Path-Based Rules in Object-Oriented Programming

James M. Crawford*
CIRL
University of Oregon
1600 Millrace, Suite 108
Eugene, OR 97403-1269 USA

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

Abstract

Object-oriented programming has recently emerged as one of the most important programming paradigms. While object-oriented programming clearly owes an intellectual debt to AI, it appears to be displacing some AI techniques, such as rule-based programming, from the marketplace. This need not be so as path-based rules—forward-chaining production rules that are restricted to follow pointers between objects—fit into the object-oriented paradigm in a clean and elegant way. The combination of path-based rules and object-oriented programming should be useful in AI applications, and in the more general problem of transferring AI techniques to the larger computer science community.

To this day, forward-chaining production rules remain largely unused in mainstream object-oriented applications. One reason for this lack of use is, of course, that many production rule systems (such as most versions of OPS (Cooper & Wogrin 1988), etc.) are written in LISP, and not in a major object-oriented language.

However even in the new rule systems that claim integration with an object-oriented language, the integration is not adequate. Many of these rule systems (such as ART-IM (Inf 1987)) operate on their own data (working memory elements), not the objects of their base language. Applications thus need to move information between the working memory and object storage. Other systems (such as CERS (Miranker *et al.* 1993)) require special calls to bring objects of the language to the attention of the rule system. Although this means that information does not need to be moved, it does require facilities outside of the object-oriented paradigm to control rules.

*Supported in part by ARPA/Rome Labs under grant numbers F30602-93-C-0031 and F30602-95-1-0023, by the National Science Foundation under grant number IRI-94 12205, and by the Air Force Office of Scientific Research under grant number F49620-92-J-0384. Some of this work was also done while the author was at AT&T Bell Laboratories.

The rules of object-oriented, production-rule systems (including the above systems, as well as ILOG rules (Albert 1994) and Rete++ (Hal 1993)) exist outside the object system, and thus are not organized in the same way as classes and objects. Further, even when the production rules directly operate on objects they still do not act like the rest of the system. In particular, production rules in all the above systems can form joins between unrelated objects, thus violating the locality assumptions implicit in object-oriented programming.

However, rules have many uses within the object-oriented paradigm. Rules can be used to enforce invariants, check constraints, react to events and states, update dependent objects, and even remove dangling pointers. Therefore, it would be useful to add rules to an object-oriented programming language in a manner that merges the rules naturally into the rest of the language.

For this effort to be a success, it must be easy for programmers who are not experts on rules to use the system. We argue that a number of stringent requirements have to be satisfied for production rules to be widely accepted outside of AI:

- A seamless integration of rules, objects, and procedural code is needed, not just an add-on of rules to an object-oriented programming language.
- The base language has to be a commonly-used object-oriented programming language. This is critical even if the programming language is less than ideal for the purpose, since the goal of the effort is to bring AI technology to mainstream programmers.
- The addition of rules has to be a "small" addition, in that it must require little in the way of education for effective use. This dictates that there be little new syntax required in the rules, that the rule concepts fit well with the object-oriented programming

language, and that the rules are subservient to the object-oriented paradigm.

- Nevertheless, the addition of rules has to be a "big" addition, in that it provides useful new functionality.

The driving philosophy behind the addition must be to take AI technology and change it as necessary to fit into the mainstream programming community.

We have augmented a common object-oriented programming language with rules in a way that answers the objections and satisfies the requirements above. We have chosen the most-common object-oriented programming language, C++ (Stroustrup 1991)[1], as the base language for our additions. Our rules work directly on C++ objects. The rules are organized into the class hierarchy, with a syntax and semantics very similar to C++ member functions. They cannot form arbitrary joins, being restricted to following the pointers in C++ objects, giving them no more access capabilities than member functions.[2] Therefore, the rules form a "small" addition to C++. They are data-driven not heuristic, i.e., they are always active and have no certainty factors. The rules can perform all the activities mentioned above, and can provide "big" benefits.

We have called our rule system **R++**, a deliberate attempt to call attention to its close relationship with C++.

Rules in an Object-Oriented Language

Path-Based Rules

The most significant conceptual difference between OPS-style rules and R++ rules is that R++ rules are *path-based*. Path-based rules do not permit the formation of arbitrary joins between unrelated objects. Instead, the conditions of path-based rules must start with some root object and can reach other objects only by following paths of pointers (*access paths*). Such *path-based rules* are not new, having appeared explicitly as "access-limited rules" in the Algernon implementation of access-limited logic (Crawford & Kuipers 1991; Crawford 1990)).

Aside from this limitation on their conditions, path-based rules are similar in concept to OPS-style rules, having a condition and an action. The conditions of path-based rules are evaluated in response to certain activities in the rest of the system. If the condition is satisfied, then the action of the rule is executed (for details on this see section). The action of a path-based rule has access to the object bindings from its condition, just as an OPS-style rule has.

Because of the restriction that it must follow access paths, a path-based rule would not be able to perform an action for every pair of persons such that the first person is older than the second. A path-based rule, however, could perform actions on pairs of persons such that the second is a child of the first, provided that the "child" relationship was provided in person objects. Thus a path-based rule could be used to enforce invariants such as every child must be younger than its parents.

The access limitation of path-based rules is not a hindrance in an object-oriented programming language because object-oriented design makes important relationships explicit as inter-object pointers. In fact, this "limitation" is seen as an advantage because path-based rules respect the locality implicit in the relations in a domain model and cannot violate this locality as OPS-style rules can. In addition, path-based rules admit a relatively simple and efficient implementation compared to the general pattern-matching inference engines of OPS-style languages.

Rules as Class Members

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, and have the rules work directly on the objects that are instances of the class. In effect we view rules as "member rules" 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 object of a rule, namely that class with which the rule is associated. Further, the association provides a specification for when objects are brought to the attention of rules, namely that the *creation* of an instance of a class brings the object to the attention of the member rules of the class. The only way to remove an object from consideration by the rules of its classes is to destroy it. Therefore a rule associated with a class is active on all objects that belong to the class.

Because rules are members of a class, they can access the private members of the class, just as member functions can. Further, a rule cannot access the private members of another class unless it has been declared a friend of that class, just as for member functions.

Rules can be overridden in sub-classes, just as member functions can. 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

[1] There is no technical reason why the ideas in R++ could not be added to a different object-oriented language, such as Smalltalk, Objective-C, or even Java. We just chose the most-used object-oriented language.

[2] It is possible to simulate joins in R++ by adding in explicit links, but this is only feasible in limited cases.

sub-class.

Rule Control

Because rules work directly on all instances of a class, there is no need for a separate working memory nor even a mechanism to keep track of which objects are active in the rule system. The object-oriented paradigm provides all the control required, obviating any need for control of rules via explicit activation or deactivation of rules or via grouping of rules.

The conditions of rules monitor data in various objects. Changes to this data may cause the condition of the rule to be satisfied, or satisfied in a different manner than before. Therefore, the conditions of rules are re-evaluated whenever the data they inspect changes or when a new object of of the appropriate type is constructed. We call these changes *relevant changes*.[3] There is no need for any other mechanism for causing the conditions of rules to be evaluated—changes to object data is the only mechanism required or allowed.

The requirement that rules fit into the object-oriented language paradigm suggests that rules not have priorities, so we have not provided priorities for rules, other than the simply requiring that rules defined in a sub-class are considered before rules from super-classes are. (Of course, super-class rules that are overridden in the sub-class are *not* run on instances of the sub-class at all.) This priority scheme fits into the object-oriented paradigm. This was a deliberate choice that has benefits, less to worry about and faster rules, as well as drawbacks, less control over the rules.

Because there is no complex priority scheme there is no rule agenda that collects all rule activations that are waiting to run.[4] Rule activations are simply run in a depth-first fashion, as soon after they are discovered as possible. That is, a change to monitored data causes the rule activations for that data to start running; a later change to other monitored data causes the rule activations for the new change to start running; rule activations for the old change that have not yet been run are deferred until the rule activations for the new change are all finished running.

Rule Syntax and Semantics

To make R++ rules fit easily in C++, and to make them as easy for C++ programmers to learn, R++ rules look and act as much like the rest of C++ as possible. Externally, R++ rules are declared and defined in a manner similar to C++ class members, as indicated above, and as shown in the simplified rule syntax given in Figure 1.

```
<rule declaration> ::= rule <identifier> ;
<rule definition> ::= rule <class name>::<identifier> {
                          <condition> => <action>
                      }
<action> ::= <statement>+
<condition> ::= <binding> |
                <boolean expression> |
                <binding> && <condition> |
                <boolean expression> && <condition>
<binding> ::=
    <class name> * <identifier> = <expression> |
    <class name> * <identifier> @ <field> |
    <class name> * <identifier> @ <variable>-><field>
```

Figure 1: Simplified R++ Rule Syntax

The condition (left-hand side) of an R++ rule is a sequence of C++ boolean expressions interspersed with variable bindings. The boolean expressions, and the expressions in the bindings can, of course, use the variables bound earlier, just as in C++.

The bindings in a condition look very much like C++ variable definitions. The first kind of binding, for example "Person * mate = spouse", looks just like a C++ variable definition, and simply sets a variable to the value of an expression, succeeding if that value is non-null. The example binding declares a variable of type Person and sets it to the value of this->spouse, succeeding only if the value is non-null. The other kinds of bindings, for example "Person * child @ children", are similar to C++ variable definitions but use "@" instead of "=". These are branch bindings, where the "@" should be read as "at" or "in", and they bind a variable to elements of a set of values. The example branch binding says to iterate over all values in the children data member, which is declared to be a set of pointers to persons, succeeding for each element of the set.

A condition is evaluated in the obvious way. It succeeds for those successful bindings that make the boolean expressions evaluate to "true".

The action (right-hand side) of an R++ rule is just a sequence of C++ statements. In the action of a rule, the variables bound in its condition can be used as expected.

[3]The use of functions in the condition of rules is a problem for R++ as it cannot, in general, analyze the data used inside a function. Relevant change is defined as change to the data mentioned directly in the condition of a rule, including argument to functions, but not including change to data used in functions called in the rule condition.

[4]A rule activation is a rule plus a set of bindings of the variables in the antecedent of the rule to objects.

Rule Execution

As indicated above, there are three parts to rule execution:

1. triggering of rules by relevant change,

2. subsequent evaluation of the rule condition, possibly delayed if other rules have been triggered by the same change, and

3. execution of the rule action if the rule condition was satisfied.

The conditions of R++ rules are evaluated only in response to relevant change (including relevant construction). However, the portion of the condition that is evaluated in response to a relevant change is unspecified, as is the order of evaluation. Thus rule conditions are normally side-effect free.

The actions of R++ rules are executed only when their conditions successfully evaluate to true, and only on the data that caused the condition to evaluate to true. As a change to a data member can cause more than one rule condition to evaluate to true, the action of a rule may be delayed from the time of the change that caused its condition to evaluate to true. If the data that caused the condition to evaluate to true is changed in the meantime, the rule's action will not be executed.

R++ ensures that rules do not execute on "old" data nor execute more than once on changed data. For example, if a relevant change (or construction) triggers two rules in a way that would satisfy both rule conditions, but the first rule to be evaluated and executed changes the original triggering data, thereby retriggering the same two rules, the second rule will eventually be evaluated on the newest data (and executed only if its condition is satisfied). Although the second was triggered twice in this scenario, it will be executed at most once and only on the newest data.

The actions of R++ rules are executed whenever possible under the above criteria. Thus, no rules are waiting to run if and only if there is no current collection of data that causes a rule's condition to evaluate to true for which the rule's action has not been run.

An Example Rule

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 provided that the person and spouse reside in the same country. The R++ code for this rule is given in Figure 2. This rule would be associated with the class Person, and would access two other objects in its condition, both belonging to Person. The rule would be given a name such as "children_spouse_dependents". In evaluations of the condition 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->spouse, for the second object).

```
class Person {
    int age;
    int salary;
    Country * residency;
    Person * spouse;
    Set_of_p<Person> children;
    Set_of_p<Person> dependents;
    . . .
    rule children_spouse_dependents;
};

rule Person::children_spouse_dependents {
    Person * mate = spouse &&
    residency == mate->residency &&
    Person * child @ children &&
    child->age < 18
=> mate -> insert_dependents(child);
}
```

Figure 2: children_spouse_dependents rule

The rule is active on all objects that belong to Person and responds to all changes that might cause its condition to be successfully evaluated. Thus it is one of the invariants associated with the class. Whenever an instance of Person is created, the rule children_spouse_dependents's condition will be evaluated with the new instance as root object. Whenever a person's spouse changes, the residency of a person with a spouse or that spouse changes, a person with a spouse residing in the same country gains a new child, or the age of a child of such a person is changed, the rule's condition will be re-evaluated on precisely all appropriate collections of objects. In effect, the rule guarantees that for all people who reside in the same country as their spouse does, that person's children under the age of 18 are dependents of the person's spouse.

Note that because the rule's condition uses the age of the child, it will be reevaluated whenever the child's age changes, and the rule's action will be executed when the age value is less than 18, This may result in the child being made a dependent of the spouse sev-

```
class Rectangle {
    int height, width, area;
    ...
    rule update_area;
};

rule Rectangle::update_area {
    height > 0 && width > 0
=> set_area(height * width);
}
```

Figure 3: update_area rule

eral times, once for each time the child's age changes. However, since the dependents is a set, such repetitions are benign.

Uses of Rules

Path-based rules can be used to enforce an invariant—something that should always be true. The simplest sort of invariant is an invariant on the data of a single object. The rule given in Figure 3 enforces such an invariant, here that a rectangle's area is equal to its height times its width. Rules on single objects are executed efficiently by the R++ rule-execution mechanism, meaning that maintaining such simple invariants can be effectively done by means of R++ rules.

A more-complex rule, such as the rules in Figures 4 and 2, can check conditions on more than one object and maintain an invariant based on these conditions. The action of the rule can also cause external activity, such as printing an informative message.

A rule declaratively represents an invariant and thus makes it easier to understand and modify. Further the rule is less prone to bugs than the equivalent collection of functions, as it does not have to be explicitly called in all the appropriate places. Invariant maintenance exploits rules' ability to react automatically to changes in objects.

R++ rules can also be used to detect situations that should never be true (constraint violations). For example, if it was not possible to turn off the call waiting and call forwarding features, the rule in Figure 4, modified to remove the portion that turns off call waiting and call forwarding, would serve as a constraint violation detection rule. Constraint violation also exploits rules' ability to react automatically to changes in objects.

It is also possible to use rules as "demons" to monitor for and react to important states. Rules are good "watch dogs" because they are triggered by every relevant change.

Of course it is possible to use R++ rules to express domain knowledge such as business policies, engineering rules, and situation-action heuristics—the sort of information that is often expressed in OPS-style rules. Such knowledge is often more clearly expressed in rules than in procedures, as has been demonstrated by the OPS community.

```
class Phone;

class Line {
    Phone * phone;
    Boolean call_waiting, call_forwarding;
    ...
    rule public_phone_test;
};

class Phone {
    PhoneType type;
    ...
    friend rule Line::public_phone_test;
};

rule Line::public_phone_test {
    call_waiting &&
    Phone * ph = phone &&
    ph->type == PUBLIC
=> cout << "Error: public phones cannot "
    << "have call waiting...";
set_call_waiting(false);
set_call_forwarding(false);
}
```

Figure 4: public_phone_test rule

Another use of R++ rules is in the place of member functions that would otherwise have to be called from many different places. Using rules as "automatic member functions" eliminates some of the burden of procedural control.

Yet another use of R++ rules is to exploit the "data-driven" or "event-driven" nature of rules to propagate change through or update the status of an object model. We are using R++ rules in this manner, among others, in a monitoring system for 4ESS telephone switches (Crawford *et al.* 1995). Two rules from this system that propagate alarms to dependent devices are shown in Figure 5. Note that these rules work for all objects that belong to the class Device, even those that also belong to subclasses such as Controller.

```
class Alarm { ...};

class Device {
    Set_of_p<Device> dependents;
    Set_of_p<Alarm> alarms;
    Set_of_p<Alarm> dependent_alarms;
    ...
    rule alarm_dependent_alarm;
    rule dependent_alarm_transitive;
};

class Controller : public Device { ...};

rule Device::alarm_dependent_alarm {
    Device * dependent @ dependents &&
    Alarm * alarm @ alarms
=> dependent->insert_dependent_alarms(alarm);
}

rule Device::dependent_alarm_transitive {
    Device * dependent @ dependents &&
    Alarm * alarm @ dependent_alarms
=> dependent->insert_dependent_alarms(alarm);
}
```

Figure 5: Rules on Device

Implementation Issues

R++ is implemented by means of an R++ to C++ translator, itself written in C++.[5] C++ is not an ideal language for our purpose. We would have preferred to make R++ largely a library of classes, thus requiring fewer extensions to the base language. However, C++ does not support the kind of control over object creation or destruction that such an architecture would require. It also does not provide monitoring of all accesses to object members. Therefore we have had to create an extension of the C++ language, as detailed above.

Even this extension process is less-than-ideal in C++. C++ is not type-safe—casts can change the typing of objects, violating encapsulation. Further, C++ allows pointers to object data members to be created, which can then be used to change the object. For C++ to be easy to extend, it should allow a comprehensive notification mechanism whereby a 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 proper code.

1. No casts can be performed on classes used by rules. (This restriction is just a stronger restatement of the normal C++ caveat that casts violate encapsulation and can result in incorrect execution.)

2. Changes to data members of classes that are used in rules must be made via stereotypic modification functions, as shown in the example rules. These modification functions are called `set_<member name>` for scalar data member and `insert_<member name>` and `remove_<member name>` for set-valued data members and are automatically generated by the R++ translator. It is generally considered good C++ programming style to use such modification functions.[6]

Violation of these restrictions may cause improper behavior of R++ rules. This is not an ideal solution, but is required because of the limitations of C++.

The algorithm for processing path-based rules is simpler and more efficient than the "inference engines" of pattern-matching rule languages in the OPS family since pointer-following is much less complex than join operations. For each data member monitored by a rule, the R++ translator adds a rule-specific data member that, during execution, contains backpointers to root objects from which the rule's path to the current object have been traced. (Root objects themselves to not need these pointers, of course.) When a relevant change is made to a data member of some object, each triggered rule will follow each of its backpointers to a root object and then evaluate its condition on all paths that can be formed from that root and that also pass through the changed data member of the changed object. Given n objects, r rules, and v variables in the largest rule rule condition, the worst-case heap-storage requirements for R++ are thus $O(rn^2)$ whereas the RETE algorithm (Forgy 1982) and its variants use space $O(rn^v)$. The algorithms used to process path-based rule are also much simpler than the algorithms that have been developed to support pattern-matching rules; so much so that there is no run-time interpreter for R++—rules simply compile into C++ procedures that are executed whenever a relevant change occurs.

The performance of the implementation of R++ relative to the pattern-matching rules of C5, a C-based

[5] The approach we have taken is similar to the approach used in the Ode (Agrawal & Gehani 1989) system, an object-oriented database management system that incorporates O++ (Dar, Agrawal, & Gehani 1993), an extension to C++.

[6] We could have translated assignments to data members into calls to the modification functions, but instead chose a method that emphasizes data encapsulation.

path length	*branching factor*	*# of leaves*	R++ *rate (for sets)*	*C5 rate*
0	NA	1	120000	2083
1	1	1	25575	1428
1	2	2	23831	1428
1	4	4	23301	1333
1	8	8	22310	1391
1	16	16	21399	1391
1	32	32	20164	1411
1	64	64	19418	1383
1	128	128	18078	1383
2	1	1	13797	1363
2	2	4	8065	1333
2	4	16	4854	1317
2	8	64	2849	1280
2	16	256	1524	1024
2	32	1024	682	640
2	64	4096	309	244
2	128	16384	138	42
3	1	1	9362	1200
3	2	8	3196	1142
3	4	64	1024	1200
3	8	512	264	682
3	16	4096	57	107
3	32	32768	13	13

Table 1: Rule-firing rates for R++ and C5

superset of OPS5 using a RETE algorithm, is shown in Table 1. Rates are given in rule firings per second on a Sun SPARCstation 10.

In each performance measurement a tree[7] of objects of a specified depth (path length) and breadth (branching factor) was created; a path-length of n means that $n+1$ objects are in the path. The tree is monitored by a single rule that follows a path from the root object to a leaf object.

To test the worst-case for R++, all triggering changes were made in leaf objects. R++ outperforms C5, especially with small path lengths and low branching factors, where it is much faster. This is largely due to the elimination of the bookkeeping that is required for the RETE algorithm.

[7]Trees can be implemented in C++ with either lists or sets, depending on whether order is important or not. Table 1 shows only the R++ rate for sets because trees in C5 are inherently unordered since they arise from joins among working memory elements. If order *is* important, it is easy to obtain in R++ simply by using lists instead of sets, though there is a performance penalty to be paid. In C5 and other OPS derivatives it is much more difficult and time-consuming to process a tree in an ordered way.

Related Work

Sophisticated programming constructs have a long history in AI. Just about any frame system has had a simple version of rules, such as the attached procedures of FRL (Roberts & Goldstein 1977) or the active values of LOOPS (Bobrow & Stefik 1981). These attached procedures are generally directly triggered by the modification or access of a single data member, and do not have a condition that can refer to multiple objects as the rules of R++ have.

Full forward-chaining rules exist in many AI-related languages and systems, including, of course, OPS (Cooper & Wogrin 1988). The rules of OPS are more general than the rules of R++, and do not fit into the object-oriented paradigm as well as the rules of R++ do. OPS-style rules have different semantics than R++ rules, involving a conflict resolution scheme with rule priorities. The above comments also apply to the rules in more-recent AI-related languages and systems such as ART-IM (Inf 1987), CERS (Miranker *et al.* 1993), ILOG rules (Albert 1994), RAL/C++ (Forgy 1994), and Rete++ (Hal 1993). Further, none of these systems has as close an integration with C++ as R++ has.

Of course, the most-closely related system to R++ is Algernon (Crawford & Kuipers 1991; Crawford 1990). R++ moves the path-based rules of Algernon into the object-oriented language C++ and integrates the rules into the C++ object system.

The work in active object-oriented databases has some similarities to R++. In particular, integrity constraints and triggers in object-oriented databases are somewhat similar to the rules of R++. For example the triggers of ODE (Gehani & Jagadish 1991; 1996) allow for the execution of C++ code when a condition is satisfied on an object. The difference between ODE triggers and R++ rules is that an ODE trigger on an object is executed only when a method is run on that object—there is no mechanism for deferring the execution of the trigger action until a condition involving other objects is satisfied.

Other active database work is less similar to R++. The event-condition-action rules in many active databases, including POSTGRES (Potamianos & Stonebraker 1996) and Starburst (Widom 1996), trigger on specific events in the database, such as adding a tuple to a relation, and not on a boolean condition involving the data members of various objects (but then allow a condition to filter the rule action). The condition-action rules in Ariel (Hanson 1996) and a few other active database systems are global rules with conditions similar to the conditions in OPS-style production rules, and thus are more general (in that they do not need to follow access paths) but less object-oriented than the rules in R++.

The rules in R++ are not at all like the rules in deductive databases, even object-oriented deductive data bases such as Coral++ (Srivastava *et al.* 1993). The rules in deductive data bases are run to satisfy queries and so have a very different purpose.

Future Work

There are a number of issues that have not been addressed in the current R++ implementation, but that are candidates for future versions of R++.

First, the current version of R++ runs rules after every change to monitored data. It would be useful to be able to defer the running of rules occasionally, treating a group of changes as an atomic batch. There are semantic problems with this, such as how to treat changes to monitored data that are undone inside the batch.

Second, the current version of R++ works entirely on in-core data. As there is considerable interest in active databases, it would be interesting to combine the R++ approach to rules with an object-oriented database. Since R++ rules are simpler to implement than the rules in most active databases, such a system should be able to support more rules without having to add triggers to control the activation of rules.

Third, R++ interacts badly with file-dependency recompilation of large C++ programs. The header files resulting from R++ are larger than standard C++ header files and are more interconnected, so recompilation takes even more time than with standard C++ programs. A better mechanism for recompiling C++ programs is required to really solve this problem, but some changes to how R++ generates header files would help somewhat.

Fourth, R++ requires that rules branch only on data in objects. This was done so that changes to the branch structure would be easy to detect. Branching on an expression is possible, under the same caveat as given above for data used by functions, but keeping track of the branching is more difficult.

Summary

As a "reactive" member function, an R++ rule is a new kind of class member that adds data-driven behavior to a class. Data-driven rules are valuable in object-oriented programs as an aid to maintaining model integrity and, more generally, as a way to express and apply situation-action directives to a domain model. Rules can make programs clearer because a single rule expressing a multi-object constraint or invariant or policy can replace several variants of the same logic scattered throughout procedural code. Also, rules relieve the programmer of the burden of explicit procedural control as the rules are triggered automatically by relevant change (and construction).

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 conditions, such rules have a more localized effect than OPS-style rules and can therefore be expected to be more predictable to the programmer. Since each path-based rule is associated with a class (like a data-driven method), it can be inherited (and overridden) in the same way as methods are in the host object-oriented language. With a syntax similar to C++ member functions and a semantics like "automatic member functions", R++ rules are relatively easy to learn and apply.

In essence, path-based rules define a kind of "member rule" that naturally extends the object-oriented framework. This allows production rules to be easily merged into the object-oriented paradigm. This synthesis is important in making AI techniques accessible to the the wider computer-science community.

To encourage further experimentation and research in multi-paradigm programming languages, R++ is available free to research institutions. For further information, including the *R++ User Manual*, see the R++ home page on the world-wide web at `http://www.research.att.com/sw/tools/r++` or send e-mail to `r++@hrmaple.hr.att.com`.

References

Agrawal, R., and Gehani, N. H. 1989. Ode (object database and environment): The language and the data model. In *Proceedings of the 1989 ACM SIGMOD International Conference on Mangement of Data*, 36–45. Association for Computing Machinery.

Albert, P. 1994. ILog Rules, embedding rules in C++: Results and limits. In EOOPS94 (1994).

Bobrow, D. G., and Stefik, M. 1981. The Loops manual. Technical Report KB-VLSI-81-13, Knowledge Systems Area, Xerox Palo Alto Research Centers.

Cooper, T. A., and Wogrin, N. 1988. *Rule-Based Programming with OPS5*. San Mateo, California: Morgan Kaufmann.

Crawford, J. M., and Kuipers, B. 1991. Negation and proof by contradiction in access-limited logic. In

Proceedings of the Ninth National Conference on Artificial Intelligence, 897–903. American Association for Artificial Intelligence.

Crawford, J.; Dvorak, D.; Litman, D.; Mishra, A.; and Patel-Schneider, P. F. 1995. Device representation and reasoning with affective relations. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, 1814–1820. International Joint Committee on Artificial Intelligence.

Crawford, J. M. 1990. *Access-Limited Logic—A language for knowledge representation.* Ph.D. 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, S.; Agrawal, R.; and Gehani, N. H. 1993. The O++ database programming language: Implementation and experience. In *Proceedings of the 9th IEEE International Conference on Data Engineering.*

1994. *Position Papers for the OOPSLA'94 Embedded Object-Oriented Production Systems Workshop (EOOPS).*

Forgy, C. L. 1982. RETE: A fast algorithm for the many pattern/many object pattern matching problem. *Artificial Intelligence* 19:17–37.

Forgy, C. L. 1994. RAL/C and RAL/C++: Rule-based extensions to C and C++. In EOOPS94 (1994).

Gehani, N. H., and Jagadish, H. V. 1991. Active database facilities in ode. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases*, 327–336.

Gehani, N. H., and Jagadish, H. V. 1996. Ode as an active database: Constraints and triggers. In Widom and Ceri (1996). 207–232.

The Haley Enterprise. 1993. *Rete++: Seamless Integration of Rules and Objects Using the Rete Algorithm and C++.*

Hanson, E. N. 1996. The Ariel project. In Widom and Ceri (1996). 63–86.

Inference Corporation. 1987. *Art Reference Manual.*

Miranker, D. P.; Burke, F. H.; Steele, J. J.; Haug, D. R.; and Kolts, J. 1993. The C++ embeddable rule system. *International Journal on Artificial Intelligence Tools* 2(1):33–46.

Potamianos, S., and Stonebraker, M. 1996. The POSTGRES rule system. In Widom and Ceri (1996). 43–61.

Roberts, R. B., and Goldstein, I. P. 1977. The FRL manual. AI Memo 409, Artificial Intelligence Laboratory, Massachusetts Institute of Technology.

Srivastava, D.; Ramakrishnan, R.; Sudarshan, S.; and Seshadri, P. 1993. Coral++: Adding object-orientation to a logic database language. In *Proceedings of the Nineteenth International Conference on Very Large Databases.*

Stroustrup, B. 1991. *The C++ Programming Language.* Reading, Massachusetts: Addison Wesley, second edition.

Widom, J., and Ceri, S., eds. 1996. *Active Database Systems: Triggers and Rules for Advanced Database Processing.* San Francisco, California: Morgan Kaufmann.

Widom, J. 1996. The Starburst rule system. In Widom and Ceri (1996). 87–109.