Encapsulating Rules *

Daniel P. Miranker

miranker@cs.utexas.edu
Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

Abstract

Venus and CERS are two experimental C++ embedded rule languages comprising very different approaches to encapsulation. CERS embraces C++ semantics with little alteration; methods could be defined using collections of rules rather than procedural C++ code. In Venus, C++ procedural code and rule definitions are disjoint. Venus rules are organized into parameterized collections called modules. Modules may be nested by listing the name of a module and its actual arguments on the right-hand side of a rule.

Venus semantics, reminiscent of the database nested transaction model, lead to both a strict data driven firing of rules and modules as well as sequences of module execution consistent with procedural intuition. A RETE compatible, demand-driven, (LEAPS-match), implementation minimizes the performance impact on the embedded object system. Our experiences with application developments suggest that the Venus approach is superior.

1 Introduction

Object definitions include the specification of internal state and a collection of methods to operate on that state and return values. Rules in forward-chaining rule languages comprise predicates on a state and actions on that state. There is considerable flexibility on the scoping of the predicates and the actions and therefore flexibility concerning the integration of rule-based programming and object-oriented programming.

*This research was funded in part by the State of Texas Advanced Technology Program, the Applied Research Laboratories Internal Research and Development Program, ARPA grant number DABT63-92-0042 and Texas Instruments.

In this position paper we will outline the approaches that we have taken in two different C++ embedded rule systems, CERS and Venus [1,7] and relate the strengths and weaknesses of each approach including applications development.

2 Semantics

2.1 CERS

The semantics of CERS is a simple extension to C++. The basic idea, (which was never implemented in its full generality), was to allow individual methods within a C++ class definition to be defined using either the traditional approach using procedural C++ code or as a collection of rules. The execution of a rule-based method was governed by operational OPS style semantics derived from the RETE match.

2.2 Venus

In Venus, rules are grouped into modules. We can view a module as a class definition and an instance of a module as an object. Module definitions include a formal parameter list. See Figure ??. Where it does not introduce ambiguity, we have adopted C++ syntax for Venus constructs.

A Venus module fires enabled rules until a fixed point is reached. A rule instantiation fires only if the RHS action results in a state change. The entire RHS action is treated as a single atomic action. Rules have an optional user specified priority. Instantiations of rules with the same priority, including instantiations of the same rule, are selected for firing by a fair non-deterministic policy. For historic reasons the current implementation dictates that fairness be defined such that a rule instantiation is fired at most once, where *once* has the OPS5 definition[2].

```
/* This rule can be read as:  for all s in
r, if there does not exist a symmetric
element t, then create it and add it to r.
*/

module  enforce_symmetry(Relation r[])
{
   rule enforce;
   from r[*] s;
   from r[?] t;
      if(!(s.domain_oid == t.range_oid &&
           s.range_oid == t.domain_oid)) {

              Relation *ins = new Relation;
              ins->domain_oid = s.range_oid;
              ins->range_oid = s.domain_oid;
              r.insert(ins);
   }
}
```

The equivalent rule in OPS5 is

```
(p enforce
   (<r> ^domain_oid <did> ^range_oid <rid>)
  -(<r> ^domain_oid <rid> ^range_oid <did>)
-->
   (make <r> ^domain_oid <rid>
             ^range_oid <did>))
```

Figure 1: Venus Rule Syntax

Module names and their actual parameters may be listed with the actions in a rule's RHS. If the rules predicate is satisfied then rules in the embedded module may also fire. Modules may be embedded arbitrarily deep. The same module with the same or different actual parameters may be activated from multiple places in the code. The current implementation supports only a static the module hierarchy in the form of a directed, acyclic activation graph. Module pointers and recursive module activations are interesting avenues to pursue.

It is convenient, but incorrect, to think of a module as being called. An explicit firing of the guard rule is not necessary for execution of rules in the child module. If the guard is enabled, any state change affecting the satisfaction of a rule predicate in a child module may trigger rule firings within that module. It does follow from atomicity that once control enters a module the firing of rules in descendant modules will take place as a depth-first traversal of the module-activation graph. Similarly, it is incorrect to think of a rule's predicate as a guard to be distributed over all the rules that are in a module listed on the RHS.

3 Scoping of Predicates and Meta-level Containers

In general, rule predicates may range over the internal state of the object in which they are defined. More commonly, it is desired that the predicates range over a meta-level container of objects and instantiations of existentially quantified rules "pick" particular objects out of the container.

The latter behavior suggests that the place for rule systems is above the object system in a meta-level. Consequently, the separation of Venus modules from C++ object definitions has worked better than the integration with C++ in CERS. In fact the CERS approach, intrinsically leads to confusion. In CERS, the meta-level containers are, through explicit procedural definition, associated with an object, but rule activation is associated with a single method. In Venus code containers of objects are explicitly listed in the actual parameter lists of the module activations. Thus, the behavior of a CERS program is determined by procedural statements distributed in time and space through a larger program. The behavior of a Venus program is specified in single calling statement.

4 Triggering and the Container Interface

An ongoing issue in both systems is really a problem with C++; C++ does not implicitly create meta-level containers of object instances. The application developer, either by embedding code in the constructors or elsewhere, must explictely create and fill containers. CERS, ILOG rules and CADIE all require developers to use a predefined container [7,3]. Since C++ supports multiple inheritance this requirement, by itself, is usually not onerous. However, if the objects were created before the rule-component was conceived, as might happen in a large system that includes a persistent object store, then special containers can introduce significant overhead.

A second problem with CERS as well as systems such as ILOG rules and CADIE is there adherence to operational rule execution semantics determined by the RETE match. The primary symptoms of this problem include

- the need for explicit notification and triggering mechanisms connecting the inference engine to the object system.

- the burdensome overhead required to maintain consistency of internal data structures of the inference engine and the state of the object system.

Venus overcomes these limitations. In conjunction with fixed-point semantics the Venus implementation is based on a demand driven execution mechanism, the LEAPS match[6]. LEAPS has been formally shown to provide execution consistent with TREAT and RETE based systems. Further, LEAPS is asymptotically faster than TREAT and RETE and completely collapses the space requirements of rule-based matching. Consequences of this approach include,

- explicit notification of object updates is limited to occasions when external C++ functions are called from Venus code, but, NOT vica-versa.

- when explicit notification is called for the actions of the matcher are limited to a simple stack push.

- no observable performance impact on the object system when rules are not firing.

Perhaps most important, the Venus engine is able to inference over instances stored in preexisting containers defined by the user. To our knowledge Venus is the first such system. The user must tell the Venus compiler, by means of a type declaration, the name and type of the container as well as the names of the iteration methods. The container's cursors must support the following properties.

- an arbitrary number of cursors on a single container may be open at once.

- a cursor must not return the same object more than once.

- a cursor may not return an object that has been deleted.

- a cursor must not return an object that was added to the fficontainer after the cursor was opened.

This list of restrictions on the behavior of cursors is not as difficult as it appears. A simple temporal qualification on standard SQL cursors satisfies these requirements.

5 Discussion

CERS was used for several small programs including rewrites of the Waltz and Manners benchmarks. Venus is in ongoing use and development. The Waltz and Manners benchmarks have been coded as well as a rewrite of a 100 rule military simulation and the original coding of a knowledge-based diagnostic system.

The CERS embedding rules in C++ objects did introduce the control paradigm of message passing to rule-based programming. Compared to our previous experiences with monolithic rule systems such as OPS5, CLIPS and ART-IM, CERS programs had much better structure[4,5]. Nevertheless, much of the declarative advantages of rule programming was lost. Behavior became procedurally distributed in both time and space. Correctness of the resulting programs relied heavily on the programming prowess of the developers and debugging.

The declarative nesting mechanisms of Venus have proved to be extremely valuable. When the nesting is combined with fixed-point semantics much of a programs specification becomes an executable invariants. The developers of the military simulation program reported writing an order of magnitude less code than in the original system. Much of the development of the diagnosis program was simply casting the english specifications into Venus syntax.

We conclude that elements of a more successful embedded object-oriented production system embody:

- Recognition the rule processing is more commonly used as a meta-level concept.

- Operational semantics of rule execution derived from the RETE match should be avoided.

- A combination of fixed-point semantics and demand driven execution leads to a better system.

References

[1] J.C. Browne et.al. A New Approach to Modularity in Rule-Based Programming to appear, Proceedings of the Int. Conf. on Tools for Articial Intelligence, November 1994.

[2] C.L. Forgy. Ops5 user's manual. Technical Report CMU-CS-81-135, Department of Computer Science, Carnegie Mellon University, July 1981.

[3] D.W. Franke. Imbedding Rule Inferencing in Applications. *IEEE Expert*, pages 8–14, Dec. 1990.

[4] J.C. Giarratano. *CLIPS User's Guide, Version 4.2.* Artificial Intelligence Section, Lyndon B. Johnson Space Center, 1988.

[5] Inference Corp. *Art Reference Manual*, 1987.

[6] D. P. Miranker, D. Brant, B.J. Lofaso, and D. Gadbois. On the Performance of Lazy Matching in Production Systems. In *Proceedings of the 1990 National Conference on Artificial Intelligence*, pages 685–692. AAAI, July 1990.

[7] D. P. Miranker, F.H. Burke, J. J. Steele, J. Kolts, and D. R. Haug. The C++ Embeddable Rule System. *Int. Journal on Artificial Intelligence Tools*, 2(1):33–46, 1993. Also in the Proc. of the 1991 Int. Conf. on Tools for Artificial Intelligence.

[8] D.P. Miranker and B. J. Lofaso. The Organization and Performance of a TREAT Based Production System Compiler. *IEEE Trans. on Knowledge and Data Engineering*, pages 3–10, March 1991.