# R++: Using Rules in Object-Oriented Designs

**Anil Mishra, Johannes P. Ros**
**Anoop Singhal, Gary Weiss**
AT&T Network and Computing Services

**Diane Litman**
**Peter F. Patel-Schneider**
AT&T Labs—Research

**Daniel Dvorak**
Jet Propulsion Laboratory

**James Crawford**
i2 Technologies

## Abstract

System requirements often express some of the behavior of a domain model in the form of policies, constraints, invariants, and rules. Although an object-oriented approach helps with the design of this model, the implementation is often cluttered with details necessary to enforce the model's policies and constraints. As a result, the code is more prone to error and is more difficult to maintain since it tends to obscure the model's fundamental structure. R++, a rule-based extension to C++, allows developers to express policies and constraints more clearly in the form of path-based rules. Since the rule evaluation mechanism is automatic, R++ rules enable developers to focus on what to do when a rule evaluates to TRUE, rather than on how and when these evaluations should be carried out. It is the absence of this mechanism in the source code, and the use of these rules in an object-oriented implementation, that makes the code easier to maintain and makes for a clearer reflection of the underlying model.

## 1 Introduction

One of the purposes of a programming language is to provide the right level of abstraction, freeing programmers from details and allowing them to work closer to the problem at hand. Determining the right programming language for a task is thus not a matter of determining whether the language can be used for the task—given adequate resources any solvable task can be solved in just about any programming language—but instead involves determining which language provides the appropriate abstractions to solve the problem at hand.

Object-oriented programming with its rich modeling capability has taken a large step away from implementation details and towards problems. Nevertheless, even object-oriented programmers must still spend too much time on low-level implementation details. This is especially true when the problem is to create and maintain a model of a changing collection of interdependent objects. While the creation of such a model is the province of object-oriented programming (classes and objects hold and organize the model, while local changes are encapsulated and hidden in methods), object-oriented programming provides little or no help for changes that affect many objects in the model. In such cases there is still far too much of a gap between what the programmer wants to say and what he or she has to say.

This need not be the case as there are recent advances in programming languages that both capitalize on the strengths of object-oriented programming and eliminate the need for many types of low-level programming. For example, the idea of integrating rules into object-oriented languages has recently been gaining currency. Such an integration allows many tasks in an object-oriented design (e.g., those that maintain model integrity) to be performed automatically, rather than with explicit procedural invocation. R++, a path-based rules extension to C++, is one such integration [1; 3]. R++ allows programmers to seamlessly incorporate data-driven rules into their C++ classes.

The lessons presented in this report were derived from our experience in applying R++ to a large telephone-switch monitoring application. This application is summarized in the next section. However, the key points in this report are not about the application but rather about how path-based rules can help in object-oriented designs. Thus, we will generalize our lessons by using a seemingly simple example, not related to alarm monitoring, to illustrate the points that we particularly want to convey:

- R++ supports more of a one-to-one mapping from requirements to code,

- R++ encourages a what-rather-than-when style of thinking, and
- R++ more easily manages asynchronous information flow among objects.

## 2 Monitoring Alarms in a Telephone Switching System

Our application is an alarm monitor for the 4ESS® telephone switching system. The 4ESS handles most AT&T domestic long-distance calls. The alarm monitor performs real-time monitoring of alarms and other informational messages from the 4ESS, e.g., to correlate alarm messages coming from the various components, ignore transient alarms, recognize chronic problems, run diagnostics, and signal problems requiring human attention. The monitoring system takes input in the form of alarm messages and produces output in the form of warning and action messages to field technicians.

A system implemented in C was originally developed for monitoring the 4ESS switches. Due to a wide variety of problems with this approach (e.g. software maintenance), in 1990 the system was redesigned and reimplemented using a rule-based paradigm. However, as this version of the alarm monitor grew in size and complexity, it too exhibited several important limitations. For example, the alarm monitor had no domain model, and only used rules to implement shallow, heuristic-style reasoning.[1]

To address these limitations, a redesign of the alarm monitor was begun in 1992. In this redesign, there is an explicit model of the network elements, and reasoning is with respect to this model. Alarm messages from the switch, in conjunction with an explicit model of the switch, trigger the dynamic creation of relevant portions of the 4ESS. Given this instantiation of the 4ESS, general purpose diagnostic rules then determine the root cause of the alarm. Further details regarding our approach to alarm monitoring can be found in [2].

With the rise of object-oriented programming and our desire to represent models of the network, it became clear that our redesign should be implemented using a language that integrated both our previous rule-based approach and more recent object-oriented advances. After determining that no existing commercial products met our needs, design and implementation of R++—our rule-based extension to C++—was begun in 1994.

R++ is a small extension to C++ that adds a single new programming construct: a rule. Rules are data-driven since they are triggered automatically by changes to the data that they monitor. In contrast to most other rule languages, rules in R++ are path-based rather than based on pattern-matching. (Path-based means that the condition of an R++ rule must traverse inter-object pointers.) This enables rules to be treated as a new kind of class member that adds data-driven object-centered behavior to a class. For more information about R++, see [1; 3], 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.

Using R++ to implement our redesigned alarm monitor allows C++ to be used to represent the domain model (the hierarchical structure of components in the switch and the functional relationships between them). The path-based rules of R++ are then used to: (1) maintain integrity of the domain model by enforcing invariants and detecting constraint violations, (2) propagate alarm information through the model, and (3) monitor for conditions that require technicians to be alerted. Note that because R++ rules are tightly integrated with C++ object-oriented class mechanisms—which are used to define the domain model—it is natural to write rules that operate on the model. In fact, because R++ rules are path-based, rules *must* be written using relations that are explicitly defined in the domain model.

Our redesign of the alarm monitor and the associated R++ implementation has yielded several important benefits, including reduction of total headcount for software development and improvement with respect to the end-to-end process. Since R++ relieves programmers from unnecessary bookkeeping, it enables system engineers and software developers to concentrate on solving the real problem. In contrast, when using C++, much time is spent dealing with low-level artifacts of C++.

The remainder of this report elaborates on these benefits. To make our points more accessible, we abstract from our alarm monitoring application to generalize our ideas and eliminate domain-specific jargon. To focus our lessons, we illustrate our points using a single type of task, namely, the specification of situation-action requirements.

## 3 Situation-Action Requirements

Many system requirements are of the form "when situation x occurs, perform action y." Such requirements, which define automatic behavior within a domain model, appear variously as invariants, business policies, engineering rules, domain laws, constraints, audits, and state transitions. At first glance, such requirements seem easy to implement; just insert tests for situation x in all the right places, and if the test is satisfied, perform action y. The example in the following sections illustrates just how error-prone this seemingly simple approach can be when implemented in a purely procedural language such as C++, and how much simpler such an approach is when implemented using the path-based rules of R++.

[1] Within AT&T, C++ has also been used to develop systems with similar functionality. However, an important limitation of these systems is their lack of rule-based reasoning.

```
...

class Line {
  Station * stationp;
  Boolean call_waiting,
          call_forwarding;
  ...};

class Station {
  PhoneType type;
  ...};

...
```

Figure 1: A partial object model for telephone provisioning

### 3.1 Example: Using C++ to Enforce Policies

When a telephone company "provisions" a telephone line (such as the one serving your home), they activate any special services the customer has requested, such as "call-waiting" and/or "call-forwarding". These two services, however, should not be activated if the telephone line is connected to a public pay phone; it's a business policy.

As shown in Figure 1, an object model for this domain contains two kinds of objects: Line and Station ("Line" refers to a port on the telephone switching system and "Station" refers to the equipment connected to the line, such as a public pay phone). Reflecting the physical connection, there is a pointer from a Line object to a Station object. To enforce the policy that a pay phone cannot have call-waiting or call-forwarding, code must be placed in four different access functions, as shown in Figure 2 using C++ code. This code enforces the policy no matter what order events occur within the application.

Although this code is correct and relatively simple, it is error-prone in two important ways. First, because the implementation of the policy resides in four different functions, and because each function sees the policy from a slightly different perspective, the implementation is vulnerable to errors of omission and duplication. Second, and more seriously in terms of long-term maintenance of code, the original policy does not "jump out" from reading the code; it takes four separate functions to implement one very simple policy. Although the policy and associated code may have been clear to the initial programmer, the poor maintenance programmer who has to modify the code some day (possibly because the policy has changed) has to find all the places where the policy was enforced. In this case the four places are divided between two classes, making the task even more prone to errors of neglect. Further, a programmer who has to modify one of the four functions for other reasons may inadvertently break the policy-enforcement. In short, the source of all these difficulties is that the programming language does not provide a single programming construct for expressing and enforcing situation-action directives.

### 3.2 Example: Using Rules in R++ to Enforce Policies

R++ solves many of problems discussed above by extending C++ with a single new programming construct: a data-driven, path-based rule. Rules allow the direct representation of dependencies between related objects, without worrying about how to do the bookkeeping required, how to divide the dependency between various classes and objects, or when to trigger the processing required to implement the dependency.

Figure 3 illustrates how the pay-phone policy is expressed in a single R++ rule. This rule (and every rule) is associated with a class, has a rule name, and has a condition and an action. In particular, the rule in Figure 3 is named "public_phone_check" and is associated with the class "Line" from Figure 1. The semantics of all rules are that whenever the condition becomes true the action is executed. Thus "public_phone_check" says that "whenever a line object has call-waiting or call-forwarding, and it has an associated station, and that station is a pay-phone, then report the mistake and reset the call-waiting and call-forwarding flags."

Note how much shorter the R++ code in Figure 3 is compared to the C++ code in Figure 2. This is because R++ turns the rules into several C++ functions that run the rule under various conditions, and arranges to trigger the rule in response to appropriate situations. With more complicated rules R++ has to do even more, including keeping track of rules whose conditions are partially satisfied.

There are other obvious and not-so-obvious benefits of using rules. One obvious benefit is that there is now a one-to-one mapping between a policy and its code. This makes it not only easier for the original programmer to encode the policy but also much easier for any subsequent programmers to see and understand such policies. This kind of clarity pays real dividends in the long maintenance phase of typical applications.

A second benefit of rules is that they are triggered automatically by any changes in the values of variables monitored in the rule condition. As long as all monitored variables are modified through R++-defined access functions (e.g., "set_call_waiting"), R++ ensures that the appropriate rules are triggered in response to changes. This means that the programmer of a rule does not have to worry about where and when to test the rule condition; instead he/she can focus on what condition the rule should watch for and what action to take when it occurs. Not only does this simplify the thought process of the programmer, it also eliminates clutter in the application logic. It also makes programs more robust because, while a programmer might ne-

```
void Line::set_call_waiting(Boolean b)
{  call_waiting = b;
   if (call_waiting && (stationp->type == Pay_Phone))
   {   call_waiting = false;
       cout << ``Error: ...'';}}

void Line::set_call_forwarding(Boolean b)
{   call_forwarding = b;
    if (call_forwarding && (stationp->type == Pay_Phone))
    {  call_forwarding = false;
       cout << ``Error: ...'';}}

void Line::set_stationp(Station *p)
{  stationp = p;
   if ((call_waiting || call_forwarding) &&
       (stationp->type == Pay_Phone))
   {  call_waiting = call_forwarding = false;
      cout << ``Error: ...'';}}

void Station::set_type(int t)
{  type = t;
   if ((type == Pay_Phone) &&
      (linep->call_forwarding || linep->call_waiting))
      linep->call_waiting = linep->call_forwarding = false;}
```

Figure 2: C++ code for enforcing the policy about pay phones

glect to insert a test somewhere, the compilation of rules by R++ guarantees that all rule triggering is inserted automatically.

A third benefit of R++ is that it reduces the conceptual complexity of the information flow among the objects. Message passing in object-oriented programming requires the programmer to be cognizant of the information flow, which may exhibit complex bi-directional relationships between objects. In event-driven models, these relationships can often be separated into requests and (asynchronous) responses. In the procedural object-oriented paradigm, it is the responsibility of the service provider to send the responses (events) back to the corresponding requester, which requires back-pointer maintenance. R++ provides this back-pointer mechanism automatically as a result of the requester monitoring the service provider. By relieving the programmer of backpointer bookkeeping, R++ effectively reduces the perceived complexity of the information flow to a directed acyclic graph.

# 4 Effect of Rules on System Engineering

Object-oriented design creates a model of a problem domain—a mini-world, in effect, that operates according to its own laws and policies and constraints. For the model to work correctly, it's essential that such laws and policies and constraints be consistently enforced in an application. Since R++ makes that easy to do, we have been encouraging our systems engineers to try to clearly identify three categories of rule-like requirements: (1) "things that must always be true," such as invariants, laws of nature, business policies, and engineering rules; (2) "things that must never be true," such as constraint violations and logical impossibilities; and (3) "expected reactions," such as propagation of information, state transitions, warnings, and logical consequences of changes. Interestingly, it is easier to do this when one is first learning the domain; the more experienced systems engineers have internalized so much domain knowledge that they can't articulate it completely.

# 5 Caveats

We have noticed three disadvantages in using R++. The first is that compilation times have increased. Part of the increase is in the time needed to run the R++ translator, but the larger part is in increased C++ compilation time. The C++ code emitted by the translator is larger and the emitted code uses several template classes; template instantiation is notoriously slow in some C++ compilers. A second (short-term) disadvantage is that it takes time for programmers to adjust to the paradigm shift, where data-driven computation can be used to express some things more clearly than in

```
rule Line::public_phone_check {
   (call_waiting || call_forwarding) &&
   Station *s = stationp  &&
   s->type == PAY_PHONE
=>
   cout << ``Error: ...'';
   set_call_waiting(false);
   set_call_forwarding(false);}
```

Figure 3: R++ rule for enforcing the policy about pay phones

conventional object-oriented programming. The third disadvantage is that object-oriented development tools (and methodologies) don't yet incorporate the idea of rules (or situation-action directives), so the current tools provide no help in working with and debugging rules.

# 6 Conclusion

Our experience using R++ in an industrial application encourages us in the belief that the addition of path-based rules brings significant benefits to object-oriented programming. The purpose of a programming language is to reduce the gap between what you want to say and what you have to say. R++ reduces the gap between what we want to say (invariants, constraints, business policies, engineering rules, etc.) and what we have to say (one rule instead of multiple functions). The most important benefits of R++ are, in our experience, the one-to-one mapping from requirements to code, the what-rather-than-when style of thinking that it encourages, and the easier way of managing asynchronous information flow among objects.

# Acknowledgments

We would like to thank our management, especially Ron Brachman, Pramod Warty, Raj Dube, and Jennifer Thien for their support of this project. Finally, we thank Walid Saba for the implementation of some of these ideas using R++.

# References

[1] James Crawford, Daniel Dvorak, Diane Litman, Anil Mishra, and Peter F. Patel-Schneider. Path-based production rules. In *Proceedings of the Embedded Object-Oriented Production Systems workshop of OOPSLA-94*, October 1994.

[2] James Crawford, Daniel Dvorak, Diane Litman, Anil Mishra, and Peter F. Patel-Schneider. Device representation and reasoning with affective relations. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1814–1820. International Joint Committee on Artificial Intelligence, August 1995.

[3] James Crawford, Daniel Dvorak, Diane Litman, Anil Mishra, and Peter F. Patel-Schneider. Path-based rules in object-oriented programming. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 490–497, Portland, Oregon, August 1996. American Association for Artificial Intelligence.