RAL/C and RAL/C++: Rule-Based Extensions to C and C++

Charles L. Forgy
Production Systems Technologies, Inc.
July 1994

At Production Systems Technologies, we have developed and implemented a rule-based extension to C++ called RAL/C++. This is a companion to an earlier integrated rule and object language, RAL/C. Based on our experiences in developing and using the RAL systems, we have formed a number of opinions about what features such systems should have. In this paper, I would like to discuss what we believe are some of the more important of these issues. Section 2 below describes two features that we believe are essential for an integrated rule and object system. Section 3 describes three features that, although not essential, have proven quite useful.

Our experience with the RAL systems indicates that efficiency in object oriented rule systems is not a problem. That is, we have found that the same algorithms that provide efficient conventional rule interpreters work equally well on object-oriented rule systems. Some benchmark results are given in Section 4. The benchmarks show that a complex RAL system that makes heavy use of objects is only 6% slower than a system that uses only conventional working memory elements. (The RAL systems use a new match algorithm, Rete 2, which is significantly faster than standard Rete; RAL runs this benchmark more than 200 times as fast as Rete-based CLIPS 6.0, for example.)

1. Overview Of RAL/C++

RAL/C++ is an extended dialect of C++. RAL/C++ adds forward-chaining production rules as a fully-supported programming paradigm. It is implemented with a preprocessor that translates RAL modules into ordinary C++ code; the resulting C++ code is compiled by the usual system C++ compiler. That means that all legal C++ programs are also legal RAL/C++ programs.

The rules in RAL/C++ are very much like those in OPS5, CLIPS, etc. except that they are based on C++ syntax rather than a proprietary (non-standard) base. For example, to add 1 to a variable x in OPS5 you would write

```
(bind <x> (compute <x> + 1))
```

In CLIPS you would write

```
(bind ?x (+ ?x 1))
```

To add 1 to x in RAL/C++ you could write

```
x = x+1;
```

or if you prefer

```
++x;
```

RAL/C++ uses ordinary C++ syntax for variables, expressions, statements, etc. We believe this is essential for achieving true integration of the rules with the rest of C++. In integrated systems, it is often the case that a single function or rule has to do both ordinary C++ operations and rule-specific operations such as changing a working memory element. If you have a different syntax for these two classes of operations, combining them in a single function will be clumsy.

In RAL, a rule RHS (right hand side) is syntactically and semantically identical to an ordinary function body. Anything that can occur in an ordinary function can occur in an RHS, and vice versa. A few new statements have been added to permit creating and working on WMEs, but these statements can be used in ordinary functions as well as in rule RHSs. The LHSs (left hand sides) of RAL/C++ rules look like OPS5 LHSs. The principal difference is that since RAL/C++ is an extended dialect of C++, the expressions in the LHS are C++ expressions rather than the non-standard expressions used in OPS5. A typical RAL/C++ rule is:

```
flows
{
        // if a tank T1 has lost material
        C1 (tank   status==^^loss;   name::T1;   delta::Delta1;)
        // and a tank T2 has gained material
        C2 (tank   status==^^gain;   name::T2;   delta::Delta2;)
        // and a pipe connects T1 to T2
        (pipe    from==T1;    to==T2;)
-->
        // then conclude material was pumped from T1 to T2
        double m = min(Delta1, Delta2);
        cout << m << " gallons were pumped from tank " << T1
             << " to tank " << T2 << ".\n";
        modify C1 {
            C1->delta -= m;
            if (C1->delta < DELTATHRESHOLD) C1->status=NIL;
            }
        modify C2 {
            C2->delta -= m;
            if (C2->delta < DELTATHRESHOLD) C2->status=NIL;
            }
}
```

A RAL/C++ working memory element is an ordinary C++ struct or class. A different keyword is used in the type declaration because the translator needs to know which structs are going to appear in working memory. A typical small declaration is:

```
wmedef HeatedTank:Tank
```

```
    {
        double  temp;
        int     heaterOn;
    };
```

This says that element type HeatedTank is derived from type Tank. To the members in type Tank are added the double member temp and the int member heaterOn.

2. Important Features

Two features of RAL/C++ are particularly important to the goal of achieving of seamless integration of rules and C++. I believe that any system that aims to integrate rules with C++ must have these features.

2.1. Support for all C++ Types

One of the design requirements of RAL/C++ was that the rules had to be able to support all C++ types, including user-defined types. Specifically, RAL was designed so that, for any C++ type X: The WMEs can contain fields of type X or type X*. The rules can access C++ variables of type X or X*, they can call functions that operate on X or X*, and they can send messages to objects of type X. In short, they can do anything than a standard function can do. I think it should be obvious why this is feature is essential in any combined rule and object system.

2.2. Recursive Rules

Another design requirements of RAL/C++ was that the runtime system for the rules in RAL/C++ had to be reentrant. A non-reentrant runtime system results in a system with very strange semantics. Obviously, in any integrated system, it is important that rules be able to call ordinary functions, and that ordinary functions be able to invoke rules. A non-reentrant runtime system causes the odd restriction that a function can invoke rules only if that function was not itself called (either directly or indirectly) by a rule. For instance, if a rule calls function F1, and F1 then calls F2, in a non-reentrant system, F2 is prohibited from invoking rules. Having this kind of restriction is obviously inelegant, but that is not the real point here. The point is that this restriction places a substantial burden on the programmer trying to use rules, and in fact makes it impossible to integrate rules and functions closely. We have seen many RAL systems that simply could not have been written if was not possible to invoke rules recursively.

3. Other Features

There are a number of other features of RAL/C++ that, while not essential in the sense of the previous two features, are nonetheless useful in an integrated rule-object system.

3.1. Shadow Objects and an Object-Based Runtime

The object-oriented capabilities of languages like C++ make it possible to structure rule interpreters more cleanly than was done in earlier languages like OPS5. The RAL systems take advantage of this ability by defining what we call "shadow" objects. Each WME has a shadow object. There is also an object to represent each instantiation in the conflict set. The shadow object contains various information about the item it shadows. For example, WME shadow objects usually contain time stamps. Using the subclassing abilities of object systems, users can define their own shadow objects that contain and use information that the runtime system's base shadow objects did not have. In this way, users can very easily extend the functionality of the runtime system. This is of course the natural way to organize programs in an object-based system. We simply did not have this option when we designed systems like OPS5.

Some may question why it is necessary to have a separate object for each WME if the WME is itself already an object. The main reason is that the WMEs and the shadow objects need different class hierarchies. The WMEs' class hierarchies are determined by the structure of knowledge in the problem at hand. The shadow objects' class hierarchies are determined by the functionality desired in the runtime system. Of course, by using multiple inheritance the two class hierarchies could be combined for each WME. However, using multiple inheritance for this purpose seems clumsy; using separate simpler object hierarchies gives a cleaner and more modular design.

3.2. Multiple Knowledge Sources

The object-oriented capabilities of C++ make it possible to organize the knowledge in a rule-based system very cleanly by making the rule bases themselves into objects. In RAL this is done by creating what we call "knowledge sources." A knowledge source consists of a working memory, a conflict set, and one or more rulesets. If the knowledge engineers using RAL wish, each knowledge source in a program may make use of a different set of shadow objects. Each knowledge source is represented by an object of a certain type. A knowledge source is activated by sending a message to the object telling it to begin executing rules.

3.3. Contexts and Context Stacks

One new feature in the RAL systems is something we call context stacks. A context stack is an explicit stack of goals for the rules to work on. RAL rules are organized into groups called rulesets. Each ruleset has one context that it works on. The rules in that ruleset can fire only when that context is active. When a context is pushed onto a stack, that context is activated, and the context that was at the top of the stack is deactivated. When a context stack is popped, the context that then becomes the new top of stack is reactivated. Thus pushing and popping context stacks is essentially the same as calling and returning from subroutines in a conventional procedural language. We believe that this explicit control makes the RAL systems easier for C or C++ programmers to learn than old systems like OPS5 that rely on ad-hoc tricks with conflict resolution to implement control.

4. Efficiency

In object systems like that in C++ (and the object system in RAL/C) as much binding as possible is done at compile time. This means that functions that operate on objects in these systems are almost as efficient as functions that operate on ordinary structs. We have done some benchmarking with RAL/C that shows that there is also almost no penalty to having the rules operate on objects. The WALTZ benchmark program from the University of Texas benchmark suite is a good one to illustrate this point because it causes rules to do a great deal of pattern matching. For a structure-based program, we translated WALTZ directly from OPS5 to RAL. For a comparison system, we then modified WALTZ to use objects rather than structs for WMEs. In the modified WALTZ, we used a separate class for every member a WME has. The results for RAL/C on a Sparcstation 1 running the 25-region version of the problem are shown below. As this shows, the penalty for using objects was only 6%.

Standard WALTZ 25 Run Time (Seconds)	Object-Based WALTZ 25 Run Time (Seconds)	Ratio
15.7	16.6	1.06

RAL/C uses the new Rete 2 match algorithm, which makes it quite efficient compared to systems based on standard Rete. The following table shows the relative speeds of RAL/C and CLIPS 6.0, a popular system based on standard Rete. As can be seen there, RAL is more than 200 times as fast as CLIPS on this relatively complex problem. These runs were made on a Decstation 3100 because CLIPS 6.0 would not run well on a Sparcstation.

RAL WALTZ 25 Run Time (Seconds)	CLIPS 6.0 WALTZ 25 Run Time (Seconds)	Ratio
10.6	2,858.6	270