# Building an Embedded Knowledge-based Application Using Abstractions

M. Bouteldja*
J. Arkko
V. Hirvisalo
J. Kuusela
E. Nuutila
M. Tamminen

Helsinki University of Technology,
Department of Computer Science,
Espoo, Finland.

**Abstract**

Knowledge-based application development for embedded systems requires an approach different from that used in other environments. This is due to the nature of embedded systems which amongst other characteristics, have strong reliability requirements and aim to optimize the use of underlying hardware resources. Any approach that satisfies these constraints must be supported by the development tools. In this paper, we show how the XE programming environment supports embedded knowledge-based software development. The key idea is the integration of the rule-based paradigm with abstract data types in the same programming language environment. We illustrate the ideas through XEDA, an embedded knowledge-based application developed at Nokia Telecommunications to support the diagnostics function of the DX 200 digital switching system.

## BACKGROUND

Embedded systems are claimed to be a promising industrial AI application. The ExBed project (expert system framework for embedded applications) was established to determine how knowledge based techniques can be applied within embedded systems running on a microcomputer. Fagan [4] was the first to attempt the construction of a continuously operating expert system. Based on his experiences the forward chaining inference strategy is usually assumed most suitable for these systems.

According to the charter of ExBed the constructor of embedded systems was to be provided with a rule based expression mechanism and the above references lead to the choice of the forward chaining inference strategy. It was decided to approach the problem, not by using or designing an expert system shell, but by embedding the required facilities in a general purpose programming language.

*also at Nokia Telecommunications, Finland.

The first language designed in the project, XC, was obtained by extending an existing language, C++ [10]. Experiences with XC [9] lead to the design and implementation of a self-contained language, XE [2], together with an integrated programming environment for it. Here we report on the features of the XE programming environment and its suitability for embedded knowledge-based software development. We illustrate our ideas through an embedded application developed at Nokia Telecommunications.

## THE XE LANGUAGE

XE is a general purpose programming language whose design owes very much to CLU [7,8]. 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 a time and each object is consumed by the *for* construct in turn. Iterators are central to the rule concept, as embodied in XE.

### 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
```

As an example, here follows a simple rule that yields three instantiations. The first contains integer 1 and an action to output "2" and "1"; the next one integer 2 and an action to output "3" and "1"; finally integer 1 and an action to output "3" and "2".

```
simple = rule(output) yields(int, proctype())
      when [for i: int in [1 .. 3]]
                  [for j: int in [1 .. 3]]
                        i > j
      :
      yield(i - j,
      proc() binds(i, j)
                  putl(:output, i)
                  putl(:output, j)
      end)
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 orders wanted 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 [3] 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.

The following example is a minimal illustration of some of these concepts. Assume a parameterized datatype *stack* with a procedure *push* and an iterator *elements*. First we define type constants *rtype* as a *simple* type of rules and *rstack* as the type of a stack of *simple* rules. Then we create *rs*, a stack of rules of the same type *simple* above. In the following example we push *simple* and other rules of the same type into the stack of rules. After that, we iterate over all rules and their instantiations (inference engine), in order to find the instantiation with the highest value (conflict resolution) of the integer attribute yielded by the rule. Finally we execute (rule firing) the procedure (action part of rule) corresponding to the highest integer value found.

```
rtype = ruletype() yields(int, proctype())
rstack = stack[rtype]
rs: rstack := rstack $ new()
imax: int := 0
pbest: proctype() := proc() end
push(:rs, simple)
...
...
for r: rtype in elements(:rs) do
    for i:int, p:proctype() in r() do
        if i > imax then imax, pbest := i, p end
    end
end
pbest()
```

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.

- **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. Straightforward code generation from a program based on abstract data types does not usually produce an efficient result. 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, iterators were especially important. Inline coding, tail recursion elimination, jump propagation and redundant code elimination greatly enhanced the quality of the code produced from XE rules. The optimization efforts produced quite good results: if array bounds check are left away the code produced by the XE compiler has been comparable to that produced by the Microsoft C compiler.

## THE XE PROGRAMMING ENVIRONMENT

The ExBed project has built an integrated programming environment for the XE language. Although languages with run-time type checking were not deemed suitable for reliability and efficiency reasons, from the point-of-view of responsiveness, LISP environments were the ones to emulate.

The XE programming environment contains facilities for editing, version control, program code inspection, smart recompilation, execution and monitoring (based on generated LISP code) [6]. The concept of files is nowhere visible to the programmer, who operates only on XE concepts. By using compiler development tools like ACK [11], code generation for different target environments is facilitated.

The development and control of large systems is facilitated by the following concepts provided by the environment.

A *system* is the topmost hierarchical component manipulated by the programmer in the XE programming environment. For example, a program or a set of programs that are used to perform a task can be a system. Another example is a graphic package that contains type definitions of various graphical entities and routines that manipulate them.

There can be many *generations* of the same system. The generations correspond to the historical development of the system and they form a tree. A generation associates with every module name that it uses a single implementation (some revision of that module). These associations are changed either manually or automatically as the system evolves. A generation has several attributes that describe its status, e.g., whether it is a released or an experimental generation.

Often it is necessary to use modules of another system as components in a system. To achieve this it is possible to *export* and *import* modules. A module is imported to the system if it is referred to in it, but not defined in it. Only modules, which are exported can be imported and only from systems which are *included* into the system, which refers to them. If a module could be imported from several systems *a name conflict* is said to arise and the user must resolve which binding he wants to use. The restrictions specified above as the import mechanism greatly reduce the amount of name conflicts.

At the present the program development environment operates on a LISP workstation under the Genera operating system [12]. The main target environment has been Intel 8086. All of the XE built-in libraries and most of the Intel run-time environment has been programmed in XE itself. Expression facilities making this possible are an important part of the design of the XE language.

## 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 usually severe in large embedded systems. 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.

## XEDA

### Function

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.

### External communication

XEDA runs as a process in the Operation and Maintenance Computer and communicates with other processes developed with conventional programming techniques (see fig. 1). 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 (see fig. 2). 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.

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.

### Main logic

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 5 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 suscriber 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.

## Structure

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. The header of the definition of this 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*]

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. Figure 3 shows an example of a test ordering rule with comments.

Working memory is a parameterized abstract data type. The parameter is the type of the element that needs to be manipulated in the working memory. Figure 4 describes the definition of a working memory.

The inference engine is a procedure that includes in its cycle, a phase where it checks for any incoming messages. 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. State-of-the-art production system languages, like OPS83 [5] do not offer this facility and could therefore not be used in an environment like ours.

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. Some of these abstraction are shown in figure 5. 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.

## 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 the XE programming environment extremely 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.

## CONCLUSION

We have presented the XE programming language and its features supporting embedded knowledge-based software development. We believe that its novelty relies primarily in the integration of the rule-based programming paradigm with ADTs and abstraction-based program construction. Through a substantial embedded knowledge-based application development, called XEDA, experience was gained in XE programming. The XE features that eased the implementation of XEDA and that we consequently find desirable in a programming environment for embedded knowledge-based software development are:

- Access to XE variables from within rules without loss of efficiency.
- Both procedural and rule-based part are written in the same high-level language.
- Complete type checking is performed at compile-time
- Modularity and genericity are supported.
- The inference engine can be fully tuned and optimized for the application.
- The programming environment offers support for programming-in-the-large.

## REFERENCES


1. 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), Ed. Limes Ry, Helsinki, 1988.

2. 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.

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

4. 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.

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

6. Kuusela, J. *XE Programming Environment User's Guide.* Report-HTKK-TKO-C, Helsinki University of Technology, Dept. of Computer Science, 1989.

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

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

9. 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.

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

11. 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.

12. Walker, J.H., Moon, D.A., Weinreb, D.L., and McMahon, M. The Symbolics Genera Programming Environment. *IEEE Software* , (Nov. 87), 36-45.