The ExBed project - some experiences

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

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

ABSTRACT

We report on experiences of the ExBed project, specifically on adding a rule based expression mechanism to an existing procedural programming language (C++) and on designing and implementing a self-contained language (XE) – and its programming environment – supporting similar but more general capabilities.

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 rule-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 elsewhere (Nuutila et al. 1987), and motivate the introduction of XE. Finally we describe XE, relating it to CLU (Liskov et al. 1977). 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 be used to avoid the combinatorical explosion. Often no working memories are needed as the rules reference program variables directly. 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 rulesets. A given working memory contained references to user-defined objects of some specific types. A ruleset was a collection of rules for which the compiler produced a separate executive function. See (Arkko et al. 1988) for examples on XC. By adding rules to C++ we could have the new constructs 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 rulesets but this could have been remedied to a certain extent. However, the approach was discarded 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. 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.

From the above remarks it can be seen that, 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 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).
3. A version control system integrally connected to the development environment (LISP).
4. Compiler back ends to generate code for target environments, the main one of which has been Intel 8086.
5. Runtime systems for the target environments.

5. XE

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 those objects. 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 data type and routines which 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 operator 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
      yield(t)
    end
  end % elements
end % stack

% An example of instantiating the stack abstraction and creating a stack object
my_stack = stack[100, int use default = int$maxint]
ms: my_stack := my_stack$new()
```

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** (existentiel) quantifiers. A *quantification* consists of a quantifier, a list of variable declarations, and an iterator invocation.

Rule invocation is performed as follows. 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 can then define a stack of rules of the above type (after defining type constants *rtype* and *rstack*), push *simple* and other rules into it and iterate over all rules and their instantiations, executing the corresponding procedure for each one:

```
rtype = ruletype() yields(int, proctype())
rstack = stack[rtype]
rs: rstack := rstack$new()
push(:rs, simple)
...
for r: rtype in elements(:rs) do
  for i:int, p:proctype() in r do
    p()
  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 be implemented in XE itself, typically as follows:

rule set	A data structure containing rules and providing an iterator over these rules.
working memory	A data structure of XE objects providing one or more iterators to be used in the left-hand sides.
conflict resolution	A routine that compares two rule instantiations and retains the "better" one.
rule firing	An invocation of an action embedded in a rule instantiation.
inference engine	Code that iterates over rules in a rule set and their instantiations.

Beside providing a rule abstraction XE extends the mechanisms of CLU in several other ways. E.g., to implement rules it has added nested routines and lexical closures. XE also makes the life of a programmer easier by removing some 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 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.

6. Conclusion

We have described an approach for including production rules in an "ordinary" procedural programming language. The approach is based on abstract data types to enhance program verification and to allow for maximal user programmability. In the oral presentation we try to convey some of our experiences in designing, implementing and using the languages described.

REFERENCES

Arkko, J., Kuusela, J., Nuutila, E., and Tamminen, M., Filex: A File System Expert Written in XC. In these proceedings, 1988.

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.

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

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

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

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.

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

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.

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

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.

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.

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.

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

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).