Rule-Based Expression Mechanisms for Procedural Languages

Jari Arkko, Vesa Hirvisalo, Juha Kuusela
Esko Nuutila, Markku Tamminen

Helsinki University of Technology
Laboratory of Information Processing Science
02150 Espoo 15

Abstract

We report on experiences on adding a rule based expression mechanism to an existing procedural programming language (C++) and on designing and implementing a self-contained language – and its integrated programming environment – supporting similar but more general capabilities. Both languages, XC and XE, are based on abstract data types and XE is a close relative of CLU. Its programming environment – implemented on a LISP workstation – contains facilities for editing and composing programs, browsing a program data base, debugging, version management, and cross-compilation to microprocessors, including the Intel 8086.

Key words: Programming languages, Embedded systems, Expert systems

Subject categories: Languages/tools

1. 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 expert system techniques can be applied within embedded systems running on a microcomputer.

Fagan (1980) 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.

The state-of-the-art of the OPS production system family, based on the RETE algorithm (Forgy 1982), is represented by OPS83 (Forgy 1984). YES/L1 (Milliken et al. 1985) is a RETE-based extension of PL/1. Hexscon (Wright et al. 1986) and Escort (Sachs 1986) are more rigid languages than what we aim for, while PICON (Moore et al. 1985) is large and specialized to continuous processes. Stimulus (Robertson 1985) has many objectives in common with us. See (Laffey 1988) for a recent survey on real-time knowledge-based 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. In this way, ExBed became more a programming language project than one on expert systems.

The first language we report on, XC, was obtained by extending an existing language, C++ (Stroustrup 1986). Experiences with XC lead us to implementing a self-contained language (XE) together with an integrated programming environment for it.

We first motivate the design decisions of XC and XE, explain why the famous RETE (Forgy 1982) match algorithm is not used, and show what data representation and program structuring facilities we think are necessary. Next we describe experiences with XC and motivate the introduction of XE. Then we describe XE, relating it to CLU (Liskov et al. 1977), and also discuss how a similar rule abstraction can be added to C. Finally we describe the results of some experiments and our experiences with implementing XE.

2. Data Abstraction as a Basis

The RETE algorithm (Forgy 1982) is a technique for matching a set of left-hand sides (conditions) of rules against a set of data (a working memory) in order to determine which rules are eligible to fire, i.e., execute the actions on their right-hand side. The algorithm derives its efficiency from two assumed features of production systems, namely: (1) structural similarity of the left hand sides, and (2) temporal redundancy in the working memory. This means that only a small number of working memory elements change from inference cycle to another. Under these assumptions the algorithm is rather fast because it never explicitly iterates over the working memory or the set of production rules. Instead, partial matches of the left-hand sides are stored in a discrimination network while complete matches are stored in a conflict set.

Modifications to the working memory are rather expensive in RETE, because they must be propagated through the discrimination network. Typically some hundreds of modifications per second can be made to the working memory – according to our experiences from OPS83 and from implementing the RETE algorithm ourselves (Kuusela and Nuutila 1986). Thus, if the temporal redundancy assumption is false, the algorithm runs very slowly.

Unfortunately in many real-time applications large amounts of data must be processed in a very short time. In such a case some procedural language will be used in the time critical parts of the program while production systems can be used in higher level, non-time-critical parts. But the problem is that the RETE algorithm can use only data that is propagated through its network. Thus, the mere existence of the network can slow down the procedural parts of the program. Further, the amount of dynamic memory needed by a discrimination network can vary much during one execution and is difficult to predict.

To overcome these problems we have studied the use of "simple-minded" production system algorithms. An example of such an algorithm is one that scans a set of rules. At every rule it generates every possible combination of working memory elements and checks whether the left-hand side of the rule is satisfied.

The performance of this kind of an algorithm is terrible, if used carelessly. An important observation is, however, that such algorithms are not as inefficient as is usually claimed. The main reason is that there are many user controllable optimizations that can help avoid the combinatorical explosion. Often no working memories are needed as the rules refer directly to program variables. Further, our process control system experts ascertained that typical applications will be highly modular with rather small rule sets.

Production systems usually have some predefined and rather restricted way representing for modeling a problem and representing its data in working memory elements. The classes of attribute - value pairs in OPS5 (Forgy 1982) are one such example. It is rather obvious that no single mechanism is suitable for all applications. Try, for example, to represent bit vectors with OPS5 classes.

Furthermore, direct access to the data is disallowed, because it could cause inconsistency in the complicated saved state of the algorithm. Sometimes this problem is solved by using multiple representations of data (Kuusela and Nuutila 1986; Wright et al. 1986). This, however, requires conversions between the different representations.

We think that an identical data representation formalism should be used by both the procedural and rule based parts of the system. It should be as general as possible and allow problem-dependent optimizations. Therefore we have taken the concept of abstract data types as a basis.

Another common problem of rule based systems is lack of modularity. Indeed, these systems generally consist of one linear set of rules that communicate via a global database. In XC and XE the programmer is free to tailor and modularize both the knowledge representations and the "inference engines" according to the needs of his problem and its solution strategy. By using abstract data types to define several object types, working memories, and sets of rules, the user can decompose the problem into manageable parts.

The high verification requirements of embedded systems are one more reason for the choice of data abstraction as the foundation of XC and XE.

3. From XC to the XE Programming Environment

The ExBed project did not start out with the plan of designing and implementing a programming language from scratch. The first programming language of the project, XC, was an extension of the object-oriented language C++ (Stroustrup 1986), implemented as a preprocessor.

The basic built-in concepts of XC were working memories and rule sets. A given working memory contained references to user-defined objects of some specific types. A rule set was a collection of rules for which the compiler produced a separate executive function. See (Nuutila et al. 1987) for examples on XC.

By adding rules to C++ the new constructs could benefit from all its powerful features, such as object classes and inheritance between them, at no extra programming cost. Similarly, without any investment in code-generation or optimization techniques we had a multi-target compiler producing efficient code.

We were satisfied with the expressive power of XC. Further, in comparative experiments (Nuutila et al. 1987) XC was always more memory efficient and in many cases also more time efficient than OPS83. The experiments also supported the importance of problem specific algorithms and data structures in rule based embedded systems.

Even though XC did not give us as much freedom as we would have wanted for choosing algorithms and data structures for working memories and rule sets this could have been remedied to a certain extent.

Despite the above, we decided to discard the XC-approach due to the following problems, partly related to the use of C++ as the host language:

1. C++ is a large language without formal specification and it carries along the misfeatures of C. It would be difficult to provide tools to analyze XC programs. Also, C++ does not support data abstraction as well as desired.

2. Program development with XC, which is built on top of C++, is tedious because of the many phases of compilation – from XC to C++, from C++ to C, from C to assembler, from assembler to machine code, and finally a great big linking stage. A typical test cycle for a 120 rule program took 6 minutes of CPU time and 15 elapsed minutes. This is mostly attributable to the compilation of a 400 KB C-language file (generated from an original 30 KB); the translation from XC to C++ took only 23 seconds.

3. There was no high level debugger for C++. To build a debugger for C++ is to build a compiler for it.

Thus, beside a good programming language, also a good programming environment is required. From the point-of-view of responsiveness, LISP environments are the ones to emulate. However, from the points-of-view of reliability and efficiency, languages with run-time type checking were not deemed suitable. Instead, the project has built an integrated programming and cross-compilation environment for its own programming language XE:

1. A program development environment containing facilities for editing, version control, program code inspection, smart recompilation, execution monitoring, etc. The concept of files is nowhere visible to the programmer, who operates only with XE concepts. There is no separate compiler, in the ordinary sense; instead, the components implementing the compiler front end and code generation (to LISP) are integrated into the programming environment. Anyway, we will use the term compiler to denote these components.

2. Compiler back ends to generate code for target environments. For code generation we have modified ACK (Tanenbaum et al. 1983).

3. Runtime systems (including controllable garbage collection) for the target environments.

At the present the program development environment operates on a LISP workstation. The main target environment has been Intel 8086.

The requirement that the XE compiler be retargetable and that an XE program can be compiled into efficient code for a 16 bit microprocessor has been a special challenge when implementing XE. The support of large programs in 16 bit address spaces has greatly affected the design of the run-time system, and also, to some extent, the code generators.

5. The XE Language

XE is a general purpose programming language that owes very much to CLU (Liskov et al. 1977), which has been used as a starting point in its design.

XE allows parameterized (generic) data abstractions as the main means of expressive power and reuse and its mechanisms for defining procedures, iterators and data types will be familiar to anybody knowing CLU. 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.

A data type defines a set of objects and a set of primitive operations to create, examine and manipulate them. One goal of XE is to make user defined types powerful by treating built-in and user defined types as uniformly as possible. A data type is implemented by a *type module* that describes a *concrete representation* for objects of that type and routines to implement its operations.

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.

It is not possible to describe XE here in detail; instead we shall demonstrate its flavor by an example of a simple user-defined type generator. Comments point to some characteristics of XE. Note that an operation of a datatype is denoted by giving both the type name and the operation name separated by a dollar sign. (A colon in front of an argument is syntactic sugar for the dollar-notation.)

The example contains definitions and invocations of *iterators*. Iterators are an abstraction that allows performing a block of code (in a **for**-loop) for each element in an abstract collection of elements – without disclosing the representation of the collection. Iterators are central to the rule concept, as embodied in XE.

```
% Stack parameterized with maximal size and type
% of elements; for illustration an iterator and
% a restriction have been included.
mystack = datatype[maxn: int, t: type ]
   is new, pop, push, elements
     where t has default: proctype() returns(t)
   % restriction on element type
   rep = record { % representation; not visible outside
                  n: int;
                  a: array[t]
   }
   new := proc() returns(cvt) % return abstract type
      return({n: 0, a: array[t]$fill(maxn, t$default())})
      % record constructor invoking an array constructor
      % restriction on type parameter used here
   end % new
   pop = proc(s: cvt) returns(t) signals(empty)
      % s viewed as the concrete representation
      if s.n = 0 then signal(empty) end % raise exception
      s.n := s.n - 1
      return(s.a[n])
   end % pop
   push = proc(s: cvt, e: t) signals(full)
      if s.n = maxn then signal(full) end
      s.a[n] := e
      s.n := s.n + 1
   end % push
   elements = iter(s: cvt) yields(t) % an iterator
      for e: t in elements(:s.a)
   % invocation of an iterator example of use below
          yield(t)
      end
   end % elements
end % stack


% Example of instantiating and using stack abstraction
a_stack = stack[100, int use default = int$maxint]
as: a_stack := a_stack$new()
for i: int in [1 .. 10] do
   push(:as, i)
end
sum: int := 0
for i: int in elements(:as) do
   sum := sum + i
end
```

In XE, a rule is a special kind of iterator that computes a sequence of *instantiations*. An instantiation is a tuple of arbitrary XE objects. Usually it 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.

A rule is of the form (somewhat simplified):

rule ::= **rule** [parms] args [yields] [signals]
when condition :
body
end

Each rule has a fixed number of arguments and it can be parameterized just like a procedure or an iterator. The optional *yields* declaration in the header declares the number, order, and types of the components of instantiations. The rest of a rule consists of a *condition*, and a *body*, which is a statement. Instantiations of the rule are created by **yield** statements in the body.

A condition is either a clause or a sequence of nested **for**-iterators, qualified by a clause:

condition ::= `[` **for** [decl, ...] **in** invocation `]` condition
| clause

clause ::= clause **and** clause
| clause **or** clause
| **not** clause
| (clause)
| `[` **some** [decl, ...] **in** invocation `]` clause
| `[` **all** [decl, ...] **in** invocation `]` clause
| predicate

A predicate is an expression of type `bool`. Complicated clauses can be built by using operators **and**, **or**, and **not**. Clauses can be quantified with **all** (universal) and **some** (existential) quantifiers. A *quantification* consists of a quantifier, a list of variable declarations, and an iterator invocation.

Rule invocation is performed as follows. (Here it may be helpful to inspect the examples below.) The actual argument objects are assigned to the formal arguments of the rule. The outermost **for**-iterator (if any) is invoked. If it yields anything, the objects are assigned to the declared variables. Nested **for**-iterators are executed in the same way, except that when one of them terminates the closest surrounding iterator is resumed. If the innermost **for** iterator yields an item the clause contained in the condition is evaluated. If its value is **true** the expressions on the right-hand side are evaluated, the corresponding instantiation is yielded and the rule is temporarily suspended. (At this point the program can use the instantiation, e.g., fire it.) If the value is **false** the innermost **for** iterator is resumed and the clause is re-evaluated with the new variable bindings. When the rule is resumed the suspended iterator of the innermost **for** quantification is resumed.

The following rule yields its arguments (i, j) if $i > j$ and otherwise nothing.

```
trivial = rule(i, j: int) yields(int, int)
  when i > j:
  yield(i, j)
end
```

The following rule 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" and the final one integer 1 and an action to output "3" and "2"

```
simple = rule(output: stream) 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
```

Assume a parameterized datatype *stack* with an iterator *elements*. We define a stack of rules of the above type (after defining type constants *rtype* and *rstack*). 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, in order to find the instantiation with the highest value of the integer attribute yielded by the rule. Finally we execute the procedure corresponding to the highest priority 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()
```

Finally we present a rule with quantification. The following rule is taken from a file system expert program. It detects files residing in a directory called "temp" or "tmp", or in its subdirectory.

```
rule_0012 = rule(f: filename) yields(string)
  when [some dir: string in componets(:f)]
         dir = "temp" or
         dir = "tmp":
  yield("File is in temporary storage")
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 be implemented in XE itself, typically as follows:

Rule set: A data structure containing rules and providing an iterator over these rules. Above, *rs* is a rule set with iterator *elements*.

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 there is no actual working memory. However, in rule *simple* the sequence of integers [1 .. 3] could be considered a working memory over which i and j iterate.

Conflict resolution: Code that compares two rule instantiations and retains the "better" one. Above conflict resolution is performed by the comparison of i and *imax*.

Rule firing: An invocation of an action embedded in a rule instantiation. Above this is the invocation of *pbest* on the last line.

Inference engine: Code that iterates over rules in a rule set and their instantiations. Above this is the **for**-loop; only one inference cycle is shown.

Beside providing a rule abstraction XE extends the mechanisms of CLU in several other ways. Its parameterization and iteration facilities are more general. Also, to implement rules it has added nested routines and lexical closures (without, however, introducing Algol-like visibility rules). XE also makes the life of a programmer easier by removing arbitrary restrictions and by making the compiler do somewhat more work.

An extension important in practice is that programs written in XE can cooperate with programs and subroutines written in lower level languages, such as PL/M or C.

For retargetability all built-in libraries are written in XE. This has been made possible by, among other techniques, including facilities for defining type generators, such as `record`, parameterized by a list of fields. In CLU the corresponding libraries are written in assembly language, for efficiency. We aim to obtain efficiency by using a general program optimizer that enhances the efficiency of both libraries and user code.

6. XE rules in C

For certain real time control applications the XE runtime environment with garbage collection, exception handling, etc. may be unsuited (Note however, that the minimal system configuration including the above features is about 6.5 KB). In such systems the programmer might want to use, say, C or C++ instead. He can also use XC for rule based programming. However, XC rules give less possibilities for optimization than the rules of XE. Thus it is reasonable to suggest adding an XE-like rule mechanism to C and C++.

The XE rule mechanism is based on iterators and closures. Once a language contains these concepts, it can be extended for rule based programming in a straight-forward fashion. Iterators are a simplified kind of coroutines, and closures require allocating bound variables from the heap. We have been able to add both concepts to C using a very simple preprocessor and three less than 10-line assembler routines. Thus we are now able to use XE-like rules also in C, and we have called the corresponding language XD. Of course, programming in XD is more tedious and error prone than in XE; among other things the user has to take care of explicitly deallocating closures when they they become garbage.

7. Some Experiments

We have programmed some of the test problems of (Nuutila et al. 1987) also in

XD. Below we compare the results obtained with XD and XE to those of OPS83. The XD and XE programs were compiled with Microsoft C and run on an IBM PC/AT. OPS83 programs were tested using a Nokia ASC microcomputer (8MHz, AT-compatible), which is somewhat more powerful.

The Towers of Hanoi

A non-recursive solution of the problem of the towers of Hanoi was implemented by three rules. The test programs were made to be "identical" in the different languages. This is also true of the other benchmarks.

The programs moved 14 disks by 16384 rule firings. The XD version required 27 seconds, which means 606 rule firings/second. The XE version required 31 seconds, which means 528 firings/second. OPS83 required 292 seconds (56 firings/second). This indicates how a simple-minded strategy can be used to advantage in small problems.

The Attribute Problems

To automatically generate a large set of rules we used the following model. There are two collections of objects (*balls* and *boxes*, say) with *nobj* elements in both. All objects possess *nattr* different attributes (e.g., color, material and size), each with *nval* different possible values (e.g., red, green, blue for color). All objects are initially asserted into the working memory. The problem is to find a matching box for each ball; when a ball is matched it is removed from the working memory.

We defined a ball to match a box whenever any two attribute values of the ball are equal to the values of the corresponding box attributes. In XE a typical rule would look as follows:

```
SizeAndColor = rule(boxes, balls)
  when [for bx: box in Elements(boxes)]
         [for bl: ball in Elements(balls)]
           (bx.size = bl.size & bx.color = bl.color):
  begin
       PutInto(bl, bx)
       Retract(bl)
  end
end
```

With *nattr* attributes we have *nattr*(*nattr*-1)/2 such rules to express a match. In the tests *nval* was set to 32.

Only very small problems could be run within the 400 - 500 KB allotted to OPS83 while the XD and XE programs required less than 50 KB of memory. The largest OPS83 problems that could be run corresponded respectively to the following parameter values:

(A) $nobj = 170$, $nattr = 8$ (28 rules) and
(B) $nobj = 60$, $nattr = 15$ (105 rules)

Problem (A) requires 170 and (B) 60 rule firings. XD solved problem (A) in 9.0 seconds and XE in 7.7 seconds while OPS83 required 27 seconds. For problem (B) the corresponding figures were 4.5, 5.8 and 15 seconds.

The memory usage of OPS83 seemed to depend on the product of the number of attributes and the number of rules. This behavior is explained by the pattern matching and conflict resolution algorithms of OPS83. When the rules in the attribute problems are modified so that one equality test in each rule is replaced by a less than or equal test the memory requirements of OPS83 are further accentuated. In this case OPS83 could handle only 60 objects in the 28 rule case and 28 objects in the 105 rule case. Note, however, that the attribute test problem is a kind of a bad case for OPS83.

The memory requirements of XD and XE depend linearly on both the number of objects and the number of rules. This example demonstrates how simple-minded algorithms may work well even for moderately large problems when complicated conflict resolution strategies are not needed. It also shows how the RETE discrimination network may greatly increase the memory requirements of a rule-based program.

The attribute test problem cannot be claimed to be representative of rule-based programs. However, it allows us to illustrate a rather generally applicable method of optimizing the matching phase in a rule-based program written in XE or XD. The optimization, which is similar to techniques used to speed up database queries, consists of moving part of the clause of a rule into the iterators. For instance, after such an optimization the rule SizeAndColor would be transformed into:

```
SizeAndColorOpt = rule(boxes, balls)
  when [for bx: box in Elements(boxes)]
          [for bl: ball in ElementsBySize(balls, bx.size)]
            (bx.color = bl.color):
  begin
        PutInto(bl, bx)
        Retract(bl)
  end
end
```

The iterator *Elements* of balls has been modified to accept as an argument the size of the balls searched. Iterator ElementsBySize returns only balls of the correct size to be tested in the clause.

In the test problem we implemented each such iterator by maintaining an array, accessed by the values of a given attribute, with each array element pointing to a linked list of balls having the given value in that attribute. Such an index was formed for each attribute "on demand" The solution times of XD for test problems (A) and (B) were respectively 1.6 and 1.4 seconds with this optimization.

8. Experiences on XE

XE is being used for XEDA, a diagnostic advisor embedded in a digital telephone exchange. The strong typing of XE helped in shortening the test phase of XEDA by helping remove many errors on the basis of compiler diagnostics. At the present XEDA has been demonstrated to run in a telephone exchange but its rule base has not yet been fully validated. The major problem in XEDA was memory shortage in the telephone exchange. As we do not yet have large scale experiences on the use of XE we will convey some of our experiences on designing and implementing XE.

The problems related to XC, reported in Section 3, were largely due to the very static nature of our compilation strategy – partly dictated by the (lack of) tools available for C++. We must admit that we discarded the XC approach before even trying to solve the problems within it by writing tools for C++.

In XE an orthodox approach of abstract data types was chosen: we wanted as much support as possible for writing error free and maintainable programs. The resulting complete compile time type checking of XE is much appreciated by programmers for detecting program errors. However, it is not trivial to design types for rule instances and working memory elements such that they comply to type checking of this strength while retaining the original flavor of rule based programming.

We do not yet have enough experiences to form a final opinion on the strength of type checking best suited to our needs in rule based programming. However, if we designed a new language we would probably try to include more polymorphic features in it.

Choosing an existing programming language (CLU) as a basis for XE was an important decision. It has, of course, guided our way. However, at the same time it has made for a rather large language and limited our freedom of design, e.g., with respect to polymorphism.

Designing and implementing XE has been a far larger task than we believed when we decided to discard the XC approach (December 1986). From current literature one gets the impression that writing the front end of a compiler is a simple exercise, for which good tools supply most of the work. According to our experiences this is not true for a language of the complexity of XE, or even CLU. The complexity of

XE is due partly to our desire to be able to write all libraries in XE, and partly to the extra features (extended to the whole language in an orthogonal fashion) that were needed for the rule abstraction.

Most demanding in our work has, however, been the increase of compiler complexity resulting from the integration of the compiler in a programming environment. For instance, irrespective of the order in which the user chooses to manipulate his program parts he obtains automatic "smart" (Tichy 1986) recompilation and useful error messages. Smart control added a lot to the difficulty of the compiler front end. For instance, the semantic checker has required much more resources from us than designing and implementing ROCC (Robust compiler-compiler), a LISP-derivative of YACC (Arkko 1987), and a related set of tools for manipulating abstract syntax trees.

A positive surprise has been that syntax directed code generation required rather little work – after a couple of insights. Also, in the LISP environment, the method produces almost as good code as writing directly in LISP. However, the simple code generation strategy puts a lot of pressure on code optimizers before good enough code is obtained for the actual target environments. We have used the peephole optimization facilities of ACK, and added several of our own. The result can be seen by comparing the XE and XD run times given in section 7: the XE compiler generates good target code also for Intel 8086.

XE has more user programmability than most rule based languages. Compared to XC the user has much more freedom to tailor the data structures of working memories and rule sets according to problem needs. It is difficult to precisely compare the results of section 7 to those reported for XC (Nuutila et al. 1987) because the XC runs were performed on a different CPU (ATT 3B2). However, the performance of a program implemented in a similar fashion seems to be about similar for all three languages.

9. Conclusion

We have described an approach for including production rules in an "ordinary" procedural programming language. The approach is based on abstract data types in order to support program verification and to allow for maximal user programmability. It has been implemented in the XC and XE (and XD) programming languages. In this article we have tried to motivate our design decisions and report on our experiences with these languages.

ACKNOWLEDGEMENTS

This work has been performed at the Helsinki University of Technology, funded by the Technology Development Centre, Nokia Corporation and KONE Corporation. We would like to thank Heikki Saikkonen for his comments. Jussi Rintanen implemented a prototype of the XD translator and performed the related test runs. The OPS83 tests were performed by M.N. Bouteldja.

REFERENCES

Arkko, J., 1987. ROCC User's Guide. Helsinki University of Technology, Laboratory of Information Processing Science.

Fagan, L., 1980. Ventilator Manager: a Program to Provide On-Line Consultative Advice in the Intensive Care Unit. PhD thesis, Computer Science Department, Stanford University.

Forgy, C.L., 1982. Rete: a Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. Artificial Intelligence 19(1982)1, pp. 17-37.

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

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

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

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

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

Moore R. L., 1985. Adding Real-Time Expert System Capabilities to Large Distributed Control Systems, Control Engineering, April.

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

Robertson, J., 1985. 'STIMULUS' - a Base Language for Real Time Expert Systems. Proc. Conf. on AI and Advanced Computer Technology, Wiesbaden, 24-26 Sept., 15 pp.

Sachs, P.A., Paterson, A.M. and Turner, M.H.M., 1986. Escort - an Expert System for Complex Operations in Real Time. Expert Systems 3(1986)1, pp. 22-29.

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

A.S. Tanenbaum, H. van Staveren, E.G. Keizer, and J.W. Stevenson, 1983. A Practical Toolkit for Making Portable Compilers, CACM, vol. 26, no. 9, pp. 654-660.

Tichy, W.F., 1986. Smart Recompilation, ACM TOPLAS, vol. 8, no. 3, pp. 273-291.

Wright M. L.,Green, M. W., Fiegl, G., Cross, P. F., 1986. An Expert System for Real-Time Control, IEEE Software, Vol. 3, No. 2, (March) pp. 16-24.