the development environment was an ordinary Unix environment. After the first year, the direction of the software effort was changed to be much more ambitious; it was decided to design a self-contained general-purpose programming language XE, together with an integrated programming environment for it. At this point the development environment came to reside on a Lisp workstation.

Helsinki University of Technology has had the main responsibility for the ExBed project and developed the general purpose software. Telenokia has been independently responsible for the design and implementation of its own application prototypes. The ones for KONE have been built in cooperation between the company and the university. The input of the supporting companies into the specification and testing of the general purpose software has been important. Especially worth mentioning is the effort Telenokia has invested into XE.

3 Project Implementation and Results

3.1 Approach to Rule Based Programming

In the following the progress and results of the project will be described in chronological order. The arguments behind the choices we made will also be presented. At first, however, a few words about the chosen approach to rule based programming (Nuutila et al. 1987).

At the start of the project the researchers had experience (Kuusela and Nuutila 1986) on implementing traditional "OPS5-like" (Forgy 1979) production system languages. Typically (before OPS83) such languages did not support compile-time type checking and and programs written in them were difficult to debug. The implementations of these languages typically use the RETE pattern-matching algorithm (Forgy 1982), which is based on building a so-called discrimination network to describe the contents of a rule set and keep track of the state of the working memory. When changes are made to the working memory, the state of the discrimination network must also be updated. Using this redundant state information the checking of the condition parts of rules can then be performed efficiently as part of an inference cycle. The program implementing the RETE algorithm forms a kind of a black box: it operates automatically for an arbitrary rule set but its operation is difficult to control.

The project decided to abandon the OPS5-like approach. First of all the RETE algorithm slows down all updates related to variables used in the condition parts of rules. Also it is difficult to control the size of the network and the CPU time requirements, which makes the approach problematic for real-time applications. Further, the algorithm leads to duplicate maintenance of data structures in both the rule based and the procedural parts of a program. Because typical rule modules of embedded systems were expected to be small the project considered the possibility of application specific tailoring more important than the the black box strategy of the RETE algorithm. The objective was to obtain data structures and inference mechanisms that could be controlled and optimized by the programmer and that would adhere closely to the concepts of a procedural host language.

The programming language was to provide as many compile time checks as possible. Data abstraction was chosen as the dominant approach because it supports strong type checking based on module interface specifications. It also provides a uniform approach to the modular construction and maintenance of verifiable programs.

Support for parallel processes and real time performance is important for embedded systems. However, these concepts were not included in the ExBed programming languages because they were mandated to be taken care by the proprietary operating system of each supporting company.

3.2 XC

The objective of the design of the XC language was to add a forward chaining rule abstraction to an expressive procedural programming language. The host language was to support data abstraction techniques and the writing of efficient code. Further, XC was to be implementable during the first project year. On these grounds the host language was chosen to be C++, an object oriented extension of C.

Among object oriented languages the distinguishing characteristic of C++ is that it is static: type checking is performed at compile time, and there is no need for an extended run time system. The implementation of C++ was based on a precompiler that produces ordinary C code. This meant rather good machine independence.

The XC language is described in (Nuutila et al. 1987). It retains all of the expressive power of C++ and adds to it the concepts of a rule set and a working memory. A rule set consists of rules (Fig. 3) that refer to working memories and other program variables. XC is modular in the sense that there can be an unlimited amount of rule sets and working memories. For each rule set the XC compiler builds an independent executive program (an "inference engine").

```
/* Assume type Object has function size(). Then rule r1
   removes all Objects bigger than MAX_SIZE from local_wm. */
rule r1(Object* obj in local\_wm) {
  when (obj->size() > MAX_SIZE) {
    local_wm.retract(obj);
  }
}
```

Figure 3: An XC rule.

XC was implemented as a preprocessor which generates C++ code. Figure 4 illustrates the stages of an XC compilation: 30 KB of XC source code (120 rules) gives rise to 450 KB of C code after two precompilation steps. A single test cycle would take 15 minutes, of which the major part is due to the C-compiler. However, it was also shown in (Nuutila et al. 1987) that in many cases the running time of a rule based program implemented in XC was better than that of the same task implemented in OPS83 (Forgy 1984), the most efficient representative of the "OPS family". This was typically true for small problems and for problems where the discrimination network grew excessively due to a large amount of possible problem solutions.

Figure 4: The phases of XC compilation.

3.3 DXPERT

The DX 200 is a family of digital switches whose architecture is based on a network of high-performance microcomputer units communicating through a fast message bus (Fig. 1).

DXPERT is an expert system for DX 200 alarm interpretation. When initiated by a DX 200 technician, it will perform a knowledge-based analysis of the alarm situation to identify the possible roots of the problems. As a result the DX operator is provided with a diagnosis of the cause of the alarms and directions how to recover from the fault. Figure 5 illustrates the role of alarms within the overall maintenance system of the DX 200.

As Telenokia's first application prototype the part of DXPERT related to I/O failures (about 100 rules) was implemented in the OPS83 language (Forgy 1984). The resulting system was made to operate in an MS-DOS based microcomputer used as the DX operator's console. See (Bouteldja 1987) for more information on the system.

3.4 DPR

The first application prototype of KONE was a system for the demand pattern recognition of a elevator group. Figure 6 shows the problem of demand pattern recognition as a simplified diagram. The X and Y axes measure the intensities of up and down traffic. Each of the regions of the diagram is amenable to a different group control strategy. It is the task of the demand pattern recognition expert (DPR) to infer what is the true type of the traffic at each moment.

Figure 5: DX 200 maintenance system.

Figure 6: Demand pattern recognition.

DPR was implemented in XC. The work was mainly performed in a Unix environment, and the operational result was later interfaced to KONE's group control system. The system makes it possible to follow the traffic pattern floor by floor. Statistical monitoring and inference methods are used, together with rule based reasoning.

To make it possible to develop DPR at the university also a simulator of elevator systems was needed. Therefore the ExLift (Experimental Lift Group Simulator) system was constructed by using C++ and the OOPS system for Smalltalk-like object-oriented programming in C++ (Gorlen 1986).

3.5 Filex

The Filex file system expert monitors the use of file space and provides the users with services for managing their files and for freeing disk space.

Filex is run every night. The rules of the system first try to infer whether a file is worth keeping on disk by trying to determine its nature and then considering this information in light of the amount of free space in the system. For each file, Filex reaches a recommendation on what should be done with it. This recommendation is presented to the owner of the file when he or she logs in the next day. In extreme cases, Filex can remove files without authorization from the owner of the files. Filex was implemented as a student project using the XC language. See (Arkko et al. 1988b) for more information.

3.6 Metaprogramming Environment

Compiler building tools were needed in the construction of the XE compiler and programming environment. These tools do not depend on the target language (XE), and so they are referred to as the metaprogramming environment. Part of the tools - those utilized in the front end of the compiler - were developed within the ExBed project in Common Lisp. The rest were provided by ACK (Amterdam Compiler Kit) (Tanenbaum et al. 1983), whose main role in the project was to provide code generation techniques and a framework for retargetability. The most important metaprogramming tools developed in the project were:

1. ROCC (Robust Compiler Compiler): an LALR parser generator patterned after YACC and designed to be used in "Lisp style" (Arkko et al. 1989/C38) Functions related to error recovery have been added to YACC.
2. AST (Abstract Syntax Tool): a small language that helps in manipulating abstract syntax trees (Nuutila et al. 1989/C39). AST makes it possible to

write code that needs very little maintenance when the syntax of a language is changed or the processing of a local part of the syntax is modified.

3. DG (Descriptor Generator Tools): a small language for defining objects based on multiple inheritance (Nuutila et al. 1989/C39). DG is used in the definition and maintenance of descriptor objects related to symbol tables.

Beside ACK the project also experimented with Twig (Aho et al. 1986), a code generator generator developed att Bell Labs. The results obtained in the generation of Intel 8086 code were encouraging.

3.7 The XE Language

Based on the experiences with XC, developed in the first phase of the project, data abstraction was deemed to be a suitable basis for implementing embedded expert systems. The main enhancement wished for rule based programming was yet increased openness. That is, the programmer was to be given complete freedom in the design of the data structures and algorithms related to both rule sets and working memories. We also preferred parameterization to inheritance as a type generalization mechanism in many cases.

The programming environment of XC (Fig. 4) was, however, a big disappointment to people used to a Lisp environment. The test cycle was very long as the there were many compilation phases, a whole rule set had to be compiled as a unit, and there was no dynamic linking. Further there was no debugging support for C++. Also, programs written in XC would be very difficult to analyze automatically, because XC contains as a subset C++ - and therefore also C. All this led to a reconsideration of the language strategy.

Thus it was decided to base the second phase of the project on a more ambitious approach than precompilation. It was taken as an objective to design a general purpose programming language XE, which would support both rule based programming and parameterized data abstractions. Because the CLU language satisfied the latter objective it was taken as a model for the design of XE. However, XE is not an extension of CLU; instead it is an integral whole specified from its own premises. When deciding about XE it was considered especially important that the XE compiler would be part of an integrated programming environment that would support programming in XE in the same way as a Lisp environment supports programming in Lisp (Walker et al. 1987).

The top level constructs of XE are constants and parameterizable procedures, iterators, rules, and datatypes. The iteration abstraction makes it possible to traverse and manipulate objects in a collection without disclosing the representation of the collection: the iterator provides the calling module with each of the objects, one at

Figure 7: Definition of a data type in XE.

a time (Fig. 7). In analogy to CLU

XE supports exception handling. The semantics and implementation of rules are intimately related to those of iterators. However, lexical closures and nested modules are concepts not existing in CLU but necessary for the XE rule abstraction.

The XE language is defined in (Arkko et al. 1988/A26), and (Arkko et al. 1989/C33) is a tutorial to its use. Further (Hirvisalo et al. 1989/C34) contains a large set of general purpose abstractions that can be built upon in XE programming, and also used as examples of XE programming style. Here we cannot describe XE in detail but Fig. 7 should provide the reader with a feeling for the nature of the language.

Compared to CLU an important practical feature of XE is that modules written in XE can communicate with modules written in other languages. Further, XE has been designed so that it has been possible to write all of the basic libraries in XE itself, instead of assembler. This leads to much easier retargetability than that of CLU. The objective of the many other extensions to CLU has mainly been to make programming more pleasant. See (Hirvisalo et al. 1989) for a more detailed comparison of XE and CLU.

Figure 8: An XE rule.

Figure 8 shows a simple example of an XE rule. It has one argument, and the type of the rule is defined by the `yields` construction. The condition part (left hand side) of the rule (the `when` construction consists of nested calls to iterators, together with conditions to be applied to the objects yielded by the iterators. In exact analogy to an iterator, the rule yields instantiations defined by its action part (right hand side), to be processed by the calling module (the "inference engine"). (The example rule yields closures that output two integers when executed.)

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 be implemented in XE itself, typically as follows:

1. Rule set: A data structure containing rules and providing an iterator over these rules.
2. Working memory: A data structure of XE objects providing one or more iterators to be used in the left-hand sides. In the above example the sequence of integers [1 .. 3] could be considered a working memory over which i and j iterate.
3. Conflict resolution: Code that compares two rule instantiations and retains the "better" one.
4. Rule firing: An invocation of an action embedded in a rule instantiation.
5. Inference engine: Code that iterates over rules in a rule set and their instantiations.

See (Arkko et al. 1989c) for some more overview of the XE rule mechanism.

3.8 XE Programming Environment

The XE compiler is not an independent program but its parts have been integrated into a uniform programming environment, together with an editor and version management. The programming environment has been implemented in Common Lisp, using XE metaprogramming tools, and uses a Symbolics workstation as its basis. The environment consists of 1.1 MB (29000 lines) of Lisp code and 186 KB (6500 lines) XE code. The use of the XE metaprogramming tools has reduced the size of the environment by many thousand lines (Nuutila et al. 1989/C39). See (Kuusela et al. 1989/C35) for a user's guide, and (Arkko et al. 1989/C40) for a description of the internal structure and implementation of the programming environment. An overview and evaluation is further provided in (Kuusela et al. 1989a). Figure 9 shows the structure of the XE compiler as if it were an independent program.

The XE programming environment provides the following functions related to the management of programs and the programming process:

1. The editing of program code: The editor and compiler are intimately coupled, so that when the compiler notices syntactic or semantic errors the editor cursor will be positioned at the corresponding locations. Also, the system recognizes which parts of a program have been changed during an editor session.

Figure 9: Structure of the XE compiler.

2. Version management and configuration control: The program base can be organized as a (nested) collection of *systems*. For each session the user can be provided with either an explicitly named version of a system, or one inferred by automatic rules (e.g., the newest version). Version management also supports browsing the program base in various fashions.

3. Smart recompilation: When a system has been changed the compiler automatically determines which modules must be recompiled in order to obtain a consistent compiled version of the system. The approach used is described in (Nuutila et al. 1989a).

4. Debugging and execution monitoring: program execution can be monitored at the source code level. The monitoring and testing of XE programs is based on an approach where the source code is compiled into Lisp.

Figure 10 shows a typical snapshot of a session with the XE programming environment.

Figure 10: A snapshot of an XE programming session.

3.9 XE Libraries, Code Generation and Run Time Environment

The basic library of XE contains definitions of the fundamental data types and operations of XE. Beside the basic library there exists a large utility library (Hirvisalo et al. 1989/C34). Both of these libraries are fully machine independent.

To be able to cross compile an XE program into the target environment, a code generator for the corresponding CPU is needed. Also, a CPU-specific run-time environment library, is needed.

The generation of code for target environments (the most important of which are based on the Intel 8086 processor at the present) has been built upon the tools of ACK (Tanenbaum et al. 1983). These tools have been modified to a certain extent in order to support special features of XE (such as garbage collection) and the memory models relevant to us. The ACK approach is based on a virtual machine called EM. EM code is generated from the abstract syntax tree on the Lisp workstation. CPU-specific code is then generated from EM.

Reliability of the generated code and rather easy retargetability are the main benefits resulting from the use of ACK. A drawback is that the compilation path is long and uses two different computers, since ACK does not run on Lisp workstations.

Independently of actual code generation, a rather large number of optimizations can be performed. These have mainly been implemented as source level transformations on the XE abstract syntax tree. The optimizations have had a decisive role in the good quality of the Intel code produced from XE. Its resource usage has usually been comparable (or in some cases even better) to that of code produced by the Microsoft C-compiler. For instance, fibonacci(27) took 10.2 seconds when compiled with the XE compiler, whereas it took 18.8 seconds in the same machine (PC-AT) compiled with the Microsoft C-compiler. Comparisons for rule-based programs were made against XD using Microsoft C-compiler. A rule based program for solving the problem of the Towers of Hanoi with 14 disks took 31.0 seconds when written in XE. The same program took 26.0 seconds when written in XD (Arkko 1989).

The XE run time environment consists mainly of program code implementing garbage collection and exception handling. Further, a description of all data types used by a program must be available at run time. The minimal size of an XE program without even the I/O libraries on an Intel 8086 is only 6.5 KB (1.8 KB in Microsoft C). With the I/O libraries the minimal size is 10.4 KB (8.4 KB in Microsoft C). These numbers include garbage collection and other required run time operations for XE.

See (Arkko 1989) for details of XE code generation, optimization and run time environment. A related overview is provided by (Arkko et al. 1989/C37).

3.10 XEDA

XEDA (XE Diagnostic Advisor) is a diagnostic knowledge based system embedded into the DX 200 digital switch. It runs as a process under a real-time operating system. Communication with other processes involved in the maintenace function of the switch, is achieved trough asynchronous message passing.

The role of XEDA is to determine, through a knowledge-based analysis of the alarm history, the optimal sequencing of diagnostics tests. The optimal order is one in which the test program identifying the fault is run as early as possible. The need for this intelligent diagnostic arises from the fact that some tests can be very lengthy (30-60 minutes).

True to its name XEDA has been completely programmed in XE, except for one PL/M module. Depending on the version (there are two versions, one for each type of switch) XEDA contains 100 - 200 rules, and altogether 5000 - 7000 lines of XE code. It has been a major challenge to make XEDA fit into the mere 128 KB allotted to it in the DX 200 memory, and this challenge has given rise to many of the optimizations in the XE compiler.

See (Bouteldja et al. 1989) for discussion of the use of XE in XEDA.

3.11 XD

During the ExBed project it was realized that, if only the rule abstraction was required from XE, it could be rather simply implemented on top of, for instance, the C or C++ languages. The implementation is based on precompilation, and three very small run time assembler routines. the preprocessor was built by modifying the GNU C-compiler. Because the parser is largely shared by GNU C and C++ it was possible to extend both languages with XE rules at the same cost.

The resulting language was named XD. It can be used in cases where it is necessary to port a rule based program designed for XE into a C environment. This may be necessary either for efficiency reasons (XD does not carry the overhead of garbage collection or built-in checks) or in the case that XE will not be taken into the permanent programming language repertory of the supporting companies.

The performance of XD is comparable or better than that of XC, and at the same time it provides all the controllability and optimization features of XE. However, due to the good code generator and optimizer of XE, code produced by XD is not significantly more efficient than that obtained from XE. See (Rintanen et al. 1989) for a description of XD and its implementation. In (Arkko et al. 1989c) and (Arkko 1989) there are more details on the comparisons referred to above.

3.12 Diagnostic Expert of a Control System

As the second phase application prototype of KONE Corporation, a diagnostic expert related to the elevator group control system is under development. Part of the functionality of the expert will be embedded in the microprocessors actually controlling the system; the other part will reside in a loosely coupled MS-DOS machine. This system will be written in XD.

4 Evaluation

The effort and results of the ExBed project have shifted focus in relation to the intentions at the start of the project. There has been a move from research and results in knowledge engineering towards those in core areas of software engineering. At the same time the results have become much more general. That is, whereas the objective of the project was to develop embedded expert systems and tools for building them, the result is a general purpose programming language and an application independent programming environment for it.

One can say that, within the domain of software tools, the objectives of the project were changed to qualitatively more demanding and more extensive than planned. The results could also have been more widely useful than planned because they are not tied just to expert system applications. However, from the point of view of the supporting companies, there are also serious problems involved in the use and maintenance of a new programming language. One of the problems is the special purpose machine (or at least very large machine resources for running Lisp) which the XE programming environment requires at the present.

The software tools of the project have, for the most part, been implemented in Common Lisp. Also the functionality of the XE programming environment is based on that of a Lisp environment (Walker et al. 1987). It would not have been otherwise possible to implement as much functionality with the given resources. At the same time one must admit that the single user Lisp workstation has also been a very serious bottleneck during the implementation effort. Also, the XE programming environment is closer to a prototype than a final product in many respects, and the supporting companies do not regard a Lisp workstation as a suitable basis for a programming environment for production use. If the Lisp environment had not been used as a starting point, the project would probably have chosen less ambitious objectives, but possibly obtained relatively more complete results with respect to them.

It is not yet possible to give a final evaluation of data abstraction as a foundation for rule based programming. On the basis of the experiences obtained the approach seems valid, at least as far as the correctness of the resulting systems is concerned

(Bouteldja et al. 1989). But it is still unclear to which extent strong type checking increase the difficulty of exploratory programming and the difficulty of designing general purpose utility modules for knowledge-based programming.

What one misses most in the general purpose results of the project are expression facilities and functions related specifically to embedded systems. The lack of these is partly due to the fact that application systems had to use the expression facilities of company specific proprietary operating systems. Extensive enough application experiences are also missing. This is partly due to the fact that, as the ambition level of the project was raised, the results, specifically the XE programming environment, were also delayed with respect to the plans.

The objectives related to application prototypes have been mostly fulfilled:

1. The DPR prototype was experimentally embedded in the elevator group control system, and KONE is developing DPR further to form part of its product.
2. DXPERT was one of the aatractions of Telenokia's stand at a major Telecommunications exhibition. There have been requests from customers for systems with the functionality of DXPERT and Telenokia plans to include similar functions in the operator consoles of its new products.
3. Filex is in continuous use on certain computers at the university.
4. XEDA has been embedded into a test version of the DX 200 system.

The Telenokia applications are based on knowledge engineering techniques, specifically rule based programming, as was planned. However, DPR uses more varied techniques.

Among the general purpose results of the project the XC and XD languages and compilers are easy to put to use anywhere. The XE language and programming environment require a much larger intellectual investment. The metaprogramming environment would have applications outside of the project, but no effort has been invested into making it a product.

Evaluation of the ExBed results continues within the WISE project (Well Informed Software Development Environment) as part of the FINSOFT research program.

ACKNOWLEDGEMENTS

DPR was designed and implemented by Jukka Veilahti together with Riitta Partanen-Jokela and Jukka Mattila from KONE Corporation; Marja-Liisa Siikonen (KONE) is continuing the work. Nassim Bouteldja worked for Telenokia during whole project, and Timo Lehtimäki (Telenokia) had a central role in the XEDA effort. Jari Heikkinen, Kimmo Talvitie and Jari Tomminen were part of the team designing and implementing Filex. XD was implemented by Jussi Rintanen. Beside application-oriented work the named people from Telenokia and KONE have also closely monitored the general purpose part of the project.

We would like to acknowledge the unflinching backing of the ExBed support group and its courage in letting us aim at such ambitious objectives. In this respect specific mention is due to Tuomo Huhtanen (Telenokia), Pekka Kemppainen (KONE), Pertti Lounamaa (Nokia Research) and Heikki Saikkonen.

REFERENCES

1. A.V. Aho, M. Ganapathi , and S.W.K. Tjiang, Code Generation Using Tree Matching and Dynamic Programming, ATT Bell Labs, 1986.

2. Jari Arkko. Asiantuntijajärjestelmäkielen optimoitu toteutus Intel-ympäristöön (Optimized Implementation of an Expert System Language). Master's Thesis (in Finnish), 1989.

3. Arkko, J., Kuusela, J., Nuutila E., Tamminen, M., Hirvisalo, V. The ExBed project - some experiences. In Proceedings of the Finnish AI Symposium (Helsinki, Aug. 15-18), Ed. Limes Ry, Helsinki, pp. 496-505, 1988a.

4. Arkko, J., Kuusela, J., Nuutila E., Tamminen, M., Filex: A File System Expert Written in XC. In Proceedings of the Finnish AI Symposium (Helsinki, Aug. 15-18), Ed. Limes Ry, Helsinki, pp. 544-552, 1988b.

5. Arkko, J., Hirvisalo, V., Kuusela, J., Nuutila, E. and Tamminen, M., Some Experiences with Rules in Procedural Programming Languages, Proc. International Computer Science Conference '88, Hong Kong, Dec. 19-21 1988c, pp. 712-718.

6. Arkko, J., Hirvisalo, V., Kuusela, J., Nuutila, E. and Tamminen, M., Rule-Based Expression Mechanisms for Procedural Languages, To appear in Computational Intelligence 5(1989)4.

7. Jari Arkko, Vesa Hirvisalo, Juha Kuusela, Esko Nuutila, and Markku Tamminen, XE Reference Manual. Report TKK-TKO-A26, Helsinki University of Technology, Laboratory of Information Processing Science, 1989.

8. Jari Arkko, Vesa Hirvisalo, Juha Kuusela, Esko Nuutila, and Markku Tamminen, Programming in XE. Report TKK-TKO-C33, Helsinki University of Technology, Laboratory of Information Processing Science, 1989.

9. Jari Arkko, Vesa Hirvisalo, Juha Kuusela, Esko Nuutila, and Markku Tamminen, Notes on XE Run Time Characteristics and Code Generation. Report TKK-TKO-C37, Helsinki University of Technology, Laboratory of Information Processing Science, 1989.

10. Jari Arkko, Vesa Hirvisalo, Juha Kuusela, Esko Nuutila, and Markku Tamminen, ROCC Programmer's Guide. Report TKK-TKO-C38, Helsinki University of Technology, Laboratory of Information Processing Science, 1989.

11. Jari Arkko, Vesa Hirvisalo, Juha Kuusela, Esko Nuutila, and Markku Tamminen, XE Programming Environment Implementation Notes. Report TKK-TKO-C40, Helsinki University of Technology, Laboratory of Information Processing Science, 1989.

12. 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.

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

14. Bouteldja, M., Arkko, J., Hirvisalo, V., Kuusela, J., Nuutila, E. and Tamminen, M., Building an Embedded Knowledge-Based Application using Abstractions, To appear in Proceedings of SCAI'89.

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

16. Brownston, L., Farrell, R and Kant, E., Programming Expert systems in OPS5. Addison-Wesley, 1985.

17. 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.

18. Finster, K. Developing Embedded Systems using Board-Level Products. In Proceeedings of BUSCON 86 West Conference, Jan. 1986, 146-154.

19. Forgy, C.L., On the Efficient Implementation of Production Systems, Carnagie-Mellon University, Department of Computer Science, 1979.

20. Forgy, C.L., OPS5 user's manual. Report CMU-CS-81-135, Carnegie-Mellon University, Department of Computer Science, 1981.

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

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

23. Foulger, R., J., Programming Embedded Microprocessors. NCC Publications, London, 1982.

24. Gorlen, K., Object-Oriented Program Support. OOPS Version 1 Reference Manual. National Institutes of Health, 1986.

25. Griesmer, J.H., Hong, S.J., Karnaugh, M., Schor, M.I., Ennis, R.L., Klein, D.A., Milliken, K.R. and VanWoerkom, H.M., YES/MVS: A continuous real time expert system. Proc. AAAI National Conference 1984, pp. 130-136.

26. Hirvisalo, V., Kuusela, J., Arkko, J., Nuutila, E. and Tamminen, M., XE Design Rationale: CLU Revisited. Submitted for publication 1989.

27. Vesa Hirvisalo, Jari Arkko, Juha Kuusela, Esko Nuutila, and Markku Tamminen, XE Utility Library Manual, Library Version 1.0. Report TKK-TKO-C34, Helsinki University of Technology, Laboratory of Information Processing Science, 1989.

28. Kuusela, J., Nuutila, E., Hybrid AI Development Tools, STeP-86 Symposium Papers, vol 2, Espoo, August 19-22, Otapaino, 1986, pp 149-156.

29. Kuusela, J., Arkko, J., Hirvisalo, V., Nuutila, E. and Tamminen, M., Overview of the XE Programming Environment. Submitted for publication 1989a.

30. Juha Kuusela, Jari Arkko, Vesa Hirvisalo, Esko Nuutila, and Markku Tamminen, XE Programming Environment User's Guide. Report TKK-TKO-C35, Helsinki University of Technology, Laboratory of Information Processing Science, 1989.

31. Laffey, T.J., Cox, P.A., Schmidt, J.L., Kao, S.M., and Read, J.Y., Real-Time Knowledge-Based Systems. AI Magazine, Vol. 9, No. 1, pp. 27-45, 1988.

32. Liskov, B., Snyder, A., Atkinson, R., Schaffert, C., Abstraction Mechanisms in CLU, Comm. ACM, vol. 20, no. 8, 1977, pp. 564-572.

33. Liskov B., et al., CLU Reference Manual. Springer-Verlag, 1981, 190p.

34. Liskov, B., Guttag, J., Abstraction and Specification in Program Development, The MIT Press, Cambridge, Massachusetts, 1986

35. Milliken, K.R., Cruise, A.V., Ennis, R.L., Hellerstein, J.L., Masullo, M.J., Rosenbloom, M., Van Woerkom, H.M., YES/L1: A Language for Implementing Real-Time Expert Systems, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, 1985.

36. Moore R. L., Adding real-time expert system capabilities to large distributed control systems, Control Engineering, (April), (1985).

37. Nuutila, E., Kuusela, J., Tamminen, M., Veilahti, J., Arkko, J. and Bouteldja, N., XC – A Language for Embedded Rule Based Systems. SIGPLAN Notices, Vol. 22, No. 9, pp. 23-32, 1987.

38. Nuutila, E., Hirvisalo, V., Arkko, J., Kuusela, J. and Tamminen, M., Smart Recompilation in the XE Compiler. Submitted for publication 1989a.

39. Esko Nuutila, Jari Arkko, Vesa Hirvisalo, Juha Kuusela, and Markku Tamminen, AST and DG Tools Programmer's Guide. Report TKK-TKO-C39, Helsinki University of Technology, Laboratory of Information Processing Science, 1989.

40. Jussi Rintanen, Markku Tamminen, Jari Arkko, Vesa Hirvisalo, Juha Kuusela and Esko Nuutila, XD Manual. Report TKK-TKO-C36, Helsinki University of Technology, Laboratory of Information Processing Science, 1989.

41. Robertson, J., 'STIMULUS' - a base language for real time expert systems. Proc. Conf. on AI and advanced Computer Technology, Wiesbaden, 24-26 Sept., 1985, 15 pp.

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

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

44. 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.

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

46. Wright M. L.,Green, M. W., Fiegl, G., Cross, P. F., An expert system for real-time control, IEEE Software, Vol. 3, No. 2, (March) pp. 16-24, (1986).

47. Ȧrzen K-E., Expert systems for process control. Proc. 1st Int. Conf. on Applications of AI in Enginering Practice, Southampton, UK, April 15-18, 1986.

Contents

1 Introduction **1**

2 Problem Formulation **1**

2.1 Embedded systems and Artificial Intelligence 1

2.2 ExBed Applications 2

2.3 Characteristics of Embedded Expert Systems 3

2.4 The Software Engineering Approach 3

2.5 Project Objectives and Outline 3

3 Project Implementation and Results **5**

3.1 Approach to Rule Based Programming 5

3.2 XC 6

3.3 DXPERT 7

3.4 DPR 7

3.5 Filex 9

3.6 Metaprogramming Environment 9

3.7 The XE Language 10

3.8 XE Programming Environment 13

3.9 XE Libraries, Code Generation and Run Time Environment 16

3.10 XEDA 17

3.11 XD 17

3.12 Diagnostic Expert of a Control System 18

4 Evaluation **18**