XC - A Language for Embedded Rule Based Systems

Esko Nuutila
Juha Kuusela
Markku Tamminen
Jukka Veilahti
Jari Arkko
Nassim Bouteldja

Helsinki University of Technology
Laboratory of Information Processing Science
Otakaari 1, 02150 Espoo 15, FINLAND

August 8, 1996

Abstract

We report on experiences in the design of the programming language XC. It is an extension of C++ and combines abstract data types with rule based programming. Our design decisions are validated by three application prototypes and by benchmarking XC and OPS83. The experiences raise some critique on the RETE algorithm and on C++ as a host language. However, most of the results are also applicable to other host languages that support data abstraction. XC has been designed specifically to be used in embedded real-time expert systems.

1 Introduction

Embedded systems are claimed to be a promising industrial AI application. The ExBed project (expert system framework for embedded real-time applications) is to determine whether AI techniques can be applied within embedded real-time control systems running on a microcomputer, and how such techniques should look like. In this article we describe the aproach to programming languages taken in the project.

Fagan [Fagan 80] 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 family, using the RETE algorithm, [Brownston 85] is represented by OPS83 [Forgy 84]. It has been built so as to be tailorable and it could be used in embedded applications. See [Årzen 86] for a use of OPS systems for process control. YES/L1 [Milliken 85], a programming language descending from the YES/MVS project [Griesmer 84] is the language most similar to XC.

Hexscon [Wright 86] and Escort [Sachs 86] are more rigid languages than what we aim for, while PICON [Moore 85] is large and specialized to continuous processes. Stimulus [Robertson 85] has many objectives in common with us.

One or more serious problems arise in many of the above systems:

- Generality limited by the idea that a single expression mechanism might be suitable for all applications.
- The difficulty of verifying programs written in these languages, except by testing in the run-time environment.

- Inefficient use of memory and/or processing power. More seriously, this usage is not always controllable.

In the ExBed project we also have to address these problems.

The applications being implemented in the ExBed project are: Filex, an expert system for managing computer file space, DPR, demand pattern recognition, and DXPERT, [Bouteldja 87] automatic maintenance of a digital switching system. The first version of DXPERT uses OPS83 to obtain a standard of comparison.

The results of ExBed consist of *programming languages* that support AI paradigms such as rule based reasoning, the *development environment*, where embedded applications are written and tested, and the actual *embedded runtime environment*.

The programming language implemented first, XC, is based on C++ [Stroustrup 86]. However, our results are also applicable to other similar host languages supporting data abstraction. Based on experiences in using XC we are implementing a more specialized development environment for real-time control applications.

Sections 2 and 3 describe the design of XC. We show it possible to implement rule based programming by few general extensions to an "object oriented" programming language. We also show how abstract data types can enhance the expressiveness, validity and efficiency of rule based programming.

In Section 4 we benchmark XC and OPS83 and briefly describe our applications. Based on these experiences we discuss the choices of XC design in Section 5. The experiences seem to support our somewhat critical view on the RETE pattern match algorithm [Forgy 82]. At the same time they make us question C++ as the host language. Finally we discuss language mechanisms to be added to XC.

2 The Principal Ideas of XC

We shall try to motivate the design decisions of XC, explain why the famous RETE algorithm is not used and show what data representation and program structuring facilities we think will be necessary in a language combining production rules and procedural code.

2.1 Pattern Matching

The RETE match algorithm developed by C. L. Forgy [Forgy 79] is widely believed to be the best algorithm for finding the set of instantiated rules in a production system. However, it also causes problems, especially when used in embedded real-time control applications.

The RETE algorithm derives its efficiency from two assumed features of production systems, namely: (1) structural similarity of the left hand sides of the rules, and (2) temporal redundancy in the working memory. This means that only a small number of working memory elements change from cycle to cycle. Under these assumptions the algorithm is rather fast because it never explicitly iterates over the working memory or the set of production rules. This is achieved by using a discrimination network compiled from the LHSs of the rules. The network "indexes" the rules by working memory elements. Partial matches of the LHSs are stored in the discrimination network while complete matches are stored in the conflict set. All modifications in the working memory are propagated through the network.

Before starting the current project we implemented an expert system building tool called ET [Kuusela 86]. It combines production rules, semantic networks, and procedural code written in LISP. A slightly modified but very fast version of the RETE match algorithm was used in ET. Together with OPS83 experiences this has formed a basis for our observations.

The RETE algorithm trades memory - both dynamic and static - for speed of execution. The amount of dynamic memory used can vary much during one execution. It is also difficult to predict the amount needed. Clearly, these are unwanted properties of an algorithm to be used for real-time control.

Modifications to the working memory are rather expensive in RETE, because they must be propagated through the network. Typically some hundreds of modifications per second can be made to the working memory. Thus, if the temporal redundancy assumption is false, the algorithm runs very slowly. Unfortunately this seems to be the case in many real-time applications. Often large amounts of data must be processed in a very short time. When the RETE algorithm is too slow 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.

As was mentioned above, it takes rather long to process one modification in the working memory. During this time the contents of the RETE network are inconsistent and the program should not be interrupted. This is undesirable in real-time applications.

To overcome these problems we have studied the use of "simple-minded" production system algorithms, i.e. algorithms that do not store any information about the partial matches and must therefore iterate over the working memory and the rules to find instantiated rules.

A "simple-minded" algorithm requires little static or dynamic memory, it does not slow down the procedural code in the way that was described above, and finally, the working memory operations are more fine-grained than in RETE and therefore the algorithm can be interrupted more rapidly.

An example of such an algorithm is one that scans the rules. At every rule it generates every possible combination of working memory elements and checks whether the LHS of the rule is satisfied. The complexity of this kind of an algorithm is terrible, if used carelessly.

The important observation we have made 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 (these are described in section 5). Note also that as in Stimulus [Robertson 85] often no working memories are needed as the rules reference program variables directly. Further our process control system experts have ascertained that typical applications will be highly modular with rather small rule sets.

2.2 Data Representation

Production systems usually have some predefined and rather restricted way for representing working memory elements, i.e. the data used by the system. The classes of attribute - value pairs in OPS5 [Forgy 81] 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, the internal representation of working memory elements is often complicated and direct access to the data is impossible, because it could cause inconsistency in the saved state (network) of the algorithm. This causes problems when a production system is interfaced with procedural code. In some systems this problem is solved by using multiple representations of data [Kuusela 86, Wright 86]. This, however, requires conversions between the different representations and introduces a new problem, namely the maintenance of consistency between them.

We think that an identical data representation formalism should be used by both the procedural and rule based parts of the system, and it should be as general as possible. Therefore we have taken the concept of abstract data types as a basis. The working memories can contain objects of any abstract data type and the same objects can be referred to by procedural code.

No conversions or cumbersome access mechanisms are needed.

2.3 Program Structure

In non-AI applications the decomposition of programs into modules has long been the basic method for improving the flexibility, comprehensibility and maintainability of large systems [Parnas 72, Liskov 86]. In rule-based systems this method has usually been neglected. Instead, these systems generally consist of one linear set of rules that communicate via a global database. This has caused problems when the systems have grown bigger.

An XC program can contain many working memories and many sets of production rules. By defining several abstract data types, working memories, and sets of rules, the user can decompose the problem into manageable parts.

3 Description of XC

In this section we describe the basic constructs of XC. They have been implemented as language extensions to C++. This is only an informal description and some more advanced features of XC are omitted. Note that it is the "object oriented" capabilities of C++ that have given XC much of its expressive power and that have also eased our implementation.

3.1 Working Memories

A working memory is a collection of pointers to objects of specified types. The working memory type specification is

```
wm(T1,T2,...,Tn)
```

where `T1,T2,...,Tn` are type names. It can be used like any type name in C++ e.g. `int` or `char`.

Example 1:

```
/* Define some C++ classes */

class Ball { ... };
class Box { ... };
class Stick { ... };

/* Define a working memory local_wm
   that can contain objects of types
   Ball and Box, but not those of
   e.g. Stick. */

 wm(Ball, Box) local_wm;
```

There are two operations for accessing working memories: `assert` and `retract`. Assert is used to insert objects into a working memory and retract is used to remove objects from a working memory.

Example 2:

```
/* A variable of type Ball. */
Ball b;

/* Insert the new object into
   working memory local_wm */
local_wm.assert(&b);

/* Remove it from local_wm */
local_wm.retract(&b);

/* A variable of type Stick. */
Stick s;

/* This is illegal. The compiler
   will detect a type error */
local_wm.assert(&s);
```

A working memory is implemented as a C++ class by the XC translator.

3.2 Rules and Rulesets

An XC rule definition consists of a header and a body. The rule body contains the condition and the action of the rule. The condition is a C++ expression and the action is a C++ block statement. The rule header consists of the name of the rule and the rule parameters. A rule parameter is a variable that iterates over the objects of a given type in a given working memory. Different parameters can iterate over different working memories. The format for a rule parameter is:

```
<type> <variable> 'in' <working-memory>
```

Example 3:

```
/* Assume that type Ball has
   function size(). Then rule r1
   removes all Balls that are
   bigger than MAX_SIZE from
   local_wm. */

rule r1(Ball* b in local_wm) {
  when (b->size() > MAX_SIZE) {
    local_wm.retract(b);
  }
}
```

In XC rules are divided into modules called rulesets. An XC ruleset is like a C++ class, but it can contain production rule declarations and definitions in addition to data and functions. This convention of distinguishing between declaration and definition is the same that is used with functions of a class in C++. Note the similarity with the definition/implementation concepts of Modula 2 and similar languages.

Example 4:

```
ruleset Blocks_World {
  wm(Stick, Ball, Box) local_wm;
    ...
  /* Declare rule Move_Ball */
  rule Move_Ball(Ball* b in local_wm);
  /* Define rule Take_Box */
  rule Take_Box(Box* bx in local_wm) {
    ...
  }
  ...
}
...
/* Define rule Move_Ball of ruleset
   Blocks_World */
rule Blocks_World::Move_Ball(Ball* b in
                             local_wm) {
  ...
}
```

Rulesets are translated to C++ classes by the XC compiler. Every rule is translated to two functions, one for testing its condition and the other one for performing its action.

3.3 Ruleset Execution

The XC compiler produces automatically an executive function called *run* for every ruleset. This routine executes the *match - conflict resolution - act* loop until no applicable rules are found or the function *halt* is called. Halt is a parameterless function that is also generated automatically for every ruleset.

In the first part of the loop - *match* - the executive function searches for rules whose condition part is satisfied. For each rule it repeatedly binds the parameters to all possible objects in the working memories and evaluates the condition of the rule.

The next phase - *conflict resolution* - depends on the strategy used. There exists a simple default strategy that selects the first satisified rule to execution. Robertson [Robertson 85] claims that this usually suffices in real-time systems. If this is not appropriate the user can supply a conflict resolution strategy of his own. This is accomplished by writing a function that compares rule instances. Every time a new rule instance is found it is compared with the best instance so far and, if it is better, the new instance is stored as the best instance.

Note that compared to the RETE algorithm we gain much in memory efficiency in not storing all rule instances - and sorting them at conflict resolution.

In the *act* phase the action part of the best rule instance is executed and the loop starts again.

3.4 Derived Rulesets

In C++ it is possible to define class hierarchies. The user can define a base class and new classes *derived* from it. The derived classes *inherit* the functions and data of the base class.

New functions and data can be defined in the derived class and those of the base class can be redefined. Inheritance facilitates code factoring, i.e. code to perform a particular task is found in only one place instead of being replicated in many places. It is believed that this eases the task of software maintenance [Pascoe 86].

In XC a ruleset hierarchy can be constructed in the same way as a class hierarchy. New and redefined rules, data and functions can be defined in the derived rulesets. Rules can also be removed by defining them in the derived ruleset with an empty body. The conflict resolution strategy is inherited by default, but it can also be redefined in the derived ruleset.

3.5 Embedding

The support for continuosly operating real-time processes in XC resembles closely what has been offered by YES/MVS [Griesmer 84] and YES/L1 [Milliken 85]. The communication between independent XC processes is based on the message passing services of the underlying operating system. Whenever a process is a ruleset its match - conflict resolution - act loop contains a phase during wich external messages are recognized and transmitted to the working memories. The concept of time as such is not built into XC but is handled by library data types and procedures.

3.6 The XC Compiler

The XC compiler is implemented as a preprocessor that translates XC definitions into C++ code which is then compiled by the C++ compiler. The preprocessor is a rather simple program that leaves as much work as possible to the C++ compiler. For example, all type checking is done by the C++ compiler.

4 Experiences with XC

4.1 Applications

The expressiveness and other characteristics of XC are being tested in the following applications.

4.1.1 Filex

Filex is an expert file system manager implemented in XC. During nights it makes a pass over the file system and tries to detect unnecessary files or files it thinks need “care”. If unnecessary files are found, a list of them is created for every user. Users are notified as they log in. Action on the files can be made using an interactive program. Actions include removing the file, moving it to an archieve file, etc.

Users and system administrators can override Filex default rules by a special file “.filexrc”. In it files can be declared to be precious or unnecessary. Precious files are not inspected by filex at all. Unnecessary files are removed if they have been touched within 24 hours.

If no orders from the user are given, Filex makes an educated guess of whether a file is unnecessary or not. This is done by first evaluating three basic properties:

- Usage. Probability that the file will be read or referenced to.
- Information. Information content of the file.
- Space. How much space would deleting this file free?

The basic properties are inferred by using rules on file name, size and age information. When different rules arrive at different estimates of basic properties, an average is taken. Below are some example rules.

If the file name is ”core”, the file is likely to be a result of a core dump and usually recreatable:

```
rule core()
{
  when (streql(basename(filename),"core")) {
    Information_content = 0.0;
    Usage_probability = 0.0;
  }
}
```

If a file has been read, written to, or created during the last 24 hours, it is likely that it will be needed again.

```
rule read_day()
{
  when (read_modify_or_create_time(filename)
        < DAY) {
    Usage_probability = 1.0;
  }
}
```

When there are more than one link to a single file, deleting one of these links would not free any space.

```
rule several_links()
{
  when (filestatus.st_nlinks > 1) {
    Space = 0;
  }
}
```

The basic properties are then used to make a decision of the appropriate action on the file. Normally, the action will be a "no operation". Operations are selected by rules such as the one below. If properties show that the file very probably will not be needed again, it is certainly recreatable and it is using more than 1 percent of the free disk space, delete it without even consulting the user.

```
rule select_rm()
{
  when (Information_content == 0.0 &&
        Usage_probability < 0.3 &&
Space/Free_space_on_disk > 0.01) {
    Operation =
          remove_file_without_consulting_user;
  }
}
```

Filex is in daily use and at the present it contains 36 default rules.

4.1.2 DXPERT

DXPERT [Bouteldja 87] is a knowledge-based system for interpreting alarms of the DX200 digital exchange. The first prototype of DXPERT has been implemented on a stand-alone MS-DOS based microcomputer in the OPS83 language. XC will be used to embed DXPERT inside the DX200 system and tackle the real-time issues of monitoring alarms continuously.

4.1.3 DPR - Demand Pattern Recognition

DPR considers a group of equipment servicing randomly varying demand. The demand is only partially observable by the sensors of the machinery. The reconstruction and filtering of data is the first XC prototype. XC will also be used in the actual statistical forecaster, which is coded mostly in C++.

4.2 Benchmarking XC

Our synthetic test problems benchmark, beside XC, various strategies used in production systems and allow some comparison between XC and OPS83 that uses the RETE algorithm.

The XC programs were run on an ATT 3B2/400. OPS83 programs were tested using a "PC/AT" Mikro-Mikko 3. In several "calibration" runs with identical C-programs the processing power of the PC ranged between 50% (32 bit data) and 120% (16 bit data) of that of the 3B2. (For identical OPS83 runs the processing power of Mikro-Mikko 3 was twice that of an IBM PC/AT).

Only very small problems could be run on within the 400 - 500 KB allotted to OPS83.

4.2.1 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 both XC and OPS83. This is also true of all the other benchmarks.

The programs moved 14 disks by 16384 rule firings (Table 1).

4.2.2 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 n_{obj} elements in both. All objects possess n_{attr} different attributes (e.g., color, material and size) each with n_{val} 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.

Two problem variants were used. In the first variant, 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, e.g.

```
when(ball->color == box->color &&
     ball->material == box->material) {
        PutInto(box,ball);
        retract(ball)
}
```

In the second variant one equality test was replaced with a less-or-equal test, e.g.

```
when(ball->size <= box->size &&
     ball->color == box->color) {
        PutInto(box,ball);
        retract(ball)
}
```

With n_{attr} attributes we have $n_{attr}(n_{attr}-1)/2$ such rules to express a match in both variants of the problem.

Tables 2 and 3 show the largest problems that could be run with OPS83.

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. Note, however, that the attribute test problem is a kind of a bad case for OPS83 in this respect.

The memory requirements of XC depend linearly on both the number of objects and the number of rules.

4.2.3 Shortest Path Tests

All shortest distances from a given node to all others on a network can be determined by a one rule program. To test such programs we constructed random grid graphs.

The problem was solved using variants of Moore's [Moore 57] algorithm: After asserting a node the following rule is fired until the working memory is empty:

```
FOR ANY node n in working memory DO
   FOR EACH neighbour n1 of n DO
      compute new distance to n1;
      IF distance(n1) has improved
        THEN assert n1;
   retract(n);
```

The problem was solved in three ways (Table 4):

1. The XC solution. The working memory is by default structured as a LIFO queue.
2. A C++ program "identical" to the XC program.
3. A C++ program using a deque [Pape 82].

5 Evaluation and Extensions

We have been satisfied with the expressive power of XC. Further, from tables 1-3 one can observe that XC was more memory efficient than OPS83, and even considering the different processing powers of Mikro-Mikko 3 and 3B2, XC was more time efficient than OPS83, for these problems.

The shortest path tests emphasize the fact that problem specific algorithms and data structures determine efficiency. The LIFO structure used by all standard conflict resolution strategies of the OPS family happens to be the worst possible for the shortest path problem (Table 4). With a deque or FIFO queue instead, the XC (or OPS83) rule firing count would have decreased by a factor of 1000 in the largest problem of Table 4.

The overhead factor of the XC rule firing mechanism compared to ordinary procedural C++ code is "only" 4 as shown in Table 4.

By allowing problem specific optimizations of production system execution XC can be made even more efficient. The most promising optimizations, which we are implementing in XC, are

- User definable working memory data structures, e.g. hash tables, according to the operations that are needed. For instance the working memories of XC are currently treated as lists and all element combinations searched for a match. In the attribute problem, however, a hashed data structure would make the searching considerably faster.
- The possibility to restrict the search of matching working memory elements by using query optimization techniques. We are already able to perform automatic query optimization for working memories containing only a single object class, and this seems sufficient.
- User definable mechanisms to modify rule instance search. At the present the only choice is between a cyclic or repeated testing order of rules that can be ordered (prioritized) by the user.

We have concluded that the approach taken in XC is valid. Note, however, that we do not claim that XC is generally more time efficient than OPS83 and other systems based on the RETE algorithm. The observations only let us believe that the XC approach enhanced with the above optimizations can be used in embedded applications. Specifically the rule system of XC is more space efficient and does not slow down the execution of procedural code like rule systems based the RETE algorithm (see section 2).

It seems that in these applications the programmer should be given expression mechanisms to control rule based execution instead of having to blindly trust a rather opaque general purpose algorithm.

We have also found some problems in the approach taken. They are mainly caused by the use of C++ as the host language.

The problem with C++ is that it is a large language without formal specification and it carries along the misfeatures of C. It does not support data abstraction as well as desired and it does not directly support concurrent and realtime programming.

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.

Another problem is that there is no high level debugger for C++. To build a debugger for C++ is to build a compiler for it. However, a trace mechanism for rule execution can be easily implemented.

6 Conclusions

We have reported experiences in the design of the programming language XC for embedded real-time expert systems.

We have shown it possible to implement rule based programming by very few and general extensions to an "object oriented" programming language. We have also demonstrated the usefulness of abstract data types in rule based programming.

By trying out XC, as well as OPS83, in real prototype applications we have tested the language mechanisms proposed. Further, by benchmarking problems we have been able to report on the performance of XC. The experiments raise some critique on the widely accepted RETE pattern matching algorithm.

One of our main conclusions is that there can be no single pre-packaged data structure and algorithm optimal for a wide class of rule based systems. Hopefully, however, good data structures can be derived automatically by a compiler - separately for each program. However, until this is possible we must provide the programmer with language mechanisms facilitating the tailoring of data structures and algorithms to his problem - and not vice versa.

Acknowledgements

The ExBed project has been funded by the Technology Development Centre together with two private companies.

This papers has greatly benifited from a careful reading by Martti Mäntylä, Heikki Saikkonen, Markku Syrjänen and Seppo Törmä.

References

[Brownston 85] Brownston, L., Farrell, R and Kant, E., Programming Expert systems in OPS5. Addison-Wesley, 1985.

[Bouteldja 87] Bouteldja, M. DXPERT: An expert system for complex alarm interpretation in the DX200 digital switching system. The 3rd International Expert Systems Conference, 2-4 June 1987, Learned Information, London, pp.477-486.

[Fagan 80] Fagan, L., Ventilator manager: a program to provide on-line consultative advice in the intensive care unit. PhD thesis, Computer Science Department, Stanfoord University, 1980.

[Forgy 79] Forgy, C.L., On the Efficient Implementation of Production Systems, Carnagie-Mellon University, Department of Computer Science, 1979

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

[Forgy 82] 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 84] Forgy, C.L., The OPS83 report. Report CMU-CS-84-133, Carnegie-Mellon University, Department of Computer Science, 1984.

[Griesmer 84] Griesmer, J.H., Hong, S.J., Karnaugh, M., Schor, M.I., Ennis, R.L., Klein, D.A., Milliken, K.R. and VanWoerkom, H.M., YES/MVS: A continuous real time expert system. Proc. AAAI National Conference 1984, pp. 130-136.

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

[Liskov 77] Liskov, B., Snyder, A., Atkinson, R., Shaffert, C., Abstraction Mechanisms in CLU, Comm. ACM, vol. 20, no. 8, 1977, pp. 564-572.

[Liskov 86] Liskov, B., Guttag, J., Abstraction and Specification in Program Development, The MIT Press, Cambridge, Massachusetts, 1986

[Milliken 85] 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 57] Moore, E.F., The shortest path through a maze. In Proc. Int. Symp. Theory of Switching, Harward Univ., 1957, Part 2.

[Moore 85] Moore R. L., Adding real-time expert system capabilities to large distributed control systems, Control Engineering, (April), (1985).

[Nuutila 86] Nuutila E., Kuusela, J., Specification of XC. Internal report, Helsinki University of Technology, July 1986.

[Pape 82] Pape, U., Algorithm 562, Shortest path lengths. TOMS 1982, pp. 450-455.

[Parnas 72] Parnas, D.L., On the criteria to be used in composing systems into modules, Comm. ACM, vol 15, no. 12, Dec. 1972

[Pascoe 86] Pascoe, G.A., Elements of object-oriented programming, Byte, Aug. 1986.

[Robertson 85] 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 86] 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 86] Stroustrup, B. The C++ Programming Language. Addison-Wesley, 1986

[Wright 86] 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).

[Årzen 86] Årzen K-E., Expert systems for process control. Proc. 1st Int. Conf. on Applications of AI in Enginering Practice, Southampton, UK, April 15-18, 1986.

Tables

Method	*memory*	*exec.time*	*firings/s*
XC	39078 bytes	25 s	655
OPS83	59780 bytes	292 s	56
Recursive C++ code	17025 bytes	.77 s	21200

Table 1: The Towers of Hanoi, 14 disks, 16384 rule firings.

n_{obj}	n_{rules}	**OPS83**			**XC**		
		exec. time	*firings/s*	*memory*	*exec. time*	*firings/s*	*memory*
170	28	27 s	6.2	>400K	4.6 s	36	50K
60	105	15 s	4.0	>400K	1.8 s	33	87K

Table 2: Comparisons between OPS83 and XC, Attribute Problem 1.

n_{obj}	n_{rules}	**OPS83**			**XC**		
		exec. time	*firings/s*	*memory*	*exec. time*	*firings/s*	*memory*
60	28	7 s	8.5	>400K	0.5 s	120	40K
28	105	7 s	4.0	>400K	0.3 s	93	84K

Table 3: Comparisons between OPS83 and XC, Attribute Problem 2.

	LIFO			DEQUE	
nodes	*firings*	*time*		*firings*	*time*
		XC	C++		
100	2820	1.53 s	0.4 s	100	20 ms
625	282424	78.2 s	19.2 s	625	120 ms
1089	1125173	551.1 s	132.1 s	1089	210 ms

Table 4: Processing time for the shortest path problem.