Path-Based Production Rules

James M. Crawford
Computational Intelligence
Research Laboratory
1269 University of Oregon
Eugene, OR 97403-1269

Daniel Dvorak
Diane Litman
Anil Mishra
Peter F. Patel-Schneider
AT&T Bell Laboratories
600 Mountain Ave.
Murray Hill, NJ 07974-0636

1 Introduction

One of the goals of this workshop was to understand how to seamlessly integrate first-order rules into an object-oriented framework. We propose a novel approach to such an integration that relinquishes the complex matching algorithms of RETE-based approaches and instead capitalizes on the access-paths inherent in object-oriented domain models. Such an approach forms the heart of a forward-chaining embedded object-oriented production system (EOOPS) currently under development within AT&T for use on projects that require both procedural and rule-based code.

Our principal goal in designing R++ has been to bring production rules into an object oriented framework in as natural a way as possible. One advantage of object-oriented languages is that modifications to an object generally affect only that object and perhaps other neighboring objects linked to that object. This type of "locality" is one of the primary sources of power for object-oriented programming—especially in large programming efforts. However, locality is completely lost when OPS-style rules are introduced because they allow joins between unrelated objects. Instead, we advocate that the antecedents of rules start with some root object and follow access-paths out from that object. (Such *path-based rules* are not new, having appeared explicitly as "access-limited rules" in the Algernon implementation of access-limited logic [Crawford & Kuipers, 91, Crawford, 90]).

The second part of integrating rules into object-oriented systems is to view rules as members of classes—much like member functions. This point has several implications. First, rules will be associated with classes and inherited (and overridden) like member functions. Second, rules will respect the same access privileges/restrictions as member functions. Finally, the syntax of the rules should be as close as possible to the syntax of the host language. We discuss this view of rules in more detail in the next section.

A final point is that in the near term, a pragmatic strategy is to design EOOPSs as extensions to existing object-oriented languages. As one of the dominant object-oriented languages, C++ [Ellis & Stroustrup, 90] is an obvious candidate to extend, even if it is not an ideal language for this purpose.

We are currently designing and implementing a forward-chaining EOOPS built on top of C++ that adheres to these criteria. This system, called R++, will be used within AT&T on projects that require an object-oriented model of some complex domain or device, and that use both procedural and rule-based code. R++ takes the basic idea of path-based rules and rule attachment from Algernon and makes the many changes required for seamless integration with C++.

2 Path-Based Rules

Path-based rules act directly on objects of the object-oriented language rather than a separate working memory. Further, they strictly follow paths of accesses and do not perform arbitrary joins. Therefore rule instantiations have one root object and a number (possibly zero) of other objects, all reachable from this root object by some access-path. In contrast to general pattern-matching (as in OPS), the simplicity of path-based inference allows for a relatively simple and efficient inference algorithm.

The natural way to make the rules fit into the existing constructs of object oriented languages is to associate rules with classes, just as data and functions are associated with classes. In effect we view rules as "rule members" of classes. To further this resemblance, rules are given class-specific names, as are the data and functions associated with the class.

The association of rules with classes provides a type for the root variable of a rule, namely that class with which the rule is associated. As accesses from a particular class are typed in essentially all object-oriented languages, the other variables of the rule can also be automatically typed.

To illustrate these points, consider a rule that ensures that a person's children under the age of 18 are dependents of that person's spouse. This rule would be associated with the class `Person`, and would have two other variables, both of type `Person`. The rule would be given a name such as "children_spouse_dependents". In instantiations of the rule, the root object would be a person, the second object would be his or her spouse, and the third object would be one of his or her children. Note that both the second and third objects are reachable from the first object (the implicit *this* object, using C++ terminology) using an access-path (*e.g.*, $this \rightarrow mate$, for the second object).

Making the internal syntax of rules close to the syntax of the host language is not difficult. The right-hand side of a rule can be just a statement or sequence of statements in the language. The left-hand side of a rule can be just a sequence of boolean expressions in the language, interspersed with variable bindings, that can look much like variable definitions. For example, the `children_spouse_dependents` rule could be written in R++ as shown in figure 1.

The clause "`Person * spouse = mate`" declares a variable of type `Person` and sets it to the value of `this->mate`, succeeding only if the value is non-zero. The `@` in the next clause is new. This is an iteration construct and should be read as

```
class Person {
    int age;
    Person * mate;
    Set_of_p⟨Person⟩ children;
    Set_of_p⟨Person⟩ dependents;
    ...
    rule children_spouse_dependents;
};

rule Person::children_spouse_dependents {
    Person * spouse = mate &&
    Person * child @ children &&
    child→ age < 18
⇒
    spouse→dependents.insert(child);
}
```

Figure 1: `children_spouse_dependents` rule

"at" or "in". This clause thus says that the rule will iterate over all values in the `children` data member, which is declared to contain a set of pointers to persons. Alternatively, one can think of this clause as being satisfied for each child in the set `this→children`.

The requirement that rule control be effected by object-oriented language mechanisms indicates against explicit control of rules by rule activation, rule deactivation, or rule groupings. It also suggests that rules not have priorities. Instead rule control can be handled in the normal course of object-oriented activities. The *creation* of an instance of a class brings the object to the attention of the member rules of the class. Thus, whenever an instance of `Person` is created, rule `children_spouse_dependents` will be considered. The only way to remove an object from consideration by the rules of its class is to destroy it. Rules follow all appropriate existing access-paths.

The above sparse rule control can be augmented by use of the class hierarchy. If allowed by the object-oriented language for member functions, as is the case in C++, a sub-class can have a rule with the same name as a rule in a super-class. Such a rule will override the rule in the super-class for instances of the sub-class.

The requirement that rules interact correctly with procedural code entails that rules be triggered directly by *changes* to the data they reference. This method of triggering rules also completes rule control by means of object-oriented mechanisms. Thus, rule `children_spouse_dependents` will also be considered whenever *either* a rule or procedural code causes, say, a new child to be inserted into `children`.

3 The Role of Search

One of the most notable differences between path-based rules and the more traditional OPS-style rules is that path-based rules do not support the kind of pattern matching sometimes seen as the heart of OPS applications. The access-path requirement is, of course, not a fundamental limitation on the *expressivity* of path-based rules since one can always create a static data-member for each class and use it to enumerate all instances of the class. The real difference between path-based rules and OPS-style rules is that a path-based rule system does not provide automated support for efficient pattern matching/search.

If we step back and look at rule-based programming we can identify two *independent* contributions/sources of power. First, rules offer a more abstract declarative language for expressing certain programming constructs (*e.g.*, the maintenance of invariants). Second, production rules efficiently support certain types of search/pattern matching—again offering a higher level of abstraction by freeing the programmer from having to explicitly write search code.

As we move into an object-oriented framework we should separately consider each of these contributions. The use of rules as a more abstract language often provides a significant reduction in code size and an increase in readability and maintainability. Clearly this is useful. The utility of automated search is perhaps less clear. One of the primary reasons for using object-oriented languages is because of the modularity/reusability they promise to large programming efforts. Because they allow for arbitrary joins with other objects, traditional OPS-style rules tend to destroy an important part of this modularity. Pragmatically, it can be difficult to convince object-oriented programmers to use a rule system if they know the system may forward chain from a change somewhere in the object-base and make a change *anywhere* else in the object-base.

Since path-based rule systems do not provide automated support for pattern-matching, the programmer has to code any necessary search procedurally. But since path-based rule systems are already tightly integrated into a procedural language this is not nearly as high a cost as it would have been in older more "stand-alone" production systems. The advantage of the path-based approach is that the locality property of object-oriented languages is preserved: changes to an object will always only propagate to "neighbors" of that object—other objects reachable by following a known access-path.

4 Embedded in C++

As we do not currently have the resources to extend the C++ language itself, we have implemented R++ as a translator to C++. This is not completely ideal because of certain aspects of the C++ language, but has allowed us to rapidly create several prototype implementations with quite different characteristics. The approach we have taken is similar to the approach used in the Ode [Agrawal & Gehani, 89] system, an object-oriented database management system that incorporates O++ [Dar et al., 93] an extension to C++. We expect to combine these two systems in the future.

C++ is not an ideal language to extend. An ideal language would allow for easy augmentation of object creation and destruction at various points in the process. It would also allow for augmentation of all modifications to data members of classes. In short, it would allow a comprehensive notification mechanism whereby the rule-system would be notified whenever a relevant change occurred to any object. In C++ such a mechanism would require complete parsing of *all* code in a program.

In order to render the notification problem tractable and avoid parsing all the C++ code in R++ programs, we have placed two restrictions on the code: all modifications to rule-affecting data members of classes be done via stereotypic access functions, and no casts be performed on classes used by rules. This is not an ideal solution, but may be required for all C++-based EOOPSs developed in the near future.

5 Summary

A theme in our work on R++ is that rules should *extend* C++, not replace it. As "reactive" member functions, rules extend the behavior of classes in a data-driven manner. Accordingly, we do not advocate that entire applications be programmed in rules; instead, we believe that rules are most sensibly used for specific tasks such as enforcing invariants in the domain model, monitoring for important states of the model, and propagating changes through the model.

In order to naturally embed into an object-oriented framework, R++ rules are path-based; object-oriented domain models make important relationships explicit as links or "paths" between objects, enabling path-based rules to build directly on such designs without the need for arbitrary joins. Because of their path-based antecedents, such rules have a more localized effect than OPS-style rules and can therefore be expected to be more predictable to the programmer. Further, the relative simplicity of path-based inference compared to general pattern matching has allowed an efficient implementation that compiles rules into the host language, with no run-time inference engine. Since each path-based rule is associated with a class (like a data-driven method), it can also be inherited (and overridden) in the same way as methods are in the host object-oriented language. In essence, path-based rules define a kind of "rule member" that naturally extends the object-oriented framework.

References

[Agrawal & Gehani, 89] R. Agrawal and N. H. Gehani, 1989, "Ode (Object Database and Environment): The Language and the Data Model", *Proceedings ACM-SIGMOD 1989 Int'l Conf. Management of Data*, pp. 36-45.

[Crawford & Kuipers, 91] J. M. Crawford and B. J. Kuipers, 1991. "Negation and proof by contradiction in Access-Limited Logic". *Proceedings of the Ninth National Conference on Artificial Intelligence (AAAI-91).*

[Crawford, 90] J. M. Crawford, 1990. "Access-Limited Logic—A language for knowledge representation", doctoral dissertation, Department of Computer Sciences, The University of Texas at Austin. Also published as Technical Report AI 90-141, Artificial Intelligence Laboratory, The University of Texas at Austin.

[Dar et al., 93] S. Dar, R. Agrawal and N. H. Gehani, 1993, "The O++ Database Programming Language: Implementation and Experience", *Proceedings IEEE 9th int'l Conf. Data Engineering*, Vienna, Austria.

[Ellis & Stroustrup, 90] Margaret A. Ellis and Bjarne Stroustrup, 1990, *The Annotated C++ Reference Manual*, Addison-Wesley Publishing Company.