Filex: A File System Expert Written in XC

Jari Arkko, Juha Kuusela, Esko Nuutila, Markku Tamminen

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

Abstract. *We present the rule based programming language XC, an extension of C++, and Filex, an expert system written in it. Filex is designed for monitoring Unix file system usage.*

Introduction

The ExBed project (expert system framework for embedded applications) was established to develop tools for applying expert system techniques within embedded systems. Specifically, the constructor of embedded systems was to be provided with a rule-based expression mechanism well integrated with an "ordinary" programming language. Literature (e.g., [1]) lead to the choice of the forward chaining inference strategy in this context. Also, 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. The language we report on, XC[2] , was obtained by extending an existing language, C++[3].

Filex is a file system expert written in XC. It is embedded in the sense that it automatically searches for unnecessary files during night and informs users of these as they log in. Users can interactively respond to the suggestions of Filex. They can also control the behaviour of Filex by user-specific permanent control files, or case by case, if they need to. Filex consists of three processes. Of these the embedded one contains the "intelligence", which is implemented in XC. The rest is written in C++. The close relationship between XC and C++ allows the power of C++ to be used in the rules.

We shall first describe XC. Then we give an overview of Filex and show examples of its rules. Finally we try to evaluate our experiences.

The XC Language

XC is an extension of C++. Thus it supports object oriented and rule based programming. XC has been designed specifically to be used in embedded real-time expert systems. The widely accepted RETE pattern match algorithm[4] is not used in XC because it may cause problems in real-time control applications, as explained in more detail in[2]. In this section we describe informally the basic constructs of XC, omitting some of the more advanced features.

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 for 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:

```
Ball b; /* A variable of type Ball. */
/* Insert the new object into working memory local_wm: */
local_wm.assert(&b);
local_wm.retract(&b);     /* Remove it from local_wm */
Stick s; /* A variable of type Stick. */
/* 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.

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 type Ball has function size(). Then rule r1
removes all Balls 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 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 languages such as Modula 2.

Example 4:

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

In the final example rule ball_on_top prints a message when it notices that a ball is located on top of a box.

Example 5:

```
rule ball_on_top(Ball* b in local_wm, Box* x in local_wm) {
  when (b.x == x.x &&
        b.y == x.y &&
        b.z == x.z + x.height) {
   printf("Ball is on top of the Box.");
  }
}
```

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. The XC compiler also 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[5] 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. In the *act* phase the action part of the best rule instance is executed and the loop starts again.

Note that compared to the RETE algorithm XC gains much in memory efficiency in not storing all rule instances - and sorting them at conflict resolution. Also, its "simple-minded" inference engine is often quite time-efficient, in contrast to what might be assumed. First of all, a ruleset is compiled into directly executable code, as are the rules. Thus there is no interpretation overhead. Normally there are also many user controllable optimizations that help to avoid the combinatorial explosion. Further, typical applications are modular with small rule sets and often no working memories are needed as the rules reference program variables directly[5].

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. 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.

The support for continuosly operating real-time processes in XC resembles closely what has been offered by YES/MVS[1]. 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 which external messages are recognized and transmitted to the working memories. The concept of time as such is not built into XC but is to be handled by library data types and procedures.

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. Rulesets can be compiled separately. However, if a rule is changed the whole ruleset must be recompiled.

Overview of Filex

The motivation behind Filex was twofold: this kind of a program was needed and at the same it served as a test application for the XC language.

The goal of Filex is to help with common problems of file storage:

- File space is used up by unnecessary files. For instance, a Unix program may dump its memory to a file without anybody ever using this file. Filex is to detect and preferably remove such files.

- Available space on a file system gradually decreases as users generate files without removing them later. This could be called the archive phenomenon: users want to keep all their information stored and readily available. When file space becomes insufficient we have to options: either move files to an archive tape or remove some of them altogether. A surprisingly large portion of file space is consumed by files the users have received from, e.g., the news system and stored for some future use which usually never comes. The second objective of Filex is to remind the owners of these files.

- o The available file space fluctuates and sometimes goes to zero. When this happens quick actions are needed to make the use of the system possible again. Automatic measures are preferred to those requiring operators. The third objective of Filex is to adapt to changes in the available file space and preferably help in situations where there is no file space available.
- o Users themselves have problems in managing their files because there is a large number of them and it is hard to remember to remove all unnecessary ones. Also, when removing files himself, a user may give a command like "rm * .o" leading to disastrous concequences. The fourth objective of Filex is to help users manage their files.

To meet these requirements, Filex performs the following operations:

- o The file system is searched periodically (typically at night) to detect waste of space. For this Filex evaluates all files and decides whether or not they seem necessary.
- o If some files are classified as unnecessary, their owners are notified as they log in. These messages keep coming twice at every log in until the user invokes the filex command described below or the files become classified as necessary. Thus there is a penalty of extraneous messages to users who choose to ignore the suggestions of Filex.
- o A user may use the filex command to perform operations on the files classified as unnecessary. The user can remove the file, move it to a staging area for loading onto magnetic tape, etc. If a user disagrees with Filex and thinks a file is necessary he can prevent Filex from complaining about it again. If a user agrees with Filex that a file is totally unnecessary, he can order Filex to remove it always when seen (see below). Also, it is possible for the user to himself initiate a search for unnecessary files.
- o As explained above, the user can command Filex to regard certain files as necessary or totally unnecessary. These instructions can be given from the filex command or they can be placed in special control files. In this way the user can, for instance, ensure that as Filex makes its search for the unnecessary files it always removes "$*$.o" files from some directory.

The Filex system consists of three independent processes: the file system search process, one process to wait for the users to log in and send them messages, and the filex command process. All the processes use a common database to store results of the file system search and a common decision module to evaluate the necessity of files.

Filex Decision Rules

Normally Filex searches over the whole file system. However, decisions are made file by file. The basic information available for decision is the name and status information of a file: modify date, size, owner, number of links etc. The file system is polled to see how much free space there is. Important information comes also from the control files written by users. The decision module is free to use any methods known to determine the usefulness of a file, but it must not use too much time on a single file. This restricts, e.g., extensive analysis of file contents. In some cases Filex looks for other files in the same directory to make a decision.

The decision process is performed in four stages:

1 The information available on a file is fetched.

2 The fetched information is used to determine four basic properties of the file:

- o Usage. This property measures how often the file is referred to or what is the probability that it will be read or referred to in the future.
- o Information. This property measures the information content of the file or the work needed to recreate it. For instance, object files typically have little information content if the sources exist.
- o Space. Filex considers the extra space obtained by removing a file, and thus large files are the ones to be removed first. This measure is not a direct function of file size, since removing a large file will free no space when there are several links to it. (Filex could be changed to consider all the links at the same time.)
- o Usefulness of non-disk storage. This measure tells if it is sensible to save the file, e.g., on tape. Some files are better suited for archival storage than others; e.g., typically it is not useful to store an object file on tape, whereas this form of storage is well suited for many source files.

3 Estimates of each basic property suggested by different rules are combined to form a single value. Currently the mean-value operator is used for this.

4 According to the basic property values arrived at, a suitable operation is suggested for the file. Operations possible are: no-operation, moving the file to an archive tape, using a file compression utility to decrease the size of the file, deleting the file, and deleting the file immediately. All other operations except the last one require the user's confirmation.

Below we present some example rules from the Filex decision module. The rules are slightly simplified, ignoring, e.g., the problem of refraction[6].

If the file name is "core", the file is likely to be a result of a core dump and is usually recreatable. Core files also get used very seldom. This rule does not remove all "core" files, since if such a file has been read after its creation or created very recently another rule will fire and generate another usage probability.

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

If the file name ends in ".o" and there is a source with the same name, it is probable that the file can be created by compiling the source. Also, one would normally not save "*.o" files on tape.

```
rule star_o_c()
{
  when (string_ends_in(filename,'.o') &&
          file_exists(string_upto(filename,'.o'),".c",".lsp" ...))  {
    Information_content = 0.2;
    Usage_probability = 0.4;
    Long_time_save = no;
  }
}
```

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;
  }
}
```

If the file name is "mbox", it is likely to contain information the user wants to look up. As it is known that mailbox files grow large, it is regrettable that we are not able to discard the useless mails. But the purpose of this rule is to prevent Filex from complaining about mailbox files all the time.

```
rule mbox()
{
  when (streql(basename(filename),"mbox")) {
    Usage_probability = 0.8;
  }
}
```

The basic properties are then used to make a decision about the appropriate action on a file. Normally, the action will be a "no operation". Operations are selected by rules such as the one below. If the basic 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 = delete_file_with_not_confirmation_from_the_user;
  }
}
```

No conflict resolution is required in Filex: it suffices that the inference engine checks rules in their definition order, which is the XC default strategy.

Experiences from the Use of Filex

Filex has been in daily use since fall 1986 and is continuously able to detect a large number of unnecessary files. It has adapted well to changes in the amount of free space in the file system and suggestions from it come generally only when free space is low. However, when free space is low most users get many messages. Some users have found this annoying since they have been forced to respond.

It seems that Filex would need somewhat better facilities to distinguish between necessary and unnecessary files. It also seems that the primary criterion for detecting a file as unnecessary is age. However, Filex does not complain about files that are obviously necessary. Also, the user control facilities enable the users to easily declare their files as necessary. These facilities have proved to be very helpful.

Filex has been able to remove files without consulting the users when free space has been low, but maybe not as many as hoped. This problem is due to the fact that there are few rules that can tell with certaintity that the file is not needed again. The normal "core" etc. files do not stay long in our file system, however.

The importance of the user interface has been greater than we though at the beginning of the project. Users need to converse with Filex and they want to do it in an easy and fast way. Currently, the user interface is somewhat awkward, which reduces the popularity of Filex.

If Filex would be developed further, we would like to emphasize the control possibilities of users. Ideally, the users could specify rules that Filex would obey. Of course, we would also like to have more default rules, e.g., rules for system files. In general, we think that file system management is a promising area for expert systems, and systems like Filex will be necessary tools in the future. This will be the case even for personal computers, as their hard disks grow.

Experiences from the Use of XC

XC was found to be a good tool for building Filex. The close relationship between

C++ and XC enabled us to use the whole C++ language in rules; e.g., normal C++ expressions in the left-hand sides. We feel that it is important to be able to use complex problem-specific data structures with rules. The types of working memory elements found in, e.g., OPS5[6] simply do not cover the needs of all rule based systems. However, we must note that beside rules XC does not bring anything new to C++; e.g., we do not have frames at our disposal.

The simplicity of XC was found valuable, because simple rules could be written for simple situations without extra overhead from complex rule and working memory definitions. In our view, this will lower the level where rule based programming becomes a practical solution. However, the simplicity was also a problem for us, since some of the features we needed were missing from XC rules. The worst problem was the lack of a refraction mechanism[6], i.e., a mechanism for disabling the rules from firing again with the same data. This drawback follows from the choice that XC rule left-hand sides can reference any data freely. In Filex this problem was solved with a flag for every rule, but we know that this is not always possible.

Since the volume of data is huge in file systems, speed of execution was a concern in the Filex system. With our simple set of rules and minimal amount of working memory elements XC offers a comfortable efficiency. We believe that this will be so even in many larger systems, due to problem modularity.

Acknowledgements

The ExBed project has been funded by the Technology Development Centre, Nokia Corporation and KONE Corporation. Filex was designed and implemented as a student project by Jari Arkko, Jari Heikkinen, Kimmo Talvitie and Jari Tomminen.

References

[1] 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.

[2] E. Nuutila. J. Kuusela, M. Tamminen, J. Veilahti, J. Arkko, N. Bouteldja: XC - A Language for Embedded Rule based Systems. SIGPLAN Notices 10/84.

[3] B. Stroustrup: The C++ programming language. Addison-Wesley, 1986.

[4] C.L. Forgy: Rete: a fast algorithm for the many pattern/many object pattern match problem. Artificial Intelligence 19(1982)1, pp. 17-37.

[5] J. Robertson: 'STIMULUS' - a base language for real time expert systems. Proc. Conf. on AI and advanced Computer Technology, Wiesbaden, 24-26 Sept., 1985, 15 pp.

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