Using Rules in Object-Oriented Designs

Daniel Dvorak
Anil Mishra
Johannes P. Ros
Anoop Singhal
Gary Weiss
AT&T Network & Computing Services
Diane Litman
AT&T Research

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.

Category: Experience Report

Contact: Anil Mishra, room 2C-053
AT&T Network & Computing Services
480 Red Hill Road
Middletown, NJ 07748

email: anil@hrmaple.hr.att.com
phone: (908) 615-4552
fax: (908) 615-5579

Note to Reviewers

The lessons presented in this report were derived from our experience in applying R++ to a large telephone-switch monitoring application. 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, to make the lessons more accessible to a general audience, we have generalized the ideas and eliminated domain-specific jargon.

For those who wish to know more about this application, its basic activity is the analysis of alarms from a largely self-checking switching system. This application uses an object model to represent the hierarchical structure of components in the switch and the functional relationships among those components. The application employs about 50 rules, used in three main capacities: (1) to maintain integrity of the domain model by enforcing invariants and detecting constraint violations, (2) to propagate alarm information through the model, and (3) to monitor for conditions that require technicians to be alerted. The application is driven by alarm messages and timeout events which trigger all monitoring and diagnosis activities [Crawford et al., 1995].

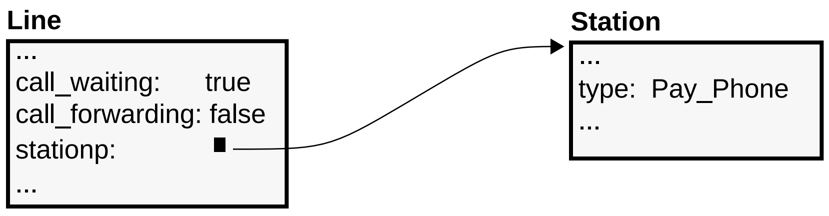
Situation–Action Requirements

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 section illustrates just how error-prone this seemingly simple approach can be.

Example: Enforcing a Policy

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.

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 below in C++ code. This code enforces the policy no matter what order events occur within the application. Although this code is correct and

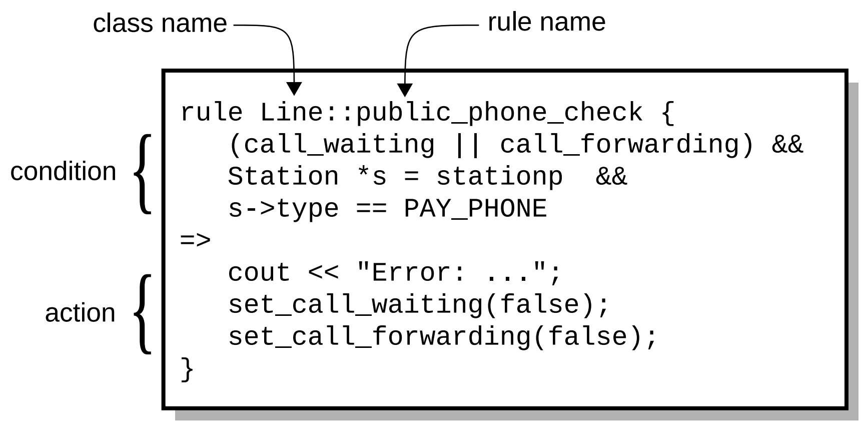
```
// Code for enforcing the policy about pay phones.

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;
}
```

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.

Using Rules in R++ to Enforce Policies

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, and thus are well-suited to enforcing situation–action requirements such as policies and invariants and constraints. In contrast to other rule languages, rules in R++ are *path-based* rather than pattern-matching. This enables rules to be treated as a new kind of class member that adds data-driven object-centered behavior to a class [Crawford et al., 1996b]. For more information about R++, see http://www.research.att.com/orgs/ssr/people/dvorak/r++.

The example below illustrates how the pay-phone policy is expressed in a single R++ rule. This



```
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);
}
```

rule (and every rule) has a condition and an action. The semantics are that whenever the condition becomes true the action is executed. Literally, this rule says "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."

There are obvious and not-so-obvious benefits of using rules to express and enforce policies like this. 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, 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 neglect 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 back-pointer bookkeeping, R++ effectively reduces the perceived complexity of the information flow to a directed acyclic graph.

A final benefit of R++ is that rules are organized by the type hierarchy and are inherited and overridden in a way similar to virtual functions. For instance, in the pay-phone policy example, the public_phone_check rule is associated with the `Line` object. This organization makes it easier to find, understand, and maintain the rules. Existing rule-based systems, such as OPS5, provide no such organizing principle for rules.

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.

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 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.

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[1]. 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 three 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. We also owe a lot to James Crawford and Peter Patel-Schneider for shaping the R++ language and implementing the R++ translator. Finally, we thank Walid Saba for the implementation of some of these ideas using R++.

References

[Crawford, 1990] James Crawford. Access-Limited Logic-A language for knowledge representation. Ph.D. thesis, department of Computer Sciences, The University of Texas at Austin, 1990. Also published as technical report AI 90-141, Artificial Intelligence Laboratory, The University of Texas at Austin.

[Crawford et al., 1994] James Crawford, Daniel Dvorak, Diane Litman, Anil Mishra, and Peter F. Patel-Schneider. Path-based Production Rules. *Proceedings of the OOPSLA 94 workshop on Embedded Object-Oriented Production Systems (EOOPS),* F. Pachet, editor, Portland, Oregon October 1994.

[Crawford et al., 1996a] James Crawford, Daniel Dvorak, Diane Litman, Anil Mishra, and Peter F. Patel-Schneider. Path-Based Rules in Object-Oriented Programming. Submitted to AAAI-96.

[Crawford et al., 1996b] James Crawford, Daniel Dvorak, Diane Litman, Anil Mishra, and Peter F. Patel-Schneider. R++: Adding Path-Based Rules to C++. Submitted to OOPSLA-96.

[Crawford and Kuipers, 1991] James M. Crawford and Benjamin Kuipers. Negation and proof by contradiction in access-limited logic. In *Proceedings of the Ninth National Conference on Artificial Intelligence,* pages 897-903, Anaheim, California, July 1991. American Association for Artificial Intelligence.

[Crawford et al., 1995] James Crawford, Daniel Dvorak, Diane Litman, Anil Mishra, and Peter F. Patel-Schneider.Device representation and reasoning with affective relation. In *Proceedings of the*

1. Thanks to Brian Kernighan for this insightful comment.

14th International Joint Conference on Artificial Intelligence (IJCAI-95), pages 1814-1820, Montreal, August 1995.