ILog Rules, embedding Rules in C++ : results and limits

Patrick Albert
Ilog. 2, Av. Gallieni,
94253 Gentilly CEDEX, France
email: albert@ilog.fr

August, 12, 1994

Abstract

This short paper presents the main ideas at the basis of the design of Ilog Rules, a commercial Rete based production system embedded in C++. We focus on the aspects related to the communications between the rules and C++, both at compile-time and at run-time.

1 Introduction

Ilog Rules is a commercially available product that extends C++ with rule-based programming. It was the result of an industrialization contract between Ilog and Thomson-LCR aimed at turning XRete, a multi-language Rete-based production system designed and prototyped by researchers of Thomson-LCR, into a product.

At the early stages of its design Ilog Rules had three main objectives : language independence, embeddability and performances. Language independence meant that the rules were highly independent of the underlying language (one of C, C++ and Le-Lisp), but following the market demand, Ilog Rules is now solely oriented to the support of C++.

Below is an example of rule :

```
(defrule warnChangeSector 0
 ?r:(Radar)
 ?p:(Plane radar=?r2 &<>?r &<>UNDEF)
 (bind int ?delay {?r->willInterceptAt(?p)})
```

```
{?delay<MaxDelay}
(not (PassControl from=?r2 to=?r plane=?p))
->
{printf("The delay is: %d\n",?delay);}
(assert (PassControl from=?r2 to=?r plane=?p)))
```

As a result, the current release of the product supports forward chaining rules on C++ objects, and a integrated TMS for "logical" facts. The development environment is composed of a set of graphic components that can be plugged as addons to most of the exisiting C++ development environments.

2 Embeddability

This aspect has been the key point behind the design of Ilog Rules. It led to the following characteristics :

- the rules must be applied to objects of the host language ;
- the rules may support the use of expressions of the host language both in the premises and in the conclusion parts.

2.1 Not reinventing a new object language

We can foresee two trends in the evolution of Rule-based programming languages. The first option is to provide an integrated proprietary development environment integrating Objects, Rules, Data access, and Graphics. The second option is to extend a given language with rule-based programming.

The integrated approach has the advantage of providing a large set of tools tightly coupled within an homogeneous environment, with the main drawback of beeing proprietary and having integration problems.

The embedded approach — that follows the global trend of the library approach promoted by the object oriented community — is open and, as it only adds rules to an existing language, allowing the developers to use any tool that is compliant with the application language.

We chose the embedded approach for two reasons : a global strategy of the company to be an actor in the class libraries market, and a conviction that the integrated approach would have problems because of its intrinsic proprietary nature.

As C++ is becoming the de-facto standard for industrial object oriented developments we chose to define Ilog Rules as an extension of C++.

2.2 Inlining expressions

Rules elements are made of filters applied to objects, of computations and of procedures calls. An embedded production system must allow the users to use any expression of the base language in a rule body.

Functions may be used to express complex conditions and statements may be used to express actions in the conclusion part.

2.3 Compiling versus interpreting.

Given these objectives — use of standard objects and support of inline expressions — the rules machinery has to follow the target language machinery.

If the target language is interpreted, the rules may be interpreted, but if the target language is compiled, the result of the compilation of the rules have to be compiled by the compiler of the target language in order to support the interpretation of the inlines.

This point appears to be of interest, because the market demands seems to evolve toward both embeddability and interpretation...

2.4 Supporting objects

Supporting Objects within rules is already a long story. The main approach is to use typed variables, the name of the types being the name of the classes of the object model.

The support of inheritance is guided by the following rule: If B is a direct or remote subclass of A, any rule that filters instances of the class A may also be applied to instances of the class B.

Finally, the semantic of the "modify" operation has to be adapted to the semantic of objects. The basic operations made upon objects are the creation, the deletion and the modification of an object's field. The "retract+assert" semantic of the modify in OPS based systems has to be adapted using the following rule: If a field f of an object of class C is modified, then the field has to be physically modified, and only the patterns that filter the field f of an object of class A, or lower, must be reevaluated.

2.5 Communication between Rules and Objects

The analysis of the communication may be analyzed at two different levels: the static communication, at compile-time, and the dynamic communication, at run-time.

2.5.1 Compile-time communication

By compilation we mean the process of reading the rules and transforming the textual definitions into an executable program.

Depending upon syntax choices, different problems may arise. Ilog Rules choose to use an OPS-based syntax. As the target language is C++, the rules are transformed into a C++ program that implement their semantic. The compiler then need two types of information : types of objects's fields and accessors.

The types are used for early type-checking one one hand, and for generating a program that is compliant with C++ typing rules on the other hand.

The accessors are used to transform the logical access to the objects's fields in the OPS syntax into operations that will retrieve or modify fields's values.

This information is provided to Ilog Rules using the `defimplementation` construct. This construct defines the mapping between the C++ classes and the names used in the rules patterns.

A specific construct `define_default_implementation` defines how accessors's names may be built out of the fields's names.

An example of `defimplementation` may be :

```
(define_default_implementation
  reader {?object->?Attribute()}
  writer {?object->?Attribute(?value)}
  constructor {new ?type}
  destructor  {delete ?object}))

(defimplementation Plane ()
  ((name type {char*})
   (posx type {float})
   (posy type {float})
   (velx type {float})
   (vely type {float})
   (radar type {Radar*})
   (time  type {int}
          reader {?object->getTime()}
          writer {?object->setTime(?value)})))
```

2.5.2 Run-time communication

At run-time the communication between rules and objects relies upon five operations: object creation, object deletion, field access (read and write), and notification.

The firsts five operations, related to data access are performed using the code specified in the `defimplementation` definitions, and work fine. To manage smoothly the creation and deletion of objects, Ilog Rule distinguishes between `internal` and `external` objects.

Internal objects are objects whose lifecycle is managed by the rules engine. The `retract` operator applied to an internal object will call the C++ `delete` operator. Conversely, the retract of an `external` object will not delete the object. The only effect of the retract in this case is to remove the object from the rules memories.

This functionality allows the users to apply rules to objects which are managed by code outside the rules, as well as to provide some automatic memory management for the objects which are created and deleted by the rules.

The last operation, "notification", is an important subject as long as embeddability is concerned. Within Ilog Rules's contract which is to be able to apply rules onto any user defined object, different cases may happen.

The right-hand-side operations `assert`, `retract` and `modify` perform the corresponding physical operations, and the notification that entails the computation of the new conflict set. When an object is modified outside of the scope of the rules, the user has to notify the rule engine that it has to recompute the patterns related to this modification. In order to do this, Ilog Rule's compiler generates accessors that notify. These accessors are methods of the classes of type `Context` that represent the methods and datas used to manage the rete network.

This approach works quite well, but it appears that users are reluctant to use these methods in their application code.

Another method would be to use the informations provided by the defimplementation constructs to generate C++ classes with accessors that automatically notify the modifications.

3 Conclusion

We have shortly presented Ilog Rules, a commercial product that adds rule-based programming to C++.

After three years of practical use, we have found that though the product seems to be well adapted to the customers needs, the notification problem is still a key topic, and that different solutions may be proposed, in order to fit the many contexts in which rules may be added to an object oriented program.