Some Experiences with Rules in Procedural Languages

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

Helsinki University of Technology
Laboratory of Information Processing Science
02150 Espoo 15
mit at hutcs.hut.fi
Tel: 358-0-4512020 (4512679, 710317)

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 various microprocessors, including the Intel 8086.

Key words: Programming languages, Embedded systems, Expert systems

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 1985) is a RETE-based extension of PL/1. Hexscon (Wright 1986) and Escort (Sachs 1986) are more rigid languages than what we aim for, while PICON (Moore 1985) is large and specialized to continuous processes. Stimulus (Robertson 1985) has many objectives in common with us. See (Laffey et al. 1988) for a recent survey on 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, reported in more detail in the companion paper (Arkko et al.), and motivate the introduction of XE. Finally we describe XE, relating it to CLU (Liskov et al. 1977) and describe our experiences.

This work has been performed at the Helsinki University of Technology, funded by the Technology Development Centre, Nokia Corporation and KONE Corporation.

2. Data Abstraction as a Basis

The RETE algorithm 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 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. In some systems this problem is solved by using multiple representations of data (Kuusela and Nuutila 1986, Wright 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 (Arkko et al.) 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. XC did not give us enough freedom for choosing algorithms and data structures for working memories and rule sets but 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 is building a comprehensive integrated programming environment for its own programming language XE:

1. A compiler front-end (on a LISP workstation at the present).

2. A program development environment containing facilities for editing, program code inspection, smart recompilation, execution monitoring, etc. (LISP). The concept of files is nowhere visible to the programmer, who operates only with XE concepts.

3. Version control (LISP).

4. Compiler back ends to generate code for target environments, the main one of which has been Intel 8086. For code generation we have modified ACK (Tanenbaum et al. 1983) and also experimented with Twig (Aho et al. 1986), based on dynamic optimization.

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

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.)

```
% Stack parameterized with maximal size and type
% of elements; for illustration an iterator and
% a restriction have been included.
stack = datatype[max_size: 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) % returned value converted to abstract type
      return({n: 0, a: array[t]$fill(max_size, t$default())})
      % record constructor invoking an array construction operation
      % restriction on type parameter required here
   end % new
   pop = proc(s: cvt) returns(t) signals(empty)
      % s viewed as the concrete representation
      if n = 0 then signal(empty) end % raising an exception
      n := n - 1
      return(s.a[n]) % syntactic sugar for array[t]$fetch(get_a(:s), n)
   end % pop
   push = proc(s: cvt, e: t) signals(full)
      if n = max_size then signal(full) end
      s.a[n] := e % syntactic sugar for array[t]$store(get_a(:s), n, e)
      n := 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 stack abstraction and iterating over a stack
my_stack = stack[100, int use default = int$maxint]
ms: my_stack := my_stack$new()
for i in [1 .. 10] do
   push(:ms, i)
end
sum: int := 0
for i: int in elements(:ms) do
   sum := sum + i
end
```

Above there were examples of defining and using *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.

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() 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(:primary_output(), i)
            putl(:primary_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()
```

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 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.

The requirement that the XE compiler is retargetable and that an XE program can be compiled into efficient code for a 16 bit microprocessor, such as, the Intel 8086 has been a special challenge and has affected every facet of the compiler implementation.

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. 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.

6. Experiences on XE

In the following we want to convey some of our experiences in designing, implementing and using XE. XC has already been reported on, in Section 3. The problems related to XC were mostly due to the very static nature of our compilation strategy – and the C++ language. And, we must admit that we discarded the XC approach before even trying to solve the problems within it.

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 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", recompilation (more discriminating than ordinary "make"), 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 (Nuutila 1988).

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.

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. However, we are not yet able to generate as efficient code from XE as from XC when identical data structures are used. This will hopefully change when our XE optimizer, having, e.g., good facilities for inline coding, is operational.

7. 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 enhance program verification and to allow for maximal user programmability. It has been implemented in the XC and XE programming languages. In this article we have tried to motivate our design decisions and report on our experiences with these languages.

ACKNOWLEDGEMENTS

We appreciated very much the advice of Dennis Allison during the early stages of our compiler writing. We would also like to thank Heikki Saikkonen for his comments.

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

[3] **Arkko, J., Kuusela, J., Nuutila, E., and Tamminen, M.**, Filex: A File System Expert Written in XC. Hopefully in these proceedings, 1988.

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

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

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

[8] **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.

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

[10] **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.

[11] **Moore R. L.**, Adding Real-Time Expert System Capabilities to Large Distributed Control Systems, Control Engineering, (April), (1985).

[12] **Nuutila, E.**, AST Tools, User's Guide. Helsinki University of Technology, Laboratory of Information Processing Science, May 1988.

[13] **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.

[14] **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.

[15] **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.

[16] **Stroustrup, B.**, The C++ Programming Language. Addison-Wesley, 1986.

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

[18] **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).